

Python for (Absolute) Beginners

A Practical Introduction to Modern Python with Simple Hands-on Projects

Harry Yoon

Version 1.6.2, 2022-09-07

Copyright

Python for Beginners:

A Practical Introduction to Modern Python

© 2021-2022 Coding Books Press

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Published: November 2021

Harry Yoon
San Diego, California

Preface

Slow and Steady Wins the Race.

Python is one of the most popular programming languages. In fact, it is widely used just about anywhere programming is done. People use Python to automate the system administration tasks. People use Python to build their Websites and Web applications. People build robots and controllers using Python. And, Python is now becoming *the* language of choice for many people to do machine learning and data science.

Python is a "high level" programming language. It is one of the most favorite languages among the people who are just starting to learn programming. It is easy to learn and use, and yet it provides enough complexity and flexibility. As a matter of fact, Python is one of the *most powerful* modern programming languages.

If you are new to programming, then this book will guide you through the initial steps in your journey to the wonderful world of programming in Python. If you have some experience with programming, then this book will give you a quick taste of the "modern Python", covering the most recent releases, up to 3.10 and 3.11, as of this writing.

Python for Beginners will not make you the "master" of Python, as many beginner's books claim. Many programming books put too much emphasis on the programming language syntax and a laundry list of the language features, and not enough on the *real programming*.

This book takes a rather different and unique approach. Throughout the book, we will work on one main programming project. In this book, we will build a Python version of the game of Rock Paper Scissors. The choice of the particular project that we use is, however, not that important. In the process of building the game program, we will cover all the basic elements of software development in Python (some, with enough theoretical depth to be useful even to the more experienced programmers).

Unlike the vast majority of the books targeted to the programming beginners, this book will make you *think* and *do*. If you are looking to gain some quick and superficial knowledge only, this book is not for you. ☺

The book starts from the absolute basics of programming. And, by the end of the book, you will have built a complete software that fully functions. (In fact, a few times over.) This will give you a sense of accomplishment and the motivation to learn more.

After learning the basics of Python programming using this book, you will be ready to pursue the more difficult topics in Python programming. And, above all, you will likely be eager to do more programming. If so, then this book has done its job.

Good luck!

Revision 1.6.2, 2022-09-07

Table of Contents

Preface	2
Introduction	11
Modern Python	11
Book Organization	12
Absolute Basics	15
1. Hello Monty Python	16
1.1. The First Step	16
1.2. The Project	18
2. Development Environment Setup	19
2.1. Command Line Interface (CLI)	20
2.2. Python Installation	20
2.3. Visual Studio Code	23
2.4. Python Extension	25
2.5. Test Program	26
2.6. How to Run Python Programs	29
2.7. Code Review	31
2.8. Type Annotations	32
2.9. Summary	34
3. Interactive Tour of Python	35
3.1. Python REPL	35
3.2. Basic Concepts of Programming	38
3.3. Summary	44
4. Numbers, Strings, and More	46
4.1. Starting Python Interactive Session	46
4.2. Numbers in Python	47
4.3. Builtin <code>type</code> Function	48
4.4. Errors	49

4.5. Expressions	50
4.6. Interactive vs Non-Interactive Modes	51
4.7. Builtin <code>print</code> Function	52
4.8. The <code>None</code> Object	54
4.9. Boolean Expressions	55
4.10. Dynamic Typing	57
4.11. Builtin <code>bool</code> Function	58
4.12. Simple and Compound Statements	60
4.13. Conditional Statement	62
4.14. Strings in Python	63
4.15. String Concatenations	64
4.16. Ending Python Interactive Session	65
4.17. Summary	66
5. Tuples, Lists, and Some Inspirations	68
5.1. Complex Types	68
5.2. Tuple Literals	69
5.3. Expression List	70
5.4. Tuple Type	71
5.5. List Literals	74
5.6. List Operations	77
5.7. "Names" in Python	81
5.8. Assignment	83
5.9. Slicing	86
5.10. Sorting	89
5.11. Help!!	91
5.12. Inspirations	93
5.13. Summary	94
6. Review - Basics	96
6.1. Questions	97

6.2. Exercises	97
Rock Paper Scissors Project	99
7. Hello Rock Paper Scissors!	100
7.1. Working on a Project	100
7.2. Let's Play Rock Paper Scissors	101
8. Software Design	104
8.1. Deconstructing Rock Paper Scissors	105
8.2. Tasks	107
9. Project Setup	108
9.1. Workspace	108
9.2. Virtual Environments	110
9.3. Package Install	113
9.4. Source Control System	115
Imperative Programming	120
10. Main Project - Rock Paper Scissors	121
10.1. Rock Paper Scissors	122
10.2. Import Statement	127
10.3. Function Definition	131
10.4. Comparison Operators	135
10.5. The <code>if</code> Statement	137
10.6. Builtin <code>input</code> Function	142
10.7. Variables/Names	144
10.8. Rules on Names	148
10.9. Naming Conventions	149
10.10. Scopes	151
10.11. String Methods	153
10.12. Random Module	155
10.13. Boolean Operators	157
10.14. Lines in Python	160

10.15. Error Handling	162
10.16. Putting It All Together	163
10.17. Code Review	169
11. Lab 1 - Expressions and Statements	173
11.1. Echo	173
11.2. Dice Rolls	174
11.3. Is It Positive?	174
11.4. To Uppercase	174
11.5. Random Letters	175
11.6. Random Arithmetic	175
11.7. Can I Buy a Vowel?	175
11.8. All True or Not	176
11.9. Spade, Heart, Diamond, or Clubs	176
11.10. Random Suit	176
11.11. The Same Suit Or Not	176
11.12. Rock Paper Scissors	177
Procedural Programming	178
12. Rock Paper Scissors - The Sequel	179
12.1. Python Modules	188
12.2. Python Packages	193
12.3. Tuple Unpacking	196
12.4. Function Definitions	199
12.5. Function <code>def</code> with Type Annotations	203
12.6. Expression Statements	205
12.7. Doc Strings	207
12.8. Ellipsis (<code>...</code>)	209
12.9. <code>random.choice()</code>	210
12.10. Sequence Replication	212
12.11. f-String Expressions	213

12.12. Conditional Expressions	216
12.13. "States"	217
12.14. For Range Loop	219
12.15. While Loop	223
12.16. Error Handling	226
12.17. Putting It All Together	233
12.18. Code Review	242
13. Lab 2 - Functions, Loops, and More	244
13.1. Sum	244
13.2. Product	245
13.3. Filtered Sum	245
13.4. Singular vs Plural Nouns	246
13.5. Power Operator	246
13.6. Tuple Parameter	247
13.7. Reverse a List	247
13.8. Rock Paper Scissors	249
13.9. How Many Rounds?	249
13.10. Best of Seven	250
Object Oriented Programming	251
14. Rock Paper Scissors - The Finale	252
14.1. Modules and Packages	261
14.2. Backslashes	264
14.3. Optional and Union Types	266
14.4. Boolean Context	269
14.5. Recursion	271
14.6. Object Oriented Programming (OOP)	276
14.7. Class	277
14.8. Class Objects	279
14.9. Constructors	280

14.10. Class Variables	284
14.11. Instance Objects	285
14.12. Instance Variables	287
14.13. Instance Methods	288
14.14. Private Members	290
14.15. Dunder Attributes	291
14.16. Inheritance	293
14.17. Polymorphism	302
14.18. Truth Values	306
14.19. Enum	307
14.20. Match Statement	310
14.21. Dictionary	316
14.22. Putting It All Together	322
14.23. Code Review	326
15. Lab 3 - OOP and Other Modern Features	328
15.1. Days of the Week	328
15.2. I'll Be Going	328
15.3. Playing Cards	329
15.4. Length Function	329
15.5. Sum Function	330
15.6. Product Function	331
15.7. String Length Comparison	331
15.8. String Concat Function	331
15.9. Multiplication Table	332
15.10. Fibonacci Sequence	333
15.11. Rock Paper Scissors	334
Wrapping Up	335
16. Final Projects	336
16.1. Computer vs Computer	336

16.2. How to Prevent Cheating	337
16.3. Student Records	338
16.4. War (Card Game)	340
17. Epilog - Let's Play!	342
Index	344
Credits	381
About the Author	382

Introduction

Practice makes perfect.

Modern Python

Python comes with many builtin high-level data structures, with a very simple and clean literal syntax. It includes a large number of builtin functions and methods. It also comes with an extensive set of standard libraries. It makes it very easy to share your code with the community using modules and packages. Python, like many other high-level languages, also does automatic memory management on behalf of the programmers.

Python supports the object-oriented programming styles as well as the functional programming styles. Its flexibility, such as the dynamic typing, makes it an ideal language for scripting and rapid prototyping in many application areas.

Dynamic languages like Python provide more freedom to the developers. They are more forgiving to the beginning programmers. For small and quick projects, they are ideal. On the flip side, it is generally harder to build larger software systems using the languages like Python.

Python has a long history. It has been around for over 30 years. The language version 2 has been deprecated, and "Python" now means *Python 3*. As of this writing, Python 3.10 is the most recent version.

Although this is a beginner's book, we will try to use some of the modern features of Python, whenever applicable, so that the readers do not have to go through a re-learning process in a short while. One of the most notable features that we will use in this book, which is not generally taught in the beginner's books, is what is called the "typing", or the "type hints" or "type annotations". Although Python is a dynamically typed language, typing provides (some of) the benefits of the static typing, which can help reduce the number of possible bugs in Python programs, especially in large projects.

Book Organization

Python for Beginners: A Practical Introduction to Modern Python is organized into a few dozen "lessons". In the first few lessons, we will briefly cover the absolute basics of programming, and programming in Python. Even if you have some experience with programming in other languages, you may find it useful to go through this part. Python is a rather unique language, and if you are coming from other "C-style programming languages" like C/C++, Java, C#, or even Javascript, then you may find these foundational lessons useful.

In some sense, this part is almost like a (mini) book in a book. It can be independently read, if you are absolute beginners, without having to go through the entire book, and you will still be able to get the essence of Python programming.

This first part also sets the common tone for the book for the readers with different backgrounds. As suggested, even if you have some experience with programming, we highly recommend that you at least browse this part. We cover some "basics" in this book that may be considered "advanced topics" for some beginners.

For the remainder of the book, starting from [Hello Rock Paper Scissors!](#), we focus on our main project, namely, building a CLI version of the Rock Paper Scissors game, from beginning to end. If you are an impatient type, and have some prior experience with programming in Python, then you can also start this book from [the beginning of the Main Project - Rock Paper Scissors](#), without losing too much context.

There is a fair amount of repetitions throughout the book. This is deliberate. We learn through repetitions. The first introductory part and this later project part have some overlaps on the one hand, but they are also complementary to each other on the other hand.

We will work on a few different variations of the Rock Paper Scissors game in this book. By doing so, we will introduce various essential concepts of programming in modern Python.

- [Part 3A](#) - We create a rock paper scissors game with simple procedural

programming.

- [Part 3B](#) - We introduce the basic concepts of Python functions, and we re-implement the game using functions.
- [Part 3C](#) - In the third and final iteration, we use the object-oriented programming techniques to implement the rock paper scissors game.

After completing each implementation, we provide a "lab session", in which the readers can practice programming in Python on their own.

Overall, ***Python for Beginners: A Practical Introduction to Modern Python*** covers the following topics, among others:

- How to install the Python tools locally on your machine.
- How to effectively use the Python interactive shell (aka REPL).
- The basic structure of a Python program.
- Python modules and packages.
- Basic constructs of Python such as expressions and statements.
- Simple builtin data types, e.g., integer, float, bool, and string.
- Complex builtin data types, e.g., list, tuple, and dictionary.
- Objects. Variables and assignments.
- Immutability vs mutability.
- Arithmetic and comparison operations.
- Builtin functions and methods, e.g., print, input, type, etc.
- Loops using the `for` and `while` statements.
- Conditional expressions and conditional statements.
- The new `match` statement. (New as of 3.10.)
- How to define a function using the `def` statement.
- How to define a custom type using the `class` statement.

- How to create a new `enum` type.
- Typing and type annotations.
- Fundamental concepts of programming such as "recursion".
- Object oriented programming.
- Basics of the software development process.

Finally, in the (optional) chapter titled "Final Projects" in [Part 4](#), we include a few project ideas so that the readers can practice what they have learned in this book. The readers are encouraged to try at least one of these projects.

As stated, Python is an "easy" language to learn, and to start programming with. But, *building a good foundation is essential* if you want to become a skilled Python programmer in the long run. Hope you find this book helpful in your journey into the programming world, in Python.

Now, let's get started!

Absolute Basics

So it begins.

— Theoden (The Lord of the Rings)

Our main focus in this book will be going through a few complete programs, from beginning to end. But, we will start with some absolute basics of Python programming.

For the readers who have never done any Python programming, we will start with how to set up a basic development environment. There are many options, including using some of the online tools, but we will mainly cover the traditional setup that uses our own computers as development machines.

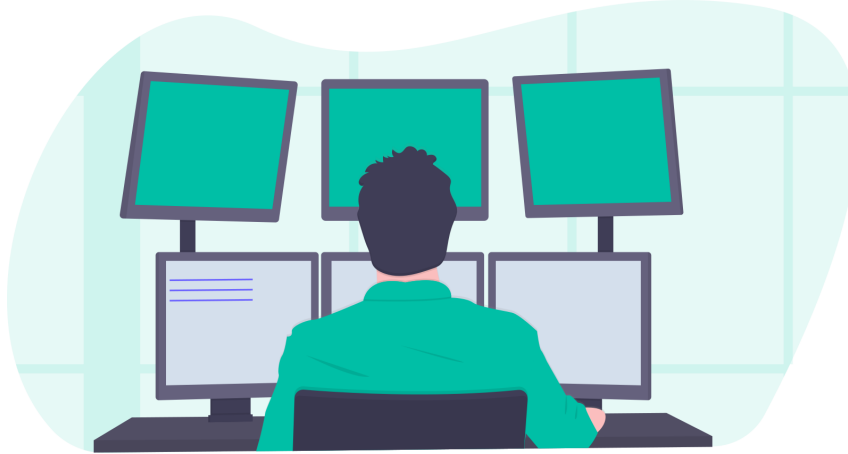
As mentioned in the beginning, we primarily, in fact exclusively, use CLI for development. You will need some familiarity with the terminal programs and Unix shell commands, etc. (We will, however, try to explain everything that is relevant as we go along.)

Python is an "interpreted language". One of the best tools available to Python programmers is the "Python REPL", an interactive Python interpreter. (Or, more precisely, the Python interpreter used in the interactive mode.) We will introduce its basic uses in the earlier lessons so that the readers can explore various features of Python on their own, if desired, while reading this book.

This [first part](#) comprises five chapters, or "lessons", in which we will cover some basic concepts of the programming (in Python) for the benefit of the readers who do not have any real programming experience. Although it is an informal introduction, these lessons will set the tone for the rest of the book.

Now, let's get started!

Chapter 1. Hello Monty Python



1.1. The First Step

Have you ever heard of Monty Python's Flying Circus? It was an old British TV show. Let's say Hi to the Pythons.

hello_python/main.py

```
print("Hello Monty Python!")
```

It is only one line, but this is a complete Python program. We will learn in this book how to write a program like this and run it on your computer. Without specifying how exactly for now, if we run this program on the terminal, then we get the following output:

```
Hello Monty Python!
```

In this book, we will work on slightly more complicated programs. 😊 But, first things first. *What does this simple program mean? What does it do? How does it work?*

1.1. The First Step

Python is a procedural, or imperative, programming language. It supports the object oriented and functional programming styles, among others, but ultimately it is imperative (as in "giving commands"). This means that a Python program consists of a series of "statements". A statement is an instruction to the computer as to what needs to be done. In the "imperative programming world", you will need to tell the computer what should be done and how they should be done, step by step, using the statements. That is programming.

This Hello Monty Python program includes one statement, `print("Hello Monty Python!")`. `print` is a "function". (Note the pair of parentheses following the function name.) In particular, it is a "builtin function", meaning that this function is defined in the language itself (and directly built into the "language interpreter"). Programmers can also define their own custom functions in their programs. We will discuss what exactly a function is and how we can create our own functions, in more detail, later in the book.

In this sample code, we "call" the builtin `print` function with a string "argument", that is, `"Hello Monty Python!"`. (Note the pair of double quotes.) The job of this function is to print, to the terminal, the string, or text, passed in to it as an argument. (An argument is what is between the pair of parentheses.) And, that was what it did when we ran the program above.

The argument `"Hello Monty Python!"` is an "object" of the `string` type. In Python, everything that we deal with is an *object*. An object has an "identity", and each object has a "type" and a "value". More on this later.



If you are new to programming, and if it all sounds gibberish to you, no worries. We will go over these concepts throughout this book. As indicated, we will do *repetition, repetition, and repetition*. ☺ You do not have to understand everything on your first encounter. At the end of the day, what we call *knowledge* mostly boils down to "familiarity".

1.2. The Project

This book takes an interesting, and rather unusual and unique, approach in teaching the basics of programming, in Python.

We will work on one main software project in this book. And, we will mostly focus on the language features and the programming techniques so far as they are useful, or relevant, to the project. On the one hand, this means that you may, or may not, be able to get the complete and comprehensive view of the programming in Python. On the other hand, the real advantage of this approach is that you will get to do the whole software project without being distracted, or overwhelmed, by the nitty gritty details of the entire programming language. The benefits of doing this way will be enormous, especially for the beginning programmers.

Over time, with more training and practice, you will gradually get more exposure to various different aspects of Python programming. But, for now, let's focus. After completing the project, you will get a sense of accomplishment. You will have learned how to *really* program in Python, from start to end. That will be a *huge* accomplishment.

The main project that we will work on in this book is the children's game of rock paper scissors. We will create a computer program that lets a user play rock paper scissors by themselves (e.g., without requiring a game partner).

Rock Paper Scissors is one of the most popular games. But in case you need a refresher, here's a Wikipedia page: [Rock paper scissors](https://en.wikipedia.org/wiki/Rock_paper_scissors) [https://en.wikipedia.org/wiki/Rock_paper_scissors]. The particular project that we work on is somewhat secondary. Our focus in this book is learning Python programming. It will be still helpful, however, if you play a few games before we start working on the project so that you are fully familiar with the problem that we are going to tackle. In any real-world programming problems, in general, a lack of domain knowledge can make your job harder as a programmer.

The computer is just a tool. It is *you* who will have to solve the problems.

Chapter 2. Development Environment Setup



We are going to start working on the rock paper scissors game shortly, but our work has already begun. The very first thing we need to do is to set up a "development environment". If you already have done this, and have done some programming with Python, then you can skip this lesson.



It goes without saying that there are many different ways in which you can develop Python programs. We describe one particular method in this lesson. Although it is not required, if you use the same setup as described here, then it will be a bit easier for you to follow the lessons in this book.

2.1. Command Line Interface (CLI)

In this book, we will mainly use the CLI tools for development. The IDEs, or Integrated Development Environment, can be useful, but they are not absolutely necessary to do programming. In fact, it is better for the beginning programmers to start with the CLI tools. The IDEs can obscure some basic processes of software development.

If you are on Linux or Mac, then you should be familiar with terminal programs. For example, on Ubuntu many people use the *Terminal* program that comes with the standard desktop distribution. On Mac, *iTerm2* is one of the more popular console programs.

If you are on Windows, then we strongly recommend that you install WSL, Windows Subsystem for Linux, with a Linux distribution such as Ubuntu 20.04LTS. Then you can use the Windows Terminal app with a Unix shell.



We will primarily use Unix shell for illustrations in this book, when relevant. If you use Command Prompt (CMD) or PowerShell on Windows, then you may need to do some "mental translations".

2.2. Python Installation

We will need the Python interpreter tools. In some systems, Python may come pre-installed. Let's try and see if you already have Python. Open a terminal app, and type the following:

```
$ python3 --version
```

Or

```
$ python --version
```

2.2. Python Installation

If either of these commands returns an output that indicates that you have a python version 3.10 or later, e.g., `Python 3.10.0` or `Python 3.10.1`, etc., then you are good to go. Use that command which generated the desired version output. Otherwise, you will need to install a recent version of Python.



The dollar sign `$` is used to represent the shell prompt in this book. You type the commands after the "`$`" prompt. The shell prompt may look different on your system.

You can download the Python source code from their *official* download page, [Download Python](https://www.python.org/downloads/) [https://www.python.org/downloads/], and you can build it yourself if you feel brave enough. That is, however, *not* recommended for the beginners. ☹

You can just use one of the platform-dependent *pre-built* Python distributions. (Although some of them may not be "official", most Python developers use one of these distributions.) Try to install the most recent stable version. (3.10.4 is the current stable release as of this writing, and by the time when you read this, newer versions like 3.11 may have been released.)



It is not required to use the most recent version to learn Python. But, if you are just starting, then there is no reason to use an older version. Every time a new version of Python is released, you will have to go through some (small or big) relearning process, sooner or later. By starting with the most current version, you are doing yourself a big favor by not having to go through these relearning processes. Besides, some of the sample code of this book use some features from Python 3.10 or later.

If you are on Windows, here's the binary distribution: [Python Releases for Windows](https://www.python.org/downloads/windows/) [https://www.python.org/downloads/windows/]. Note that if you use WSL on Windows, as we recommend, then you may instead want to install an appropriate distribution for Linux on your Windows subsystem.

If you are on Mac, you can find the installers on this page: [Python Releases for](#)

macOS [<https://www.python.org/downloads/macos/>].

If you use Linux (including Linux on WSL), then you may have to search the Web to find the appropriate installation instruction for your specific Linux distribution. Most Linux distributions come with Python, but the version might be old. As stated, we recommend you install version 3.10 or later.



For example, you can do a Web search with the keyword phrases like these: "how to install the latest version of python on ubuntu", "install python 3.10 on debian", etc. Note that you can have multiple installations of Python (e.g., with different versions) on your system.

If you use Ubuntu 20.04 or 22.04LTS, for example, then you can install Python 3.10 using the `apt` tool.

```
sudo apt install -y python3 ①
```

① The `python3` package currently includes the version 3.10.

Make sure that your installation was successful by trying out a few python commands. For example,

```
$ python -h
```

The actual command name can be different on your system. It could be `python` or `python3` or something else. Although Python version 2.x has been deprecated, due to the historical reasons, many (Linux) Python distributions may still use the name `python3`, or something similar, for the python command.

Does it successfully run? Do you get an output that starts with "*usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...*" or something similar? If you type `python --version` (or, `python3 --version`, etc.), then what is the output?

2.3. Visual Studio Code

What about this?

```
$ python3 -c 'print("hello world!")'
```

Do you get an output, *hello world!*? (You can run a simple Python script, without having to save it in a file, using the `-c` flag.)

OK, assuming that everything has gone smoothly, let's move on to the next task.



Some Python distributions come with a GUI Python tool, called IDLE. It is a basic IDE, which lets you save and load Python programs. It also includes a Python interactive shell with the syntax highlighting. If you prefer to use this tool, and if it is not already installed, then again do a Web search to find out how to install it on your platform. On Ubuntu, you can again use `apt`, for example, `sudo apt install idle-python3`. Then you can start IDLE with `idle-python3`.

2.3. Visual Studio Code

Although we are not going to use any (specific) "IDEs" in this book, we will still need a good text editor for programming. Those days are over when we used Notepad to program. ☺ There are many good options, including Sublime Text, and emacs and vim if you are a Unix/Linux user. In this book, we will install and use VS Code (aka Visual Studio Code), for illustration.

So, what is the difference between the IDEs and the programmer's editors? In fact, there is really not much difference. These editors are so good these days that you can do pretty much anything you can do with the full-blown IDEs. The difference is really minimal until you need the full power of the IDEs (which can be "never" for most people).

One main difference for the beginning programmers, and for those of us who

program in multiple programming languages, is that the IDEs are often specific to one (or a few) particular programming language(s). (They often come with particular language "SDKs" already installed.) When you program in Python, you may use PyCharm, or Spyder, etc. When you program in Go, you may use GoLand or something else. When you program in C#, you may use Visual Studio. What do you use when you program in Java? What about C++? Javascript? Haskell? Rust? ...

It used to be the case that this was required despite this obvious inconvenience. But, those days are over now. (When you do heavy GUI programming, you may still benefit from using the specialized IDEs.)

When you use the modern programmer's editors (also known as the "lightweight IDEs"), you install an extension(s) specific to each programming language that you use, which may need to connect to the "language server" processes. These language servers provide all necessary services that have been traditionally provided by the specialized IDEs.

Now, let's install VS Code, that is, if you don't have it already installed on your computer. Here's the download page: [Download Visual Studio Code](https://code.visualstudio.com/download) [https://code.visualstudio.com/download]. Follow the instructions specific to your platform.

If you use WSL on Windows, then the single installation of VS Code (on Windows) can be used both on your Windows host machine and on your Ubuntu subsystem, for example. Here's an instruction: [Developing in WSL](https://code.visualstudio.com/docs/remote/wsl) [https://code.visualstudio.com/docs/remote/wsl]. No need to be intimidated. There are only three steps, the first two of which you have already done at this point.

Even for the third step, what you really need is the [Remote - WSL extension](https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-wsl) [https://marketplace.visualstudio.com/items?itemName=ms-vscode-remote.remote-wsl]. VSCode has an interesting mechanism through which you can install multiple extensions as if they are a single extension. It is called an "extension pack". If you would like, you can install the Remote Development extension pack.



Windows users get special treatments here, but if you are serious about learning programming, even as a hobby programmer, then it

2.4. Python Extension

is important for you to learn the CLI basics on Unix-like systems. If you don't learn anything about Python, or anything else, from this book, it will be still worthwhile your time if you start using WSL/Linux subsystem for programming on Windows. As a matter of fact, the "Windows Subsystem for Linux" is the best thing that ever happened to the programmers using Windows. ☺

- [Install WSL](https://docs.microsoft.com/en-us/windows/wsl/install) [https://docs.microsoft.com/en-us/windows/wsl/install]

Note that this is a long term investment. Eventually, over time, you will learn more Unix shell commands and shell scripting, and you will become a better developer (regardless of what programming languages you use).

BTW, here's the link to the Windows Terminal app on Windows Store: [Windows Terminal by Microsoft](https://www.microsoft.com/en-us/p/windows-terminal/9n0dx20hk701?activetab=pivot:overviewtab) [https://www.microsoft.com/en-us/p/windows-terminal/9n0dx20hk701?activetab=pivot:overviewtab]. You can run CMD, PowerShell, or even Linux Shell such as BASH on Windows Terminal.

2.4. Python Extension

Now that we have VSCode on our machine, let's open it. Open your terminal program (Terminal, iTerm2, or Windows Terminal, etc.), and `cd` to any folder. For example,

```
$ mkdir hello_1
$ cd hello_1
```

Then try this, on your shell:

```
$ code .
```

If VSCode is correctly installed, and if the command `code` is in your "system path" (which may have required some additional steps depending on your platform), then it will open an instance of VSCode. You can use an icon on your GUI desktop to open VSCode, but we will always use the command line in this book for illustration. The dot `.` represents the "current working directory" in Unix-like systems.



They call the currently open folder the "workspace" in VSCode. In fact, you can include multiple folders (from different locations) in one workspace. You can also open, and use, multiple instances of VSCode at the same time as long as they are associated with different workspaces.

Now install the [Python extension \(by Microsoft\)](https://marketplace.visualstudio.com/items?itemName=ms-python.python) [https://marketplace.visualstudio.com/items?itemName=ms-python.python]. You can do this from the "Extensions" menu, e.g., on the left hand side, inside VSCode. The search box at the top (of the left hand side panel) can be useful when you are searching for particular extensions.

If you are done, then close the VSCode. You can delete the empty directory as follows (from its parent directory), if you want:

```
$ rmdir hello_1
```



For more information on using Python with VSCode, refer to this page: [Python in Visual Studio Code](https://code.visualstudio.com/docs/languages/python) [https://code.visualstudio.com/docs/languages/python].

2.5. Test Program

Here's a small Python program.

dev_setup/hello.py

```
1 def hello(name: str) -> None:
```

2.5. Test Program

```
2     print("hello " + name)
3
4
5 hello("joe")
```

Create a folder anywhere on your file system. This is a temporary folder that you can delete later. You can use the previously created directory, *hello_1*, if you haven't deleted it, or you can create a new one. Let's use the name "hello_world" this time. (The code sample is located in the folder *dev_setup* in the author's computer, as indicated by the label. The directory names are not significant in this example.)

```
$ mkdir hello_world && cd $_
```

If you use BASH, then `$_` refers to the previously used "command line argument", *hello_world* in this case. (Note the underscore `_`. The dollar prefix `$` is used to refer to the "variables" in shell, and in the shell scripts.) Otherwise, you can just do `cd hello_world`. The second command after `&&` is executed only if the first command succeeds.



If you use CMD or PowerShell on Windows, then you may need to use `md` and `del` to create and delete a folder, respectively, instead of `mkdir` and `rmdir`. You can also use Windows File Explorer if you prefer to use the graphical user interface.

Next, open VSCode in that directory:

```
$ code .
```

Create a new file in that folder, and name it "hello.py". (You can use the "File | New File" menu to create a new file, or use the keyboard shortcut, Control+N or Command+N, depending on your platform.) Type in the code above in the file and save it.



The line numbers, displayed on the left-hand side of the code box, are not part of the program. Do not include them in your program source file.

Then open a builtin terminal in VSCode. You can again use the menu, "View | Terminal", or use the keyboard shortcut (as shown on the menu, either Control+(back tick) or Command+(back tick)). This is one of the few VSCode keyboard shortcuts you may want to learn by heart (because you will likely use it frequently since it is very convenient to do development using the builtin terminals).

Note that you can even open and use multiple terminals at the same time. We will leave it to the readers to figure out how to do this.



Note that we do not include any screenshots in this book. Screenshots can be easily outdated, and they can be misleading in some cases. But, more importantly, it is a good training to try and figure things out for yourself. Remember, learning to program is 10 times, or 100 times, harder than learning to use a new software (like VSCode). 😊

In the terminal, type the following, and press Enter. (The "Press Enter" part is always implied.)

```
$ python hello.py
```

The argument *hello.py* following the python command is the name of the file that we just used to save the sample code. (Here and throughout this book, replace the command *python* with whatever you use on your system to start the Python interpreter.)

Do you see an output like this?

2.6. How to Run Python Programs

```
hello joe
```

If so, then *CONGRATULATIONS!* You are ready to go!

If not, don't despair. As with any real world problems, when you program, there are a million things that can go wrong (without *any* exaggerations ☹).

The thing is, though, this book cannot help each reader with every possible problem. You will have to find a solution. In many cases, the particular error messages you are getting will help you troubleshoot the issue.

In this particular case, there can be many reasons. Are you in the correct directory? Is the file saved? In the correct directory? Did you type the code exactly as shown above? Including all the white spaces? Without the line numbers? Are you running the python command in the correct location? And so forth.

Whenever you run into an issue, however, just remember that it is one more chance that you can learn something new. (And, you will end up making the same mistakes over time. Again and again. ☹ Don't be discouraged. Nobody learns these things in their first try.)

2.6. How to Run Python Programs

The *python* command generally follows the same pattern/convention that is common with many commands available on Unix. There are a number of different ways to run a Python program (or, script).

First, we can specify the Python script file following the *python* command, as a "command line argument", as we just did with the *hello.py* file.

```
$ python hello.py  
hello joe
```

Second, we can run a Python "module" with the `-m` "command line flag".

```
$ python -m hello
hello joe
```

Note that we do not include the file name extension ".py" in the argument following the `-m` flag. In Python, a Python file is both a *script* and a *module*, and the file name, without the file extension, is the name of the module, by default. We will discuss this further later in the book.

We also have seen another way to run a Python command in the first lesson, namely, using the `-c` command line flag.

```
$ python -c 'print("hello"); \
print("joe")'
hello
joe
```

The backslash (\) plus newline combination is used in Unix shell to input a single line command in multiple lines. (The terminology is pretty confusing, but unfortunately that's the way it is. ☺) That is, the above command is the same as this:

```
$ python -c 'print("hello"); print("joe")'
hello
joe
```

Note that the strings "hello" and "joe" are printed out in two separate lines. This is because each `print` function adds a newline to its argument.

Running a Python script (text) using the `-c` flag is not very practical for all but some short scripts or simple commands. In the next lesson, we will see a few more different ways of running a Python program.



If you use IDEs, including some programmer's editors like VSCode, then you can easily run, and "debug", your Python programs in the IDEs. As stated, however, we will exclusively use the CLI commands for illustration in this book.

2.7. Code Review

Before we continue, let's take a look at the program that we just wrote. What do you *see*?

There are three lines of code (excluding the empty lines), and the second line is "indented" relative to the other two lines. This means that the second line "belongs" to the "less-indented" line above (that ends with a colon `:`), the first line in this case.

```
def hello(name: str) -> None:
    print("hello " + name)
```

A Python program is essentially a series of "statements". Lines 1-2 form one statement (because line 2 is part of the larger structure that includes both lines 1 and 2). This is called the "compound statement" in Python (because it comprises one or more other statements). Line 5 is another statement. A "simple statement".

```
hello("joe")
```

The first statement "defines" a function (using a Python "keyword" `def`), which is named `hello`. In this case, the `hello` function, when it is "called", prints out a text, which is the concatenation of the string `"hello "` and the function argument (named `name`). The plus "operator" `+` is used to concatenate two strings. (Note that we are assuming the name `name` is also a string here.)

The second statement, line 5, then calls this newly defined function with an argument `"joe"`. (Again, using a pair of parentheses.) When we run this program,

we get the output, `hello joe` (without the double quotes). Note that the function parameter `name` (line 1) is replaced by the supplied argument `"joe"` when we "call the function" like this (line 5).

In fact, we could have instead called it as follows:

```
hello(name="joe")
```

They are more or less equivalent to each other. You will get the same output from this alternative way of calling a function using the "keyword argument". This latter "syntax" is more explicit, but it is more verbose. You will have a lot of choices when you program. 😊

You do not have to understand all this now. We will go through this later in more detail, while working on the rock paper scissors game.

2.8. Type Annotations

One thing to note, before we close this chapter, is that we use the "type annotations" in this book. For example, the word `str` used in line 1 of the sample code indicates that the function takes a string argument (`: str`). And, it "returns none" (`→ None`).

At the risk of oversimplification, it is generally easier to write a code in the dynamically typed programming languages like Python, compared to the statically typed languages like C/C++, Java, Go, or C#. On the flip side, you will generally end up with more bugs when you program in the dynamically typed languages. This is because the compilers of the statically typed languages do a lot more to find certain classes of bugs. In languages like Python, you may run into certain types of bugs only at run time, e.g, in production, which is generally not the best time to find bugs. 😊 (Software testing has limitations as well.)

Python has a feature called the "type hint" or "typing" (which is a standard library "module"). It is optional, but by using `typing`, you can get (some of) the benefits of using the statically typed programming languages. Clearly, this is a tradeoff. There

2.8. Type Annotations

are pros and cons. In particular, typing adds some overhead when you develop a software. And, you will have to use additional tools like "MyPy" to get the full benefits of typing, which is well beyond the scope of this book.

It should be noted that many of the experienced Python programmers still do not use typing even though it was first introduced to Python a few years ago. This is partly because it is hard to change the habit. If you have been programming in Python in a certain way for years, then it is rather hard to adopt a new way of programming (even if you intellectually understand the benefits). As the saying goes, you can't teach an old dog new tricks. ☺

That is one of the reasons why we decided to introduce typing in this beginner's book. We are not going to do anything with typing. We will just use a small subset of the feature called the "type annotations". It is optional. If you think that this is just too much complication for a beginner like you, then you can simply ignore it. The `hello` function definition above, for example, could have been written as follows without the type annotations. (It is a lot easier to "remove" than "add".)

```
def hello(name):  
    print("hello " + name)
```

It is your choice. As a matter of fact, that is precisely why we introduce the type annotations in this book: To give a choice to the beginning Python programmers. If you don't know if something exists, then you don't have a choice. You follow the same old "traditional" way of programming in Python without even knowing that there is an alternative. Possibly a *better* alternative.

The readers of this book have a choice. After finishing this book, if you decide to use typing more in your future Python projects, good for you. If you decide otherwise, then that's fine too. As stated, a vast majority of Python programmers still have not adopted the typing.

Just to be clear, the Python interpreter (mostly) does not know or care about typing. (The type "annotations" have to follow certain Python syntax, however.) When we

use the type annotations in the sample code of this book, those are just for us, programmers. But, even at this level of (minimal) use, we believe that the benefits are enormous. By forcing you to think about the types in the early stage, it will make you a better programmer in the long run.

It is a promise. 😊

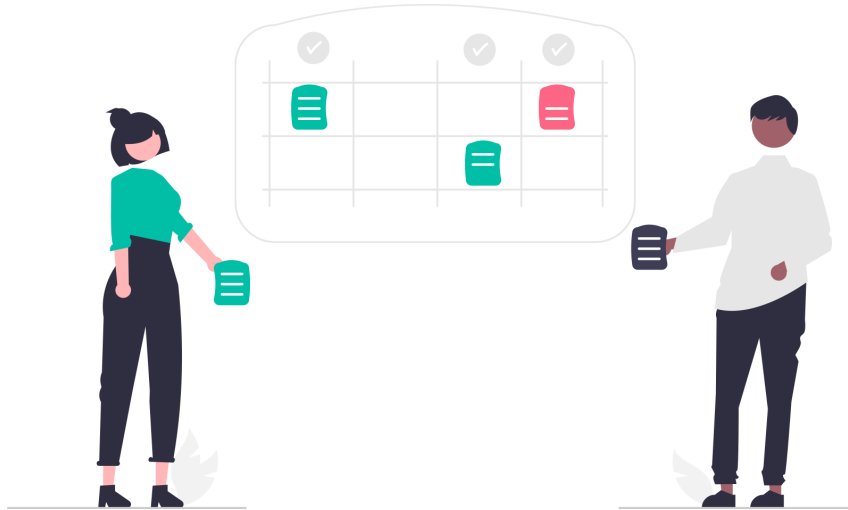
2.9. Summary

We installed the Python CLI tool, VS Code, and the necessary extensions. We tested our setup using a simple Python program. And, we reviewed our first program together.

Although this book can be used as a reading material, you will get the most out of this book by doing all, or most of, the tasks (and, exercises). If you intend to do so, then the dev environment setup is a crucial step. Make sure that you have a working environment before you continue. (If you cannot install Python 3.10, for instance, an older version should be good enough to follow most of the lessons in this book. It is never *all or none*.)

One thing to note is that *this is not a competition*. You are in no rush. Take your time, and *enjoy what you are doing*. Also, remember, troubleshooting is part of "life". 😊

Chapter 3. Interactive Tour of Python



3.1. Python REPL

You can start a Python interactive session by simply invoking the `python` command (e.g., without any arguments).

```
$ python
```



The actual `python` command on your system might not be exactly *python*, as explained in the previous lesson. For consistency, however, we will just use the name *python* throughout this book. You can substitute the appropriate command name for *python*.

On the author's computer, it looks like this:

```
$ python3.11 ①  
Python 3.11.0rc1 (main, Aug 8 2022, 18:31:02) [GCC 11.2.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.
```

```
>>>
```

- ① "python3.11" is the command that the author used to start the Python interpreter in this example. This is a pre-release version of 3.11, as of this revision (early September, 2022).

The symbol ">>> " is the (default) Python prompt. You type (line-based) Python code after this prompt. For example,

```
>>> pass ①  
>>>
```

- ① We typed 'p', 'a', 's', 's', and pressed Enter (which is not directly visible in this output).

This four-letter word `pass` is a "keyword" in Python, and it has a special meaning to the Python interpreter (unlike random strings like "hello" or "world"). In this case, this is also a complete statement.

As mentioned earlier, a "statement" is an instruction to the computer, the "Python interpreter" in this case, as to what needs to be done.

`pass` is the most useful and most useless statement at the same time in Python. 😊 The `pass` statement tells the Python interpreter to *do nothing*. And yet, in Python programs, especially during the active/early-stage development, it is indispensable. We will discuss this further later in the book.

Let's try something just a little bit more interesting. We used the `print` function in the previous lessons. Let's try it here.

```
>>> print("Hello Python the Snake!") ①  
Hello Python the Snake!  
>>>
```

3.1. Python REPL

- ① We are not going to mention the "press enter" part any more in this book. Sometimes you should be able to see what is not shown. 😊

The text `Hello Python the Snake!` displayed by itself on a line (e.g., without the leading `>>>`) is the output generated by the "function call" `(print(...))`, which is also considered a statement in Python. Notice that the `print()` function automatically adds a newline in the output, after the given text. (Can you see it? 😊)

To quit the Python interactive shell, you can call Python's `quit` or `exit` functions, or use the EOF signal (e.g., Control+D on Unix-like systems and Control+Z on Windows). For example,

```
>>> quit()
$
```

`"$ "` represents the shell prompt, here and throughout this book. As you can easily notice, the prompt `>>>` indicates that we are inside the Python REPL, and the prompt `"$ "` signifies that we are not.

We have seen three different ways to run a Python program or script so far. Running it as the first command line argument to the `python` command, and running the script with the `-m` or `-c` flags, as we discussed in the previous lesson.

We can also run a script while starting the Python interpreter interactively, using the `-i` command line option.

Here's how to do it:

```
$ python -i hello.py
hello joe                                ①
>>>                                     ②
```

- ① Output from the `hello.py` script.

② The Python shell prompt.

We are using the `hello.py` program file that we used in the previous lesson. Notice the Python interactive shell prompt (`>>>`), which is displayed *after* the specified script has run.

You can even run the program after starting a Python shell. For instance,

```
>>> import hello
hello joe
```

We use the `import` statement in this case, Note that we do not include the file extension ".py" in this `import` statement. As mentioned earlier, `hello` is the name of the "module" defined in the "hello.py" file. This is *not*, however, a common way to "run" a Python script. The `import` statements are primarily used to import the names defined in other modules, both in the interactive mode and non-interactive mode.



During the development, if you would like to try out different ideas and what not, then running a script via `import`, and importing other functions, etc. from the script, can be useful at times.

3.2. Basic Concepts of Programming



If you have some experience with programming, in Python or otherwise, then you can skip this section. Most of the basics explained in the earlier lessons will be repeated throughout the book.

A computer is really no different from a digital calculator. A computer does a lot more, and it is a lot more complicated, than a simple desktop calculator, but the fundamentals are the same.

3.2. Basic Concepts of Programming

Let's start from the absolute basics. If we want to add two numbers, say, **1** and **2**, using a calculator, then what do we do?

We press number **1**, operator **+**, and the number **2** button, in this order. Finally, we press the equality operator **=** button (or, something equivalent). Then, the result is shown, **3**, on the display.

This particular example input sequence can be written as follows:

```
1 + 2
```

This is an "expression" in the programming parlance. An expression is something that evaluates to a "value". **1** and **2** are values. So is **3**. And hence **1 + 2** is an expression (since it evaluates to a value, **3**). A value by itself is an expression (because a value trivially evaluates to the same value).

The numbers like **1**, **2**, or **3**, or **100** are called the "literals" in programming. More specifically, the "integer literals" in this case.

In Python, there are a few different kinds of literals. We use the term "types" in programming, but it is not important to know what exactly a "type" is at this point. A more important thing to note here is that the literals in Python belong to a few different categories. For example, "numbers" belong to two categories, integer numbers and real numbers.

Integers are the whole numbers like **1**, **-2**, or **1000**. Real numbers are the numbers with the decimal point, like **2.5**, **-10.75**, or even **5.0**. We call real numbers the "floating point numbers" in programming because of the way they are represented in the computer memory (e.g., with **0**s and **1**s). (The decimal point literally "floats" in these representations depending on the value/size of the number 😊, meaning that it does not have a "fixed" position.)

Python has another builtin number type, **complex**, for representing the mathematical "complex numbers", but we will not use complex numbers in this

book.



We will use various programming terms in this book without first defining them. You will learn them through examples. Not by memorizing some formal definitions (even if such things exist). For instance, REPL is an acronym for *read-eval-print-loop*. Now, do you "understand REPL better" now that you know that it is an acronym and what it stands for? Probably not. 😊 It is just a word. Just a name.

"What's in a name? That which we call a rose. By any other name would smell as sweet."

3.2.1. Boolean Literals

Python has a special kind of integer numbers, called `bool`. There are only two literals in the `bool` type, `True` and `False`, which represent the logical true and false values, respectively. In the "numerical context", they have the values, `1` and `0`, respectively. (If you are coming from other modern programming languages, where Boolean is a separate and distinct type, this may come as a surprise. But, yes, `bools` are indeed (special) integers in Python.)



Can you guess what would be the value of `5 + True` as a Python expression? Or, how about `10 * False`? 😊

3.2.2. Arithmetic Operations

Python supports all the usual binary arithmetic operations between two numbers. For example, addition:

```
>>> 10 + 15
25
```

3.2. Basic Concepts of Programming

Subtraction:

```
>>> 30 - 10
20
```

Multiplication:

```
>>> 2 * 5
10
>>> 2.5 * 4
10.0
```

One thing to note is that if at least one of the operands is a `float` number, the result is a `float` number. Note that the value of `2.5 * 4` is `10.0` (float) not `10` (int). An arithmetic operation between two integers yields an `int` value. This is true for additions, subtractions, and multiplications. The division is an exception:

```
>>> 4 / 2
2.0
```

Regardless of the types of the two operands, the division operator `/` always returns a `float` value. Python, however, has another division operator `//`, which always returns an `int` value between two integers.

```
>>> 5 // 2
2
```

Although mathematically `5` divided by `2` is `2.5`, this expression `5 // 2` evaluates to `2`, an `int`. This operation is sometimes called the "integer division". It truncates the decimal part. This is true even between the operands of the `float` type.

```
>>> 5.5 // 2
2.0
```

It returns a `float`, but the value has still been truncated. In general, this operator `//` is known as the "floor division".

Another arithmetic operation that is primarily used for integers is the so-called modulo, or remainder, operation. The modulo operator `%` returns the remainder of the first argument after dividing it with the second argument. If both arguments are `int` then it returns `int`. Otherwise it returns `float`. For example,

```
>>> 5.5 % 2
1.5
>>> 5 % 2
1
```

Note that `5.5 // 2` is `2.0` and hence its remainder is `5.5 - 2 * 2.0`, that is, `1.5`. Likewise, `5 // 2` is `2` and hence its remainder is `1`. (Make sure that you understand what these statements means. If you don't, then pause, and take your time.)

Now, just for fun ☺, let's try these operations with `bool` values.

```
>>> 2.5 + True
3.5
>>> 3 - False
3
>>> 5.0 * False
0.0
>>> 10 / True
10.0
>>> 10 // True
10
>>> 5.5 % True
```

0.5

Unlike in mathematics, the numbers in programming, and hence their operations, have finite "precisions". Integers and floating point numbers are typically represented by 4 or 8 bytes in many programming languages (as defined by their language specifications).

In Python, on the other hand, there are no limits as to how many bytes can, or should, be used to represent integers. It is really implementation dependent. In a sense, Python integers use infinite precisions, for all intents and purposes. Python `float` number objects, however, still use 8 bytes (in most systems), to represent their values, which corresponds to `double`, or the "double precision" floating point numbers, in other programming languages. It should be noted that computations on a computer involving floating point numbers have the "round-off errors".



As we briefly mentioned, an object in Python is more than just its value. Hence, for instance, Python may require more than 8 bytes to store a `float` number object in memory.

3.2.3. Immutability

Numbers are "objects" in Python, just like anything else. They are stored in memory just like any other objects. One thing special about these number objects is that they are "immutable". That is, once created, their values do not change.

There are a number of "builtin types" in Python. `float` and `int` (and `bool`), along with a few others, are called the "simple types". Objects of simple types are all immutable. In fact, there is no way to modify the value of an immutable object in Python. In contrast, the vast majority of the objects are actually "mutable", as we will see throughout this book.

Each object in Python has a unique "identity". The builtin `id` function returns the identity of a given object. If the identities of two objects are different, then they are different objects. An object has the same and invariant identity, and the identity

cannot change (regardless of whether it is mutable or immutable).

For immutable types, there cannot be more than one immutable object of the same value.

```
>>> 10
10
>>> id(10)
140263172407824
>>> id(2 + 8)
140263172407824
>>> id(10 * True)
140263172407824
>>> id(20 // 2)
140263172407824
```

As we can see from this example, all **10**'s in these expressions have the same identity. They *are* one and the same object, not just the objects with the same value. Next time we start the Python interpreter, the number **10** may end up with a different identity (e.g., located at a different place in memory), but still there will be only one **10** at any given moment. You cannot modify the value of this object **10**.

We will continue this "tour" in the next couple of lessons.

3.3. Summary

We learned how to start a Python interpreter in the interactive mode. You can type Python statements to execute them, and see the results, interactively. The Python interpreter used in the interactive mode is sometimes called the "Python REPL". (BTW, we typically read REPL as one word, which rhymes with pebble, not as an acronym.) In this mode, if you input an expression, it shows the value of the expression. (This behavior is somewhat different in the non-interactive mode, as we will see later in the book.)

3.3. Summary

Python has a number of "builtin types". We briefly looked at two (or, three) numerical types in this lesson, floating point numbers (`float`), integers (`int`), and booleans (`bool`). The `bool` type is a "subtype" of `int`, and it has only two values, `True` and `False`. These builtin numeric types are "immutable", that is, once an object of a numeric type is created with a certain value, this value never changes.

One special property of an immutable object in Python is that there cannot be more than one immutable object with the same value at any given moment.

The values/expressions of numerical types support the usual arithmetic operations such as additions (+), subtractions (-), multiplications (*), and divisions (/ and //). We will assume that the readers are familiar with these basic mathematics, and we will use them throughout this book.

One thing to note is that Python has two kinds of divisions. The // operator is called the "floor division", and when it is used for two integer numbers, it return a (truncated) `int` value. The modulo operator % is used to get the remainder after floor division. All operators, +, -, *, %, and //, except /, return integer values when both operands are integers. Otherwise they return `float` values. As for the (normal) division (/), it always returns a `float` value regardless of the types of their arguments.

Chapter 4. Numbers, Strings, and More



This lesson is a continuation of the ongoing "tour", and it will be more "hands on".

4.1. Starting Python Interactive Session

Let's start the Python shell (or, the "REPL"). You can follow along if you have an access to a computer with the Python tools installed *right now*. Otherwise, everything that you need to understand the content of this lesson is included in the book.

```
$ python ①  
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on linux  
Type "help", "copyright", "credits" or "license" for more information.
```

4.2. Numbers in Python

```
>>>
```

- ① As indicated, we will just use the generic *python* for the actual python command, which may be different on your system. "\$ " represents a shell prompt. Incidentally, the author uses Ubuntu 22.04, and the BASH shell, to create the sample programs for this book.

The Python prompt, `>>>`, in the last line indicates that the interpreter is ready and is waiting for the user input.

4.2. Numbers in Python

Let's try typing **1** at the prompt (and Enter):

```
>>> 1          ①
1
>>>          ②
```

- ① You can follow along if you'd like.
- ② Python waits for the next "command" once it executes/evaluates the given statement/expression. We will omit this "next prompt" from the sample outputs from now on.

The interpreter echoes back its value, namely, **1**. The value of the number **1** is **1**. That is so obvious. 😊 One can easily guess what the value of **100** is. Let's try it. Again, type **100** and press the Return key (or, the Enter key, depending on whichever name you prefer):

```
>>> 100
100
```

Yes, the value of **100** is indeed **100**. Wow, programming is so easy. 😊 How about some "real" numbers like **5.25**?

```
>>> 5.25  
5.25
```

Of course. Numbers are numbers, in Python.

In Python, the data, like a number, is called an "object". All objects are associated with "types". As we saw in the previous lesson, Python has two different "built-in types" for numbers, **int** and **float**. (As mentioned, we are ignoring the **complex** numbers in this book. Also note that, although **bool** is technically a subtype of **int** in Python, people do not often call **bool** a numeric type.)

We can view the type of an object by using a "built-in function", **type**.



As stated, we use various programming terms without first defining them. This is mainly because the "formal definitions" are not very useful, especially for the beginning programmers. We learned how to recognize an apple, for instance, by seeing a lot of different apples, and tasting them ☺, and not by learning some dictionary definition of the word *apple*.

Throughout this book, we will often use the quotes to indicate that we are introducing new terms, like "types", "built-in types", or "built-in functions", etc., or to indicate that we are using them without explaining them first. If you are already familiar with those terms, you do not even have to pay attention. If you don't know what they mean, no worries. After reading this book, you will. (Otherwise, ask the author for money back. ☺)

4.3. Builtin **type** Function

Let's try the **type** function:

```
>>> type(55)
```

4.4. Errors

```
<class 'int'>
```

(Note again the parentheses after the function name `type`.) Or, how about this?

```
>>> type(1.5)
<class 'float'>
```

Aha, the type of number `55` is `int` whereas that of number `1.5` is `float`. (Incidentally, all types in Python are "classes". These two terms are more or less synonymous. More on this later.)

In mathematics, an integer number *is* a real number. In fact, an integer is a rational number, and a rational number is a real number. In many computer programming languages including Python, however, the `int` type is different from the `float` type. A value of one number type may be "converted" to that of the other type in some way, as we will see shortly, but nonetheless they are two distinct types.



A *type* plays important, and critical, roles in modern programming. As an example, if you have a sequence of 0's and 1's at a certain memory location, say, a sequence of 8 bytes, what does it mean? We can only tell what it means if we know the type of the object in that memory location. If it is an `int`, it may be `1201`. If it is a `float`, it may be `0.456`. These are just arbitrary/made-up numbers for illustration, but you get the point. Without types, there will be no (high level) programming. You will run into types, again and again, throughout this book, and throughout your programming career. 😊

4.4. Errors

Incidentally, only a *valid expression* in Python has a value. Likewise, Python only executes the *syntactically valid statements*. For example, if you type "what?" at the prompt, the Python interpreter will not be very happy:

```
>>> what?
      File "<stdin>", line 1
        what?
        ^
SyntaxError: invalid syntax
```

Or, if you just type "what",

```
>>> what
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'what' is not defined
```

Oh my, what's all this? If you ask people a question that they don't know an answer to, then some of them may get rather irritated. The Python interpreter is just like those people. ☹ It just spits out some gibberish. And, it gives up. (The Python interpreter is essentially saying that it is *your* fault and not his. ☹) We will get to this concept of "program errors" later in the book.



Or, more precisely, it's the "programmer errors". But, *for some reason*, programmers do not like to use that term. It's always the *program* errors. ☹

4.5. Expressions

An expression is anything that evaluates to a value. Remember? ☹ For example, `1 + 2` is an expression. Why? Because it evaluates to `3`, which is a value.

```
>>> 1 + 2
3
```

4.6. Interactive vs Non-Interactive Modes

And, a value is an expression. Hence, `3` is also an expression.



We typed `1 + 2` in this example. We could have just typed `1+2`, and we would have gotten the same result. In Python, white spaces (e.g., spaces, tabs, and newlines) are important in certain contexts. In some other contexts, however, they are not significant. This can be very confusing. Unfortunately, or fortunately, the best way to learn this is again through examples, and not by memorizing the grammatical rules.

In this particular example, we use spaces for readability, as is the common practice. For most people, `1 + 2 + 3` is easier to read than `1+2+3`. (You may or may not be the "most people", but that's OK. 😊)

As we have seen earlier, some (invalid) statements or expressions may raise errors. As another example,

```
>>> 10 / False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Since the value of `False` in the numerical context is zero, this expression raises a `ZeroDivisionError` exception. In mathematics, a division by zero might have resulted in the "infinity", but in computer programming (in general), this kind of expressions cannot be evaluated.

4.6. Interactive vs Non-Interactive Modes

The interactive Python interpreter is somewhat special. You can type an expression or a statement at the prompt. If you type a valid expression, the interpreter shows its value (unless it is `None`, as we will see shortly). If you type a valid statement, then it

executes the statement.

In general, a "Python program" is a sequence of statements (e.g., in the non-interactive mode). An expression can be part of a statement. An expression by itself can be a statement (e.g., if it is written on a line by itself). The result of an expression, when used alone in a line in a program (e.g., in the non-interactive mode), will be ignored by the Python interpreter.

In Python, in particular, a "function call" is an expression, and it can also be used as a statement by itself. A function in Python, and in other "imperative programming languages", is not a *real function* in the mathematical sense. Functions in imperative programming languages are more like procedures or subroutines, and they *can* have "side effects". In fact, many of them do.

4.7. Builtin `print` Function

For instance, we have used a builtin function `print` before, which is probably one of the most commonly used functions in Python programs.

If we try calling it as follows:

```
>>> print(_)  
3
```

It prints out `3`. The symbol `_` is a special "variable", which is predefined in the Python interactive shell (but, not in the non-interactive mode). It refers to the "previous value". In this example, we happened to calculate `1 + 2` just before this statement, and hence `_` *refers to* the object `3`. In this case, therefore, the statement `print(_)` is equivalent to `print(3)`.

We will come back to the concept of variables, or names, later in this lesson and the next. For now, note that the `print` function has a *side effect*. Its side effect is printing out its "argument" to the "stdout", or the terminal.

4.7. Builtin `print` Function

The argument of `print()` in this function call is the variable `_`, which is currently `3`. Hence the result of this statement being executed is printing out `3`, as a side effect of the `print()` call, as shown above.



At the risk of sounding like a broken record (that is, if the reader knows what the "record" is 😊), you can ignore any of the terms that you do not understand. We are using a lot of words and phrases here that we do not precisely define. But they will all seem obvious, or self-evident (as in "not requiring a definition"), *after a while*. Like after a million repetitions. *Just kidding.* 😊

As indicated, a function call is an expression (which can also be used as a statement by itself). An expression has a value, by definition. And, the value of an expression is printed in the interactive mode to the terminal (but not in the non-interactive/program mode).

So, what is the value of the expression `print(_)`? Let's try the following in the Python REPL:

```
>>> print(print(_))
3
None
```

Very very interesting. Since `print(_)` is an expression, a value, we can use it as an argument to (another) function call `print()`. So, we can do `print(print(_))`. What does that even mean?



Raise your hand if you don't know what an "argument" is. We did not define the term, but we used it before. Something that is inside the pair of parentheses in a "function call" is an argument(s). In fact, functions can take zero, one, or more than one arguments (separated by commas), depending on their definitions, as we will see later in the book.

Raise your hand if you don't know what a "function" is. Raise your hand if you don't know what a "parenthesis" is. 😊

In a function call, an argument(s) is evaluated first before the interpreter actually "calls" the function. In the statement `print(print(_))`, the argument of the outer `print()` call is an expression `print(_)` (whose function argument is `_`). We know the value of `_`. It is `3` at this point. Then, `print(3)` is evaluated next, which is the argument of the outer `print()`. Calling `print(3)` has a side effect, as stated above. It prints out `3` to the terminal. That is what we see in the sample output, in the first line. But, it is not the value of `print(3)`.

Next, the outer `print()` function call also has a side effect. It prints out its argument's value. The output shows the value of `print(3)` as `None`. That is the output we see above, `3` and `None`, in two separate lines. The Python REPL does not print out the value of a stand-alone expression when it is `None`.

4.8. The **None** Object

So, what is **None**?

```
>>> print(None)
None
>>> type(None)
<class 'NoneType'>
```

Interesting. **NoneType** is another built-in type in Python just like **int** or **float**. There is only one valid value of type **NoneType**, which is **None**.

None has a number of uses in Python. In this particular example, it simply means that the `print()` function does not "return" any meaningful values like `100` or anything else (e.g., to the "caller"). *It returns none.*

Now we can guess what would be the value of `print(print(_))` as an expression. Of course, it is **None** because the (outer) `print()` function returns **None** regardless of

the value of its argument.

The *side effect* is a "dirty word" in functional programming. In pure functional programming languages like Haskell, for example, functions have no side effects, *by definition*. The modern trend in software engineering is adopting various functional programming principles to reduce errors in creating software, among other things. And, "no side effect" is considered a good thing.



Python is fundamentally an imperative programming language. That is, it is based on executing a series of statements, and it is not based on the applications of "pure functions" (as in functional programming). Nonetheless, a lot of functional programming techniques are used in the *modern* Python programming. The "statements" have no place in functional programming. On the other hand, "expressions" are much more versatile, and they can be used in both imperative and functional programming styles.

4.9. Boolean Expressions

So far, we have discussed a few builtin types, namely, `NoneType` and the two number types, `int` and `float`. We also have been using strings.

As previously indicated, `bool` is also a type, a "subtype" of `int`, which represents *logical values*. There are only two values for type `bool`. `True` and `False`. Their meanings are almost obvious.



Note the capitalizations in the literals `True` and `False`. In Python, and in virtually all other high-level programming languages, names are case-sensitive. For example, `true` is not a `bool` literal.

We can give another (informal) definition of *type* at this point. A type is a "set" of values (e.g., a mathematical set). A set of two values, `True` and `False`, is a type, called `bool`. The `NoneType` type happens to include only one element, `None`.

Likewise, a set of (a large but finite number of) values like -1, 0, 1, 2, 3 is called the `int` type. The same with `float`. (Or, with any other types in Python.) It is harder to conceptualize because there can be so many different values, but the `float` type is a (ultimately finite) set of these real number values.

An expression that evaluates to a `bool` value, either `True` or `False`, is called a Boolean expression. Hence `True` is a (trivial) Boolean expression, and so is `False`.

```
>>> True
True
>>> type(False)
<class 'bool'>
```

As stated, `bool` is a subtype of `int` in Python.

```
>>> isinstance(True, bool)
True
>>> isinstance(3, bool)
False
>>> isinstance(True, int)
True
>>> issubclass(bool, int)
True
```

We are using two other builtin functions, `isinstance` and `issubclass`, in this example. We will not discuss what exactly these functions do at this point, but the example is sort of self-explanatory.

As stated, a "function call" is an expression and it has a value. The Python REPL prints out the value of each expression (unless it is `None`). That is, the value of `issubclass(bool, int)` is `True`, for instance. We can easily interpret what this means even without knowing the precise definition of the `issubclass` function. The same with the `isinstance` function. For example, `isinstance(True, int)` is true, indicating that the value `True` is of the type `int` (possibly, among other types).



Again, we are using the terms like "subclass" and "instance" without giving their precise definitions. *Pay attention to the overall contexts where these terms are used.*

The (special) values like `True` or `False`, as well as `10` or `5.5`, etc., are called the "literals" in programming, as mentioned before. (They are *special* in that they have a special syntax (although it may not be obvious at this point). Only (some of) the "builtin types" have the literal syntax.) `True` and `False` are `bool` literals. Likewise, `1000.0` and `0.0001` are `float` literals.

The integer literals, for example, have to follow certain literal syntax rules. Although we do not use in this book, integers can be written in the bases, 2, 8, and 16 (binary, octal, and hexadecimal, respectively) in addition to the base 10 (the regular decimal numbers). Regardless, all integer literals have to start with a digit (0 through 9), among other things.



For example, `0b11` is a binary number, `3.011` is an octal number, `9`, and `0x11` is a hexadecimal number, `17`. They all start with a digit, in particular 0 for all non-decimal numbers.

Binary numbers start with `0b` and they have to be followed by at least one number. Otherwise it is not a binary number literal. Likewise, hexadecimal numbers have to start with `0x` and they likewise have to be followed by at least one number. All other numbers that start with 0 are octal numbers. Again, octal numbers have to have at least one additional digit. The number 0 is considered a decimal number, base 10.

4.10. Dynamic Typing

Python is a "dynamically typed" programming language, meaning that although all objects in Python have types, they are only checked (or enforced, if you will) at run

time.

In contrast, in the "statically typed" languages, the types of the variables are verified at build time. For example, if you call a certain method on a certain object of a certain type, the compiler checks if the method is allowed on that object of that specific type. If not, it throws a compile time error. For dynamically types languages, this type validation, if you will, happens at runtime.



If you are coming from other C-style languages such as C/C++, Java, or C#, *conceptually*, types are associated with "variables" in C-style languages. In Python, however, types are associated with values. (Or, more precisely, with the "objects". An object in Python has a type and a value, as briefly stated before.) This is an interesting distinction (albeit abstract), which can be helpful in understanding certain important concepts in Python.

Python is also a "loosely typed" language, so to speak. That is, the type of an expression, or a variable, can (implicitly) change *depending on the context*, among other things. (Again, an important distinction. The type of an object does *not* (normally) change.) As we will see later in the book, a certain non-Boolean expression can be evaluated as **True** or **False** where a Boolean expression is expected. This is rather uncommon, or almost non-existent, in the statically typed, and "strongly typed", modern programming languages.



It is interesting to note that the two *currently* most popular programming languages, Python and Javascript, are dynamically, and loosely, typed languages. Virtually all other widely used languages, C/C++, Java, C#, Go, Swift, Kotlin, Rust, ..., are statically typed.

4.11. Builtin **bool** Function

Python has a builtin function named **bool**, which explicitly converts a non-Boolean

4.11. Builtin `bool` Function

expression to a Boolean value. There are many such functions in Python that create a value of one builtin type from an expression of another type. (They are often called the "constructor functions", which will make sense once you understand the "secret of Python". 😊)



We often use the terms like "conversion" or "casting" in programming. In Python, "conversion" does not mean *morphing* an object from one type to another, which is not possible for the objects of the builtin types. Even for the user-defined types, that is not commonly done. We will discuss further what these "constructor functions" do, throughout this book.

Let's try the `bool` function to see what kind of conversion happens for objects of the number types. For integers,

```
>>> bool(100)
True
>>> bool(1)
True
>>> bool(0)
False
>>> bool(-5)
True
```

For floating point numbers,

```
>>> bool(5.5)
True
>>> bool(0.0)
False
>>> bool(-10.75)
True
```

As we can see from these few examples, numbers are generally converted to **True** except for **0** (**int**) and **0.0** (**float**), whose Boolean values are **False**.

What about **None**?

```
>>> bool(None)
False
```

The Boolean value of **None** is always **False**.

4.12. Simple and Compound Statements

We have used simple statements before. For example,

```
>>> print("Hola!")
Hola!
```

Just to get the taste of what's coming, let's try a slightly more "complex" statement (literally). First, try typing **if 3 > 0:**, with the trailing colon (:), into the Python REPL (and press Enter):

```
>>> if 3 > 0:
... 
```

The prompt changes from "**>>>**" to "**...**" indicating that the statement is not complete. Let's finish this statement:

```
>>> if 3 > 0:
...     print("Bonjour!")
...
Bonjour!
```

4.12. Simple and Compound Statements

Note that we typed `print("Bonjour!")`, with the leading spaces (how many there are is not that important at this point), and pressed Enter *twice*. (Can you see that? 😊) As indicated earlier, the leading spaces indicate that the line "belongs" to the first less-indented line somewhere above (that ends with a colon, ignoring white spaces). These two lines, in this example, form a single "compound statement". In fact, we could have inputted it in one line:

```
>>> if 3 > 0: print("Bonjour!")
...
Bonjour!
```

Note that, even in this case, we had to press Enter *twice* to execute the statement. The reason is that the statement might have continued, and the Python REPL cannot tell whether this (compound) statement is complete or not. For example, the user might have done this:

```
>>> if 3 > 0:
...     print("Bonjour!")
...     print("Guten Tag!")
...
Bonjour!
Guten Tag!
```

The user could have even typed more than two statements (e.g., `print(...)`) as part of this "compound statement". Again, the double Enter's indicate the end of the compound statement. (In the non-interactive mode, this is not required, although it is still a good practice to include an empty line after a compound statement.)

Note that the two statements inside the "suite" (`print("Bonjour!")` and `print("Guten Tag!")`) are *aligned*. The number of the leading white spaces is not important, but they should be the same within a suite. (Again, these terms are not important as long as you know what we are referring to. Remember, *context, context, context*. A set of the two statements in this example makes up a "suite" under

```
if 3 > 0:.)
```

Just to be consistent, you should use the same amount of indentations in the same program (and, even across different programs). The Python style guideline recommends *four spaces*. It should be noted that a compound statement may include another compound statement, and so on, and the indentations can be nested. So, it will be like 4 spaces, 8 spaces, 12 spaces, etc. as we go down the indentation level.

In the interactive mode, however, this is not that important. As long as you use the same indentations within the same suite (e.g., 2 spaces, or even a tab, which is usually not used in program files), that should be fine.

4.13. Conditional Statement

Although it is not a focus of this lesson, `if` is a "Python keyword". It is used to create a "conditional statement". The `if` keyword is followed by a Boolean expression (e.g., `3 > 0`), and if the expression evaluates to `True`, then the statement(s) within the given suite are executed. Otherwise, those statements are ignored by the Python interpreter.

In the examples above, the expression `3 > 0` is trivially `True` (e.g., the answer to the question "is 3 bigger than 0?" is always "yes"), and hence the `print()` statements following the `if` line are executed, whose side effects are printing the string argument(s) to the terminal.

The `if` "clause" is often followed by one or more other "conditional clauses" or an (optional) `else` clause (using keywords `elif` and `else`, respectively). We will discuss this further later in this book. (A "clause" in the `if` statement, for instance, refers to a suite and a line that precedes the suite, e.g., `if 3 > 0:` in this example. A compound statement can include one or more clauses.)

Note that, unlike in many other (C-style) programming languages, at least one statement (simple or compound) is needed within the `if` suite (or, any suite). If you have nothing to execute (which is generally unlikely but not implausible), then the venerable `pass` statement will do. ☺

```
>>> if 3 > 0:
...     pass # This whole if statement is pretty much useless ;)
... 
```

4.14. Strings in Python

Another important builtin type in Python is "string" (or, `str`), which we have been using throughout this introductory part.

A string essentially represents a sequence of "characters" (e.g., English alphabets or digits, etc.). (Python, however, does not have a separate character type unlike many other programming languages.) We have seen a few examples of the "string literals" before. For example, `"Hello Monty Python"` is a string literal.

In Python, we can use a pair of *matching* single quotes (`'`) or double quotes (`"`) to denote a string literal. For instance, `'Hello Brian'` is also a string literal. As far as the Python interpreter is concerned, there is no difference whether we use a pair of single quotes or a pair of double quotes. Some people prefer one or the other, but it is really a matter of preference.



The pairs of (single/double) quotes are used for a number of different things in Python (e.g., with various "prefixes") other than for the string literals. We will discuss some of them, including the "f-string" expression, later in the book.

Also note the "special syntax" that we use for the string literals. Numeric literals are so plain that it is hard to recognize that they are "special", but in fact they are special, as we have seen before, e.g., with the integer literals. All literals in Python, by definition, have certain "special syntaxes".

Python supports another kind of string literals, namely, the "long strings", which are sometimes called the "multiline strings" (although they do not always have to span

multiple lines). A long string uses a pair of *matching* triple single quotes (`'''`) or a pair of matching triple double quotes (`"""`). It can include certain characters that are not normally allowed (e.g., without "escaping") in the regular (short) strings, including newlines. For example,

```
>>> """Only those who will risk going too far
... can possibly find out how far one can go."""
'Only those who will risk going too far\n can possibly find out how far one
can go.'
```

Note the continuation prompt in the second line (`...`). We pressed Enter at the end of the first line, after "too far". The *value* of this multiline string literal is just a string, `"Only those ... one can go."`, including the newline character (`\n`) in the middle. This newline is the one from the (invisible) newline character after "too far" in the input. (Can you see it? 😊)

The multi-character sequence (`\` and `n`) is called an "escape sequence" or an "escape character". It represents *one* character, a newline in this case. As another example, `\t` (backslash + `t`) represents a tab. As seen from the above example, characters like newlines or tabs can be directly included in the multiline string literals without escaping.



Inside the double quoted string literals (`"..."`), the double quotes need to be escaped, e.g., as `\"`. But single quotes do not require escaping. Likewise, inside the single quoted string literals (`'...'`), the single quotes have to be escaped as `\'`. But double quotes need not be escaped. Sometimes this can be a good reason to pick single quote pairs vs double quote pairs for short string literals.

4.15. String Concatenations

As briefly alluded earlier, strings can be concatenated using the addition `+` operator. That is, the `+` operation returns a new concatenated string if both operands are

4.16. Ending Python Interactive Session

strings.

```
>>> "hello " + "universe"  
'hello universe'
```

In case *both operands* are string literals, the `+` operator can be omitted. (Incidentally, this kind of language features are often informally called the "syntactic sugar". What a name, although they could have been just as *sweet* "by any other name", a la Shakespeare. 😊)

For example,

```
>>> "hello" " " 'universe'  
'hello universe'
```

In this (somewhat convoluted) example, *two* string concatenations have been performed (from left to right). First, `"hello" + " "`, which yields `"hello "`, and then `"hello " + "universe"`, which evaluates to the final result shown in the example `"hello universe"`. This example mixes single quote and double quote string literals, for illustration. As stated, however, it is best to be consistent (unless there is a special reason otherwise). These "short strings" can also be concatenated with long strings.



Does it make sense to you why the evaluation of this particular (three argument) expression involves *two* operations and not one?

4.16. Ending Python Interactive Session

Once you are done using the Python REPL, you can exit by calling the builtin functions `quit` or `exit`:

```
>>> exit()
```

\$

Or, you can just use the EOF signal (End of File). That is, **Ctrl+D** on Unix/Linux platforms (or, **Cmd+D** on Mac) and **Ctrl+Z** on Windows.

4.17. Summary

In this lesson, we learned a few "simple types", or primitive types, in Python.

There is only one value **None** for the type **NoneType** in Python. The **None** value is used in special contexts as we will see throughout this book. There are only two logical values, **True** and **False**, for the type **bool**. An expression can be evaluated to a **bool** value using the builtin **bool** function. (The fact that the type **bool** and the function **bool** have the same name is not a coincidence.)



If you are coming from the C-style languages, Python's **None** is different from **null** in those languages. It is also different from **undefined** in Javascript. Python has no values, or syntax, corresponding to either **null** or **undefined**.

There are two number types, **int** and **float**. The values (or, the objects) of the **int** type are the whole numbers, or integers. The values of the 'float' type are the real numbers, or the *floating point numbers* as we call them in programming. As indicated before, objects of all simple builtin types, including **int** and **float**, are *immutable*.

In addition, we used a few additional "builtin functions of Python" like **type**, **print**, **isinstance**, and **issubclass**, etc. Although we did not explicitly define what a *function* is, we did see many example uses of the functions throughout this and the previous lessons.

A function, roughly speaking, is a sequence of statements packaged as a unit, so to speak. Functions are one of the basic constructs of Python for organizing, and sharing, Python code.

4.17. Summary

A function takes zero or more arguments (depending on its precise definition), and it returns exactly one value, which can possibly be **None**. That is, we "call" a function with certain argument values, or objects, if needed, and we handle the return value, unless it is **None**.



It can be confusing, but we sometimes say that a function does not return a value when the function returns **None**.

Chapter 5. Tuples, Lists, and Some Inspirations



5.1. Complex Types

We will continue exploring some more basics of Python using the Python REPL, as part of our "tour". As we have seen earlier, the REPL lets you easily evaluate Python expressions or run Python statements.

In the previous chapter, we learned the simple type literals like `True` and `False`, or other `int` and `float` numbers as well as the (short and long) string literals. As we pointed out, all objects, and all expressions, in Python, including the simple literals, have types. These are the "simple types". The objects of the simple types in Python are "immutable".

5.2. Tuple Literals

In this lesson, we will learn "complex types" and "complex literals". In particular, we will learn some basic builtin types like `tuple` and `list` in Python. They are also called the "compound types" because they are built from other types. Python's builtin complex types, some of which we will discuss in this lesson, and the user-defined types, which we will discuss later in the book, are built using these (simple and complex) types as building blocks.

5.2. Tuple Literals

Once again, let's begin by starting a Python shell:

```
$ python
Python 3.10.4 (main, Jun 29 2022, 12:14:53) [GCC 11.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Let's try typing a pair of parentheses (or, round brackets):

```
>>> ( )
()
```

The value of a pair of parentheses is a pair of parentheses. The pair of parentheses, without any content inside, is an empty "tuple literal". (White spaces are ignored.)

A `tuple` in Python is a sequence of other objects, zero, one, or more. In a `tuple`, the element objects (values or names, or expressions) are separated by commas `,`. Here are a few examples of the tuple literals.

```
>>> (1,)
(1,)
>>> (1, 2)
(1, 2)
>>> (1, True, 100)
```

```
(1, True, 100)
>>> (None, (5, 55))
(None, (5, 55))
```

As we can see, a tuple can contain elements of different types (simple or complex, or user-defined). A tuple can even contain other tuples. (In the example, `(5, 55)` is a tuple, which is an element of another (two element) tuple, `(None, (5, 55))`.)

An interesting thing to note is that when there is only one element in a tuple (e.g., `(1,)`), the trailing comma is required. Otherwise, the Python interpreter will not see it as a tuple. This is because the parentheses are often used for "grouping" (as well as for function calls, etc.), and that meaning takes a precedence over the tuple literal syntax when there are no commas.

For instance,

```
>>> (100)
100
```

That is, `(100)` is an `int` literal `100`, and not a tuple. The enclosing pair of parentheses is redundant in this case.

5.3. Expression List

In Python, we can evaluate multiple expressions at the same time. For example, instead of evaluating `100` and `200` separately (whose values are trivially `100` and `200`, respectively), we can compute them together in one go, by listing them together, separated by commas, like `100, 200`. This is called the "expression list" in Python.



Although we informally use the phrase "at the same time" here, more precisely speaking, the expressions in an expression list are evaluated *from left to right*.

5.4. Tuple Type

For instance,

```
>>> 50 + 50, 100 * 2
(100, 200)
```

The Python interpreter first computes `50 + 50`, which is evaluated to `100`, and then it computes `100 * 2`, which is evaluated to `200`.

The *value* of the above expression list (e.g., two expressions separated by a comma) is a tuple, e.g., as represented by a tuple literal `(100, 200)` in this case.

As in the tuple examples earlier, a single expression will be evaluated to a single value. If you need a one-element tuple result, then you will need to add a trailing comma at the end of the expression. For example,

```
>>> 1 / 2,
(0.5,)
```

In this case, the input `1 / 2`, is an expression list (comprising one expression), and not a single expression. Likewise, the result is a tuple (with a single element), and not just a single number. As we learned earlier, the division in Python (`/`) always yields a `float` number, unlike in many other (C-style) programming languages, even when both operands are integers as in this example.

5.4. Tuple Type

The type of a tuple literal is `tuple`. For example,

```
>>> type((1, True))
<class 'tuple'>
```



If you are new to programming, then the syntax can be a bit

confusing. We are using two pairs of parentheses in this expression, `type((1, True))`. The outer pair is used for the "function call" (for the function `type`). The inner pair is part of the tuple literal, `(1, True)`.

Each element in a tuple can be accessed using the "index notation" (`[]`). The elements in a tuple are given an integer "index" starting from `0`. That is, the first/leftmost element is given the index `0`, and the indexes increase by `1` as we move to the right. The last/rightmost element ends up having the biggest index in a given tuple. In Python, a pair of square brackets (`[]`), e.g., after a sequence type object, is called the "subscript operator".

For example, the first element in a tuple, `(1, 3)`, is `1`, and its index is `0`. The index of the second element `3` is `1`.

Using the "index notation", we can get the value of the first element. For example,

```
>>> (1, 3)[0]
1
```

Likewise, the notation for accessing the value of the second element is `(1, 3)[1]`, which evaluates to `3`. Note again that `(1, 3)` is a tuple, a *single* object (although it, as a compound type object, *includes* other objects, e.g., `1` and `3`).

As an example, we can compute the sum of the two elements in a tuple `(1, 3)` as follows:

```
>>> (1,3)[0] + (1,3)[1]
4
```

Or, using the implicitly defined `_` variable (which only works in the Python REPL), we can do this:

5.4. Tuple Type

```
>>> (1, 3)
(1, 3)
>>> _[0] + _[1]
4
```

This works because the (strange-looking) name `_` is "bound" to the tuple object `(1, 3)` when we (trivially) compute its value (the first expression). The value of `_`, when we compute the sum (the second expression), is `(1, 3)`.



What will happen if we do the same evaluation, `_[0] + _[1]`, one more time right after this last (sum) evaluation? You can try it in REPL if it does not seem obvious to you.

Not to give away the answer 😊, but if you are getting an error, why do you think that is? Note that the subscript operator can only be used with objects (and names) with particular types, known as "sequences", like `tuple` or `list`, or other collection types. This is another example as to why the "types" are crucial in programming.

Incidentally, the type of both first and second elements of this tuple, `(1, 3)`, is `int`.

```
>>> type((1, 3)[0]), type((1, 3)[1])
(<class 'int'>, <class 'int'>) ①
```

① Why is this enclosed in parentheses?

On the other hand, as stated, the type of this tuple is `tuple`. Or, more precisely, it is a `tuple` of `int` and `int` (two items). In the type annotations, it is denoted as `typing.Tuple[int, int]`. (The prefix `typing`, before the dot `.`, is the name of the *typing module*.)

The *length* of a tuple, that is, the number of elements, or items, in the tuple, can be computed using the Python builtin function, `len`. For example,

```
>>> len((True, False, "Hello", 123.0))
4
```

The type of the argument, the four-element tuple literal in this example, is `tuple`. Or, more precisely, it is `typing.Tuple[bool, bool, str, float]` in the type annotations. As stated, it is not uncommon that the elements of a tuple have different types.

5.5. List Literals

The `list` type is another important builtin compound data type in Python. `list` is a "sequence type", just like `tuple`. A list object is essentially a dynamic size array.

Most programming languages natively support an "array" collection type, which is a sequence of elements in the contiguous memory space. In Python, the array type is called the `list`, and unlike most builtin array types in other languages, the length of a list in Python can change (e.g., by adding more items into the list).



The Python standard library includes the `array` module, which defines the `array` type for the builtin (numerical) types. This `array` type is, however, used only in special circumstances. Other popular libraries like `numpy` also include array types which can be used in place of the builtin `list` type.

A Python list value is represented with square brackets `[]`. (Not to be confused with the index notation.) For example, this is an empty "list literal":

```
>>> []
[]
```

A list of one element `100`:

5.5. List Literals

```
>>> [100]
[100]
```

Note that, unlike in the case of tuples, we do not need a trailing comma for a one-element list.

A list that contains three elements, `1`, `True`, and `10`:

```
>>> [1, True, 10]
[1, True, 10]
```

The type of a list object is `list`:

```
>>> type([True, False])
<class 'list'>
```

As with tuples, the order is important. That is, for instance, `[1, 10, True]` is a different list from `[1, True, 10]`.

In fact, the tuples and the lists are rather similar in Python. There are a few important differences, however. First of all, a tuple is a fixed size whereas the length of a list can change. We will see shortly how we can do that with lists.

The second important difference is that a tuple object is "immutable". That is, once created, (roughly speaking) the value of a tuple cannot change. We cannot directly change its elements. On the other hand, as we will see shortly, we can change the value, or content, of a list object using various methods. That is, a list in Python is "mutable".



In Python, the concepts of *mutability* and *immutability* are more complicated than the simple description that we have just given here. More on this later.

Another difference in the way we use tuples and lists is that, although elements of different types are allowed for both tuples and lists, it is generally considered a good practice to use lists with the same element types.

That is, `[1, 10, 'happy']` is not a good list since it mixes `int` and `str` objects (although it is grammatically valid).

In the type annotations, the type of both `[1, 2]` and `[100, 200, 300]` is `typing.List[int]`. The same for `[5]`, `[1, 2, 3, 4, 5]`, etc., that is, regardless of how many elements are in the lists. (Contrast this with the type annotation for the tuples.)

Just to be clear, there are times and places for the mixed type lists. For such a list, however, we view the type of its items as its common "ancestor" type, or the ultimate base type `object`. Therefore, conceptually, even in such cases, the lists are "homogeneous". All elements have a single element type, e.g., `object`.



Python typing also supports the "union types", as we will briefly mention later in the book.

As with tuples, the `len` builtin function can be used to get the number of elements in a list. For instance,

```
>>> len([1.0, 1.5, 2.0])
3
```

Likewise, we can access each element in a list using the "index notation". For example,

```
>>> [10,100,1000][1]
100
```

The value of this expression is `100`, that is, the second element of the given list.



Again, this kind of syntax can be confusing to beginners. It takes a bit of getting used to. In this particular case, the first pair of square brackets is part of the list literal `[10,100,1000]`, whereas the second pair is part of the index notation, referring to the second element (`[1]`) of the list.

5.6. List Operations

As suggested above, we can change the content of a list, e.g., through "assignment".

For instance,

```
>>> [10,100,1000]
[10, 100, 1000]
>>> _[1] = 5
>>> _
[10, 5, 1000]
```

In this example, we have a list of three elements, `[10, 100, 1000]`, and we change the second element of the list to `5`. (Note that `5` is an *object*, just like everything else in Python, whose value happens to be `5`, of the `int` type.)

This is done through an assignment `_[1] = 5`. In mathematics, the `=` symbol is the equality operator. On the other hand, it represents an assignment in Python, and in many other programming languages.

In an assignment statement, what's on the left-hand side of the assignment operator (`=`) is replaced/modified by the value on the right-hand side (int `5` in this case). If we view the value of the variable `_` bound to the object `[10,100,1000]`, its value is now different. It is `[10, 5, 1000]`. As we pointed out earlier, a list object is "mutable".

It is important to note that it is still the same object. Only its value has changed. (Remember what we talked about regarding the "objects" (with "unique identities")

and their "values"? Objects' identities do not change. Only their values can change.)



The assignment statement in Python can have slightly different "semantics" (or, meanings) depending on the context. We will see a few different uses of assignment throughout this book.

We can append an element(s) at the end of a list using the builtin "list method", `append`.

```
>>> [10,100,1000]
[10, 100, 1000]
>>> _
[10, 100, 1000]
>>> _.append(10000)
>>> _
[10, 100, 1000, 10000]
```

As before, we define an example list `[10,100,1000]` first. The implicit variable `_` is then automatically bound to this list object. We verify that by evaluating `_`.

Now we can add an additional element, say `10000` in this example, to the object using the builtin `append` method. Note the syntax. Instead of calling it like other functions, it uses the "dot notation". It calls the `append` method *on the object*. ("The functions" and "methods" in Python are pretty much the same except for, primarily, this syntactic difference.)

An object of the `list` type has certain "attributes" associated with it, which automatically come from the `list` type (when the object is created), and the `append` method happens to be one of them. We use the dot notation to access an attribute of an object.

Note that the name `_` currently refers to the object `[10,100,1000]` at this point, and hence calling the method on `_` is more or less equivalent to calling it on `[10,100,1000]`.

5.6. List Operations

One caveat is that, for immutable objects, we can mostly deal with objects themselves. However, for mutable objects like the lists in this example, the values of objects can change and hence we cannot really refer to an object by its value alone. We need something "more permanent" or something "invariant" regardless of its current value.

This is where the idea of "variables" comes in. In Python, a variable is simply a name for an object. Interestingly, or not so interestingly, an object can have no name, one name, or more than one names, at any given moment.

Python includes a number of builtin methods for `list` ("built in" to the Python interpreter program). We will use a few of them later in the book. Note, however, that this book is not a language reference. The readers are encouraged to continue learning Python, that is, *after finishing this book*. ☺

All imperative programming languages use variables. In fact, the concept of variables is one of the most important components of the imperative programming.

In C-style languages like C/C++, Java, or C#, variables are like "containers" (like bowls or glasses or plates). You put something (e.g., a value) in the container. Later, you can remove the existing content and put something else in the container. *A variable holds a value*, and the program manipulates these variables to do computation.



In contrast, the variables in Python are just names. Objects exist (with unique identities), and we use variables, or names, to refer to these objects.

These two different interpretations (e.g., "variable-centered" vs "object-centered", if you will) often lead to the same conclusion. But, that is not always the case. For example, in Python, there are no pointers or reference types, which are rather common in most C-style programming languages. The variables in Python are neither

values nor references. They are just names.

This is a rather advanced concept, and if it does not make sense to you, then that is perfectly all right. In fact, if Python is your first programming language (and if you have no plans to learn any other languages for the rest of your life ☺), then it is actually of little significance. You can just ignore this. On the contrary, if you are coming from other C-style programming language, then it is rather important to realize the difference.

In C-style languages, we mainly deal with variables. A variable may happen to be associated with a certain value at any given time. Or it may not. A variable is `null` when it does not have any value associated with it.

In contrast, in Python, we mainly deal with objects. The variables are secondary. They are just names to refer to these objects. In fact, the variables are called the "references" in Python. (Not to be confused with the terms, references or pointers, used in other C-style languages.) In many cases, however, we *do* need names and we *do* use variables in Python programs, as we will see throughout this book.

In Python, a variable cannot be `null` since that does not make sense. A container (in C-style languages) may not contain anything, which is indicated by the `null` value. In Python, that does not make sense. A name cannot exist without an object that it is referring to. Python's `None` has different meanings and uses from those of `null`. (Incidentally, as a corollary, Python does not have this *infamous* "null pointer exceptions". ☺)

One other thing to note is that neither an object nor a reference in Python is a "value", as just stated. In Python, an object *has* a value (and a type), and the value (of an object of a mutable type) can change, in general. We use a variable, or name or reference or alias,

to refer to this object (not its value). We will continue this discussion in the next section, and throughout the book.

5.7. "Names" in Python

In the Python interactive shell, as stated, the special name `_` is automatically bound to the "last evaluated" object whose value is not `None`. This name can refer to only one object at a time.

The statement `_.append(10000)` above, for example, does not alter the name `_`. Before and after the statement, it refers to the same object. The `append` method does not return any value. (Or, it returns `None`.) This list method only alters the value of the list object (as its "side effect").

Its value was `[10, 100, 1000]` before the `append()` call, and it became `[10, 100, 1000, 10000]` by appending a value `10000` at the end of the list. This is verified by evaluating `_`, after the operation, as shown in the above example.

The "opposite" of `list.append` is `list.pop`. The `pop` method removes, or pops, the last item, if any, from a given list. In this case, we need to introduce a new name to refer to the existing list. The `_` variable is not sufficient. Here's an example:

```
>>> [10,100,1000]
[10, 100, 1000]
>>> _.pop()
1000
>>> _
1000
```

In this example, `_.pop()` removes the last element/item from the given list, `[10, 100, 1000]`, and it *returns* the removed, or popped, item as its value. That is, when we call the `pop` method, the `_` name refers to the list object `[10, 100, 1000]`. And it will remove the last item `1000` from the object. The value of the object will be `[10, 100]`. But, there is no way to verify that. There is no way to refer to this object any

more since the `_` name now refer to `1000`, the removed item via `pop()`. (This is because `_` always refers to the "last evaluated value", which happens to be the value of the method call `_.pop()`. As stated, the value of a function or method call is the value returned by the called function/method.)

The object, whose values are `[10, 100, 1000]` and `[10, 100]`, before and after the call, respectively, now has no name. Clearly, it is rather inconvenient to use an object without any names (as in "impossible" ☺), especially for the *mutable objects*.

To demonstrate the use of the `pop` method, therefore, we will need to explicitly name a list object. Here's an example:

```
>>> a = [10, 100, 1000]
>>> a.pop()
1000
>>> a
[10, 100]
```

The first line `a = [10, 100, 1000]` is an assignment. It "binds" the (new) name `a` to an object `[10, 100, 1000]`. (The choice of `a` is just arbitrary. Any (valid) name can be used.) Now we can refer to this object, the *same object* (not just an object with the same value), with the name `a`. In Python, this type of assignment is a statement (although there is a separate assignment expression), and hence it has no value. In this example, note that the Python interpreter does not print anything, after the assignment statement (the first line), because it is a statement.

Now, after calling `a.pop()`, whose return value happens to be `1000` (the last item of the list `a`), the value of the object which `a` is referring to is now `[10, 100]`. It is still the same object. It is only its value that has changed.



In the text, we often use the value of an object to informally refer to the object itself. For example, we say "the object `[10, 100, 1000]`". What this really means is "the object whose (current) value is `[10, 100, 1000]`". For the immutable objects, it matters little

since their values never change (while the program is running). "Objects" and "values" are almost synonymous (except when it matters 😊). For the mutable objects, however, we will often need to be careful as to what exactly we are really talking about, *objects* or their *values*.

5.8. Assignment

Assignment is one of the "simple statements" in Python. The left-hand side of an assignment statement can be (1) a name referring to an object, or (2a) an attribute or (2b) item of a mutable object. (An attribute is a field or method of an object.)

First, the assignment statement is used to bind a new name, or rebind an existing name, to an object. In the above example, for instance, a new name `a` is introduced and it is bound to the list object via the assignment.

As indicated before, a variable in Python is a reference, or an alias, to an object. The names are often used to merely refer to objects, and the program primarily manipulates the objects (via their names). This may be called the "object-centric" view, as we briefly alluded before. This viewpoint is generally useful in Python programming, but it is more so when dealing with the mutable objects.

On the other hand, variables may play a central role, e.g., to keep track of the "program state", while being assigned to different objects while the program is running. This may be called the "variable-centric" view, and it is an essential part of programming when using the imperative programming languages, including Python. (We will see a number of examples throughout this book.)

Here are a few examples of assignment:

```
>>> x = 30
>>> y = x
>>> z = 30
>>> x, y, z
```

```
(30, 30, 30)
```

The name `x` refers to the `int` object `30`. Likewise, the names `y` and `z` refer to the same object `30` that `x` references (because `30` is an immutable object).

For the mutable types,

```
>>> w = [1, 2]
>>> p = w
>>> q = [1, 2]
>>> w, p, q
([1, 2], [1, 2], [1, 2])
```

It appears to work the same way. The name `w` references a list object `[1, 2]`. So does `p`. It refers to the same list object `[1, 2]`. However, the variable `q` refers to a different list object, albeit with the same value `[1, 2]`.

```
>>> id(x), id(y), id(z)
(140593326933136, 140593326933136, 140593326933136)
>>> id(w), id(p), id(q)
(140593324707200, 140593324707200, 140593324867456)
```

All three variables `x`, `y`, and `z` refer to the same object, whose `id` happens to be `140593326933136`. (The specific `id` values are not important. They will likely change from program run to run.)

On the other hand, the variable `w` and the variable `p`, which has been bound to the object that `w` refers to via the assignment `p = w`, reference the same object while the variable `q` is a name for a different object, i.e., with a different `id`.

Note that assigning a name to a different name ends up both names referring to the *same object*, whereas assigning a value to different variables *can* result in them pointing to different objects, e.g., in case of the mutable types.

5.8. Assignment

In Python, we can also give multiple names to a single object in one statement. For example,

```
>>> x = y = "dragon"
>>> p = q = ['a', 'b']
>>> id(x), id(y)
(140593323269680, 140593323269680)
>>> id(p), id(q)
(140593324864960, 140593324864960)
```

The variables `x` and `y` refer to the same object, and `p` and `q` refer to the same object.

One thing to note is that, especially for the beginners in programming, although it is not an assignment, and we do not generally use the term "assignment", there are a couple of situations where exactly the same semantics is used as that of assignments.

When we call a function with an argument, e.g., an object or a name, this object, or the object referred to by this argument, is assigned to the corresponding variable/parameter inside the function. That is, when a function is called, the parameters of the function, if any, are all bound to the respective argument objects.

Another situation is when a called function returns an object. In such a situation, there may not be an explicit name to be bound to the returned object in the calling context, but the same semantics applies, e.g., when the returned object, or more precisely the (implicit) reference to the returned object, is subsequently used, for example, in another explicit assignment (to another variable). We will see some concrete examples while working on our main project.

As stated, the assignment statement is also used to update the value of a "data attribute" of a *mutable object*, among other things. Likewise, the items of a mutable sequence or other collection type objects can be changed using assignments, as we saw earlier, e.g., in the context of indexing. We will discuss this further later in the book.

5.9. Slicing

Speaking of "indexing", we briefly looked at the indexing expressions earlier. An element of an object of a sequence type can be addressed, or referred to, using the subscript operator. For example,

```
>>> a = ['a', 'b', 'c', 'd', 'e']
>>> a[1], a[2], a[4], a[-1], a[-5]
('b', 'c', 'e', 'e', 'a')
```

The valid indexes are from `0` to `len(a) - 1`, or, `0, 1, 2, 3, 4` in this example. Python will raise an `IndexError` exception if we use an invalid index.

Note that we can even use a negative index. Negative indexes run from `-1` to `-len(a)` starting from the last element and moving to the left, that is, `-1, -2, -3, -4, -5` from right to left in this example. This is almost like using a modulo operation, using the length of the list as a denominator. For instance, `a[-1]` is the same as `a[-1 % 5]`, which is `a[4]`, etc. Using any invalid negative index will also raise `IndexError`.

Python supports another similar operation called "slicing", as with many other *modern* programming languages. Unlike indexing, however, the slice operation returns a sequence type object, e.g., an object referring to a part of the original sequence, roughly speaking. Tuple slicing returns another tuple. List slicing returns another list. String slicing returns another string.

Slicing uses two indices to specify a region, or range. For instance, using the same object `a` from the example above,

```
>>> a = ['a', 'b', 'c', 'd', 'e']
>>> a[1:3]
['b', 'c']
```

5.9. Slicing

It uses two indices, start (inclusive) and end (exclusive). That is, `start`, `start + 1`, `start + 2`, ... `end - 2`, `end - 1`. Hence, in this example, `a[1:3]` picks two items, `a[1]` and `a[2]` (but, not `a[3]`), and it returns a list including these two items.

Likewise,

```
>>> a[0:5]
['a', 'b', 'c', 'd', 'e']
```

This returns the whole sequence since the specified range covers the entire sequence in this case. Note, however, that the returned list is not the same as the original list `a`. It returns a new list.

One thing to note is that the returned (non-empty) sequence from slicing includes the *same items* that are in the original sequence. This holds true for both mutable and immutable sequences. This can have some unexpected consequences, if you are not aware of this. Changing the value of one of the items in a list *can* affect the slices of this list, and reversely, changing the value of an item in any slice of this list *can* affect the original list as well as all other slices of the same list. What exactly happens in this situation depends on what kind of operations we use and whether the items, not only the sequence, are mutable or immutable types, etc.

We will not discuss this any further in this book. But, all the basics related to this concept have already been explained, including the differences between the mutable and immutable types.

Now, going back to the slicing, when you pick a range that includes one item, slicing still returns a sequence (which includes this one item). For example,

```
>>> a[2:3]
['c']
```

If the range is empty, or the first index is equal to, or effectively bigger than, the

second, then it returns an empty sequence.

```
>>> a[2:2], a[4:1]
([], [])
```

One thing to note is that, unlike indexing, any integer value index is accepted in slicing. Slicing does not raise `IndexError` unlike indexing. You can use arbitrarily large positive integers for either start or end index. For instance,

```
>>> a[2:100], a[10:20]
(['c', 'd', 'e'], [])
```

There is no more elements beyond the index `4`, in this example, and hence that part of slicing range simply returns an empty set of elements. For the case of negative number indexes, modulo operation is taken, and its positive index, in the interval from `0` to `len(seq) - 1`, is used instead. For example,

```
>>> a[-4:-3], a[-4:3], a[-1:2]
(['b'], ['b', 'c'], [])
```

Note that `-4 % 5`, `-3 % 5`, and `-1 % 5` are `1`, `2`, and `4`, respectively, and hence this expression list is equivalent to the following:

```
>>> a[1:2], a[1:3], a[4:2]
(['b'], ['b', 'c'], [])
```

Likewise, index `-100` is the same as `0`, and `-9` is `1`, etc., as far as slicing is concerned. (Note that the index modulo is taken for negative indices, but not for positive indices.)

One other difference between indexing and slicing is that the item of a mutable

5.10. Sorting

sequence, selected by indexing, can be used on the left hand side of assignment. In that case, the object on the right hand side can be any object, including sequences.

On the other hand, if a part of a mutable sequence, e.g., selected by the slicing syntax, is used on the left hand side of an assignment statement, then the expression on the right hand side must be a sequence object of the same type. For instance,

```
>>> x = [1, 2]
>>> x[0:1] = [10, 11, 12]
>>> x
[10, 11, 12, 2]
>>> x[0:2] = []
>>> x
[12, 2]
```

When the start slice index is the same as the beginning index of a sequence, e.g., `0`, we can omit the start index. Likewise, when the end slice index is equal to, or bigger than, `len(seq)`, e.g., `5` in this example, we can omit the end index. For example,

```
>>> a[0:2], a[:2], a[4:5], a[4:]
(['a', 'b'], ['a', 'b'], ['e'], ['e'])
```

If we omit both, it returns a sequence corresponding to the entire valid range.

```
>>> a[0:5]
['a', 'b', 'c', 'd', 'e']
>>> a[:]
['a', 'b', 'c', 'd', 'e']
```

5.10. Sorting

Finally, the items of sequence objects are "ordered", as we briefly mentioned. That is,

for example, "ohell" is different from "hello". ☺

You can create a sequence object, e.g., a list, a tuple, or a string, in a particular order, in various ways. Clearly, you can create a brand new list, for instance, using the list literal syntax, or using the `list` constructor function with particularly ordered items, etc. (Every builtin type has a constructor function. The name of the constructor function is the same as that of the type. And, that is not a coincidence. More on this later.)

We can also create a sequence in a particular order from an existing sequence which has a different ordering. For this, we use the builtin `sorted` function.

For example,

```
>>> t = (1, 7, 5, 21, 11)
>>> sorted(t)
[1, 5, 7, 11, 21]
>>> sorted(t, reverse=True)
[21, 11, 7, 5, 1]
```

```
>>> s = ['p', 'y', 't', 'h', 'o', 'n']
>>> sorted(s)
['h', 'n', 'o', 'p', 't', 'y']
>>> sorted(s, reverse=True)
['y', 't', 'p', 'o', 'n', 'h']
```

The `sorted` function takes a sequence, e.g., a list or a tuple, as its first argument and it returns a *new* sequence sorted in the natural "ascending order". It also takes an optional Boolean "keyword argument" `reverse`, and if `reverse` is `True` then the sorting is done in the natural "descending order".

When we say "natural", the numbers are naturally ordered, e.g., from small to big. Likewise, the alphabets are ordered, e.g., from `A` to `Z` and then `a` to `z`. The `sorted` function call can be customized using another optional keyword argument `key`, but

5.11. Help!!

we will not discuss its usage in this book.

As stated before in the context of slicing, even though the `sorted()` function returns a new list or a new tuple, etc., their items are the same (e.g., the names pointing to the same objects in memory). They are just re-ordered. Hence, depending on whether their items are mutable or immutable, etc., manipulating or updating the sorted sequence, and their items, can affect the original sequence.

For a list, which is mutable, there is another builtin method that can sort a given list "in place". It has essentially the same signature as `sorted`, but it is a method defined on `list`, not a global function.

Here's an example:

```
>>> r = [1, 7, 5, 21, 11]
>>> r.sort()
>>> r
[1, 5, 7, 11, 21]
>>> r.sort(reverse=True)
>>> r
[21, 11, 7, 5, 1]
```

You can also use the same optional keyword argument `key` to customize the sorting behavior. Note that this method is not available for tuples, which are immutable.

5.11. Help!!

As stated, the Python REPL is a very important tool for Python programmers. Before we end this tour, let's review a couple of basic commands/functions that are useful in Python REPL.

As stated, all data in Python are *objects*. An object can have one, zero, or more associated *attributes*, e.g., data attributes or methods. For the objects of the `list` type, for instance, we have used the methods like `append`, `pop`, and `sort` in this

lesson.

Python's builtin `dir` function lists all attributes of a given object, including all available methods. As we will see later in the book, the set of the (initial) attributes of an object are based on the attributes of its type (which is also an object ☺). Hence all *list objects* include the attributes of the *list type*.

For instance,

```
>>> dir(list)
['__add__', '__class__', '__class_getitem__', '__contains__',
 '__delattr__', '__delitem__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__getitem__', '__gt__',
 '__hash__', '__iadd__', '__imul__', '__init__', '__init_subclass__',
 '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
 '__setattr__', '__setitem__', '__sizeof__', '__str__', '__subclasshook__',
 'append', 'clear', 'copy', 'count', 'extend', 'index', 'insert', 'pop',
 'remove', 'reverse', 'sort']
```

There is quite a bit. But, we recognize the aforementioned three methods, `append`, `pop`, and `sort`, which we used with some list objects earlier. Note that the value of the `dir` function call is a list (i.e., a list literal), as indicated by the pair of starting and ending square brackets, and its items are all of the string type.

You can notice that the names of many methods, and other attributes, start and end with double underscores (`__`), which are often called the "dunder attributes" (or, the "dunder methods", etc.). These are special attributes, as we will discuss later in the book.

If you need more information on a particular method, you can again use the `help` function. For instance,

```
>>> help(list.append)
```

5.12. Inspirations

Help on method_descriptor:

```
append(self, object, /)
    Append object to the end of the list.
(END)
```

①

① You may have to press "q" at this point to go back to the REPL prompt.

You can also use `help()` on a type, e.g., `help(float)` or `help(list)`, etc. Give it a try.

5.12. Inspirations

There is an "Easter egg" hidden in Python. ☺ Try "import this" in the Python REPL:

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
```

Namespaces are one honking great idea -- let's do more of those!



5.13. Summary

We looked at a couple of complex or compound data types built into Python, namely, `tuple` and `list`.

A tuple literal is denoted with parentheses, `()`. The elements in a tuple are separated by commas. An expression list in Python (one or more expressions likewise separated by commas) is evaluated to a tuple. A list literal is denoted with a pair of square brackets, `[]`. The elements in a list are separated by commas as well.

A tuple object is immutable (that is, you cannot directly change its value and its elements, including its length, etc.) whereas a list object is mutable (although "immutable" does not mean "truly immutable" in Python).

Indexing can be used to refer to an item in a tuple or a list, e.g., using the subscript operator `[]`. A valid index is an integer from `0` to the length of the sequence minus `1`. For an empty sequence, even `0` is not a valid index. Indexing an item with an invalid index raises an `IndexError` exception.

The value of an object of the `list` type can be altered in various ways, e.g., using an item assignment, or using methods like `append()` and `pop()`. These mutating operations are not defined for the `tuple` type.

In Python, we can also create a new sequence from another sequence using "slicing". Slicing uses the similar index notation as indexing, but with a range, e.g., `start:end`. Slicing does not raise an `IndexError` exception.

The `sorted` function is another way to create a new sequence from another sequence, e.g., with the items sorted in a particular order. The mutable list objects also support the `sort` method, which re-orders the items in the given list without

5.13. Summary

creating a new list.

Although we did not discuss in this lesson, "dictionary" (or, `dict`) is another complex (mutable) type in Python, which represents a key-value pair collection data structure. The dictionary type (or, the `dict` type) uses curly braces `{}` for its literals. We will use dictionaries later in the book.



Python also supports a syntax called the "comprehension", e.g., the list comprehension, the tuple comprehension, and the dictionary comprehension, to create a new collection from an existing sequence object like a list or "range". We do not discuss this in this book, but it is one of the most important (functional programming) concepts in Python. The readers are encouraged to look this up (when you feel like you are ready 😊).

We also introduced in this lesson the important concept of variables in Python. A variable is just a name for, or a *reference* to, an object. We often need variables, or names, to manipulate objects in Python programs. Objects, especially mutable ones, that do not have names have limited uses in Python.

This concludes the introductory tour. We will work on a few "real programs" for the rest of this book.

Chapter 6. Review - Basics

Deeds will not be less valiant because they are unpraised.

— Aragon (The Lord of the Rings)

We briefly looked at the basics of programming in Python, primarily using the Python interpreter in the interactive mode. The Python interactive shell, or REPL, can be rather useful for quickly testing ideas or verifying Python syntax, etc.

In the interactive mode, the Python interpreter has some unique characteristics that are slightly different from when it is used in the non-interactive mode, that is, when it is used to run a "script" (e.g., from a file).

First, an input expression is evaluated and its value is printed out in the interactive mode (unless the value is **None**). This comes in handy when you want to quickly see how an expression evaluates. You can also evaluate multiple expressions "at the same time" (as in *from left to right* 😊) using the expression list syntax.

Second, you cannot run more than one statements at the same time, or together, in the interactive mode. Each statement, including a (multiline) compound statement, is executed upon entering, e.g., after an empty line input (two Enters). Simple statements can be executed at the same time (again, *from left to right* 😊), that is, on a single physical line, separated by semicolons.

Another convenient feature, when the Python interpreter is used in the interactive mode, is the use of the predefined variable `_`. Whenever an expression is evaluated, its value, unless it is **None**, is stored in the variable `_`. Hence, we need not always assign (temporary) values to variables.

Clearly, the major downside of using the Python interpreter in the interactive mode is that once you close the interpreter, all your code is gone. We primarily use "scripts" (e.g., program files) to develop "real programs".

6.1. Questions

1. Have you installed Python on your machine? It's a yes-no question. No judgement. ☺
2. How do you start the Python interpreter in the interactive mode on your machine?
3. What version of Python are you using? How would you find out what version you are running?
4. Do you have a good text editor that you can use to create and edit Python programs (program files)?
5. If you haven't installed Python on your computer, then there are a few other options like using online tools. Do a Web search, and decide on your online editor/Python interpreter that you are going to use while reading this book. ([replit](https://replit.com/) [https://replit.com/] is one of the popular services. BTW, as of this writing, VS Code also released its online service: [VS Code](https://vscode.dev/) [https://vscode.dev/].)
6. Are you going to keep reading this book? (The *correct answer* is yes! ☺)

6.2. Exercises

The following exercises can be done in the Python REPL.

1. Create an empty tuple.
2. Create a tuple with one item, "Holy Grail".
3. Create a tuple with three elements, 1, 3, and 5.
4. Add three numbers using the addition operator (+), 2, 4, 24. What is the value?
5. What is the value of this expression, 21 // 2 / 5? What is its type?
6. What is the value of this expression, 2 + 3 * 3 - 7? What is its type?
7. Create an empty list. Append an item 'a'. Append an item 'b'. And, append an item 'c'. What is the current value of the list object?

8. Create a list with three elements, `10`, `20`, `30`. Remove the last element. Then, remove the last element again from this list. What is the current value of the list?
9. What is `dict` (dictionary) in Python? How do you use the `help` function to get the info on `dict`?
10. How do you get all the attributes of the `dict` type?
11. Define a function that takes three `int` parameters and returns their sum. Try calling this function with `101`, `102`, and `103`, and print out the result.
12. Try calling this three integer sum function with `True`, `False`, and `True`. What is the result?
13. Define a function that takes three `float` parameters and prints out their product to the terminal. Call this function with `2.0`, `2.0`, and `5.25` and verify the output.
14. Define a function with one `int` parameter that returns `False` if the given argument is less than, or equal to, zero, and `True` otherwise. How would you name this function?
15. Define a function that takes no arguments, does nothing, and returns none. ☺
16. Create a list with four items, `200`, `100`, `600`, and `400`. Create a new list which is sorted in the ascending order. Sort this new list in the reverse order in place. (Python also has a builtin list method `reverse`, which will give the same result, in this particular example.)
17. Create a tuple of four list items, `[2,1]`, `[4,3]`, `[6,5]`, and `[8,7]`. Take a three item slice of this tuple, from index 0 to index 2, both inclusive. Sort each of the three list items in this new (slice) tuple, in the ascending order. Print out the updated slice. Print out the original (4 item) tuple, and see what happens.
18. Using the original tuple in the previous exercise, add a new number, `4`, to the end of the third list item (index 2). Remove the last item `7` from the last list item in the original tuple. Print out the 3-item list tuple (from the previous exercise) and the original 4 item tuple and see what happens. A tuple is an immutable type, but we can easily see from these (simple) examples that tuple objects may not be truly immutable.

Rock Paper Scissors Project

It's the job that's never started as takes longest to finish.

— Sam (The Lord of the Rings)

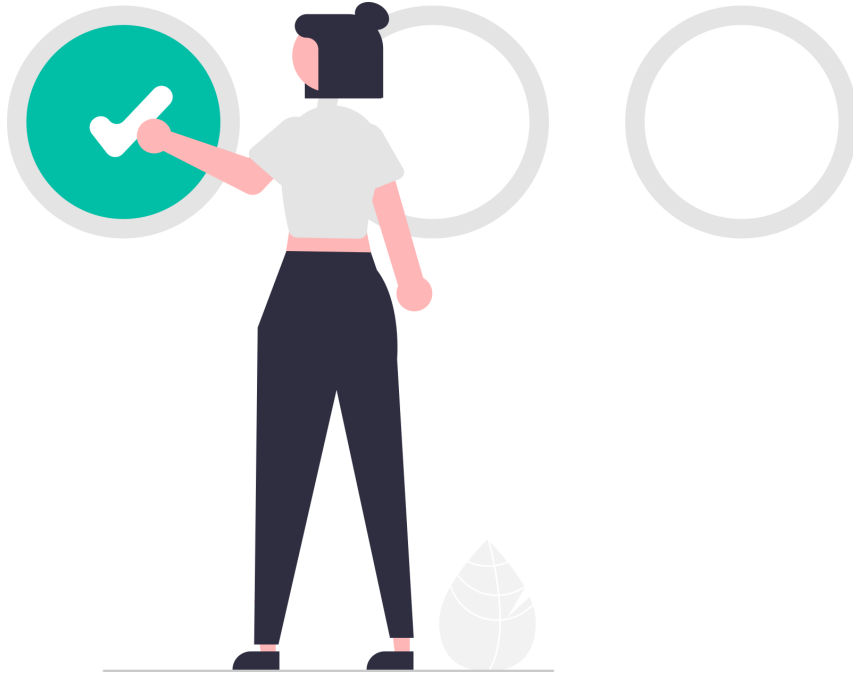
Congratulations!

You are now going to learn "real software development" in Python, regardless of your prior programming experience (or, lack thereof). We will build a command line version(s) of the rock paper scissors game for the rest of the book. As stated, the choice of this particular project is not that significant. We will learn the essentials of Python programming while working on this project. A simple, but real, project. We deliberately picked one of the simplest problems so that we can focus on learning programming.

Many beginner's programming books only teach how the `for` statement works, how to use the `if` statement, and so forth. After reading those books, most beginners end up knowing very little about real programming. They cannot even write a simple program in Python. This is because programming is not a sum of the features of the programming language. You can learn programming only through real, actual, programming, not by memorizing the language syntax.

This book does not promise that you will become a master programmer after learning programming from this book. But, you will gain much more "real programming experience" from this book than from any other resources, in terms of your investment, hour for hour. If you still prefer to get only the superficial knowledge on Python, then you can go back to those resources. ☹️ But, if you are interested in learning real programming, then keep reading. The best is yet to come. 😊

Chapter 7. Hello Rock Paper Scissors!



7.1. Working on a Project

Any programming project has (at least) two components, the problem that we are trying to solve, and the method that we are going to use to solve the problem using the computer (although they are intricately connected).

In books like this, we tend to emphasize the computer programming part. In the real-world projects, however, understanding the problem (in depth) can be more important. The "quality" of your solution generally depends on your "domain knowledge", and other general problem solving skills. As stated, the computer is just a tool, and you will have to solve the problem.

For our Rock Paper Scissors project, the problem is almost trivial. Most of us are familiar with the rock paper scissors game (which may be called by different names

7.2. Let's Play Rock Paper Scissors

in different parts of the world). Nonetheless, let's spend some time "analyzing" the problem, before diving into programming.

If you have never played rock paper scissors before (really? ☺), then it will be useful to look up some resources on the Web. Here's the link to a Wikipedia page: [Rock paper scissors](https://en.wikipedia.org/wiki/Rock_paper_scissors) [https://en.wikipedia.org/wiki/Rock_paper_scissors].

7.2. Let's Play Rock Paper Scissors

We are not trying to put the cart before the horse, but let's use the app that we *are going to build* for practice for now. There are also online games available that you can play on your Web browser. You can always play the game, "for real", with a friend.

We will use the "second version" here. (We will end up creating three different versions of Rock Paper Scissors in this book. ☺)

When we start the game (e.g., by invoking the Python interpreter),

```
$ python main.py
+++++
Let's play Rock Paper Scissors!
+++++
Rock (r), Paper (p), or Scissors (s)?
```

It prints out the "banner", and it starts the game. The game then prompts the user to play their hand, and it waits for a user input. There is no time limit for the user input.

```
$ python main.py
+++++
Let's play Rock Paper Scissors!
+++++
Rock (r), Paper (p), or Scissors (s)? r ①
```

```

You -- Rock vs Scissors -- Computer
You win!
+---+---+---+---+---+---+---+---+---+---+
Rock (r), Paper (p), or Scissors (s)?

```

- ① **r** is the player input. The rest is printed out by the program. (Or, more precisely by the Python interpreter, which executes the program/script. But, this distinction is not very important for our purposes.)

When the user plays his/her hand, *Rock* in this case, the computer selects a random hand, which happens to be *Scissors* in this example, and decides who wins.



If you "cheat", as a programmer, then your program can always win, 100 percent of the time. You just read the player's hand first and generate the computer hand based on the player's hand so that it is a winning hand. Or, you can make it win just a little bit more than 50% so that the player would not suspect. 😊

This is a general problem when you, as a user, deal with computers. How do you know what the computer is doing is "fair"? We will briefly address this issue at the end of the book.

In this example, the user has won, and the computer then moves on to the next "round". It asks the user for another hand. In this particular example, once three "rounds" are played, the game ends.

```

$ python main.py
+---+---+---+---+---+---+---+---+---+---+
Let's play Rock Paper Scissors!
+---+---+---+---+---+---+---+---+---+---+
Rock (r), Paper (p), or Scissors (s)? r
You -- Rock vs Scissors -- Computer
You win!
+---+---+---+---+---+---+---+---+---+---+
Rock (r), Paper (p), or Scissors (s)? p

```

7.2. Let's Play Rock Paper Scissors

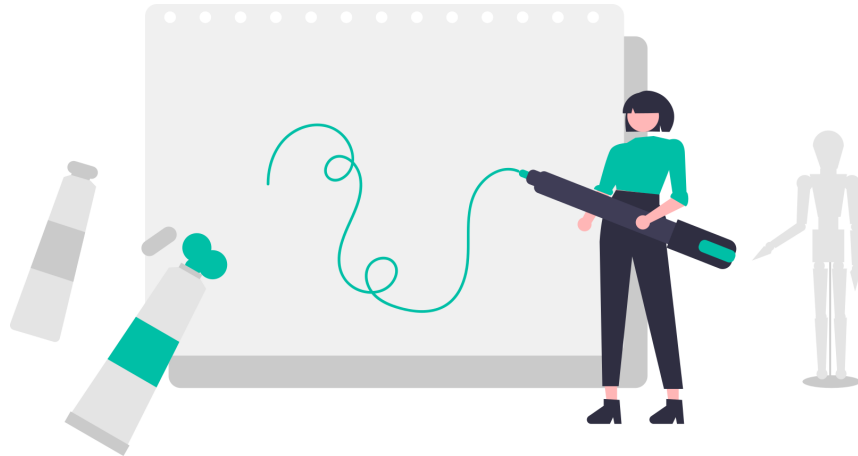
```
You -- Paper vs Scissors -- Computer
You lose!
+---+---+---+---+---+---+---+---+---+
Rock (r), Paper (p), or Scissors (s)? s
You -- Scissors vs Scissors -- Computer
Tie!
+---+---+---+---+---+---+---+---+---+
Thanks for playing Rock Paper Scissors!
+---+---+---+---+---+---+---+---+---+
$
```

Note that this sample output shows three rounds of play, and they are more or less repetitions. Each round includes only three lines of output, each of which seems to mean something. 😊

Do you think you can write a program that does something like this in Python? We will do it together later in the book. In fact, we will create three (slightly) different implementations of the game.

OK, let's do it!

Chapter 8. Software Design



Writing a program is not unlike writing a fictional, or non-fictional, story. When you write a novel, for instance, you will first have to come up with a "plot", a storyline, before you can even write the first sentence. The same with a nonfiction like a newspaper article.



Legend has it that J.K. Rowling came up with the (whole) storyline for the Harry Potter series while she was waiting on a train. Apparently, the train was delayed for 4 hours, and that was when she came up with this wizard idea. She also worked out all the stories in that 4 hour time according to the legend. Well, regardless of whether that is (entirely) true or not 😊, it illustrates the importance of having a plot *before* you actually start writing.

Different kinds of software need different kinds of "preparations". The details can be different. But, the goal is the same. In essence, we are trying to solve a given problem via the "divide and conquer" method. Or, more precisely,

- Divide a big problem into smaller problems,
- Solve the small problems, and

8.1. Deconstructing Rock Paper Scissors

- Assemble the small solutions into the solution of the original big problem.

As you can imagine, an arbitrary or random division might not be very useful. We will have to carefully think over how to divide the problem (and, how to assemble back the complete solution). The process of "division" may not be limited to breaking a big thing into smaller pieces. We may have to divide a large task into a series of smaller consecutive tasks, for instance. In the object orient programming style, the "division" might be more abstract. Etc.

One thing to note is that this is a recursive process. A "small" part of a big problem, or a "small" piece of a big task, might not be small enough and it may need to be further divided in some way into even smaller parts and tasks, and so on.

8.1. Deconstructing Rock Paper Scissors

Let's understand the problem at hand. Our goal is to implement a program that plays the rock paper scissors game with a user. What do we need to do? What does our program need to do?

Here's one possible breakdown:

- First, (optionally) we will need a way to indicate the start of a new game.
- Once a game starts, we will need to be able to read the user's "hand".
- Then, we will need to pick a (random) hand, for the computer.
- Next, we should compare the two hands, and decide which side wins.
- Finally, we print out the result, win, lose, or tie, to the player, and we end the game after (optionally) printing out the game-end message.
- (Optionally) we can let the user play more than one "round" in a game.

Here's a "flowchart":

```
A game start
Display the game start banner
```

```

Loop:
  Ask the player for a hand
  Read the player's hand
  Generate a random hand (for the computer)
  Compare two hands and determine who wins
  Display the result
Game end

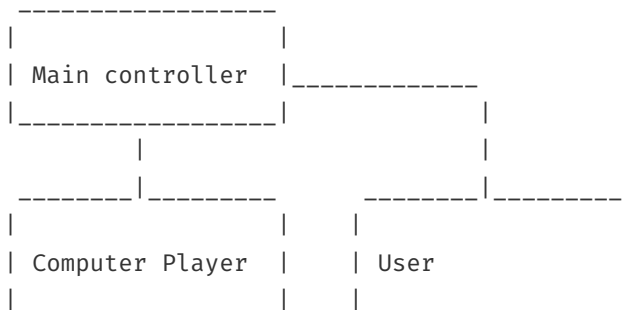
```

As stated, our "big" problem of "creating a rock paper scissors computer game" has been divided into multiple small steps, or partial tasks.



Does this make sense to you? More or less? Are you convinced that we will need to implement something like this to create our Rock Paper Scissors app? Remember, the computer does only what you tell it to do. If your program does not work, then it is not the computer's fault. 😊

Let's look at this from another angle. The program really consists of two components: A "computer player", who should be able to choose a hand (rock, paper, or scissors), and the "main controller", who manages the overall game flow, including comparing the hands and determining the winner, etc. In fact, we can separate out another small component from the main part, which plays the role of a player. This component will read the player's input (a "hand") and communicate it to the main controller.



8.2. Tasks

Again, our "big" problem has been divided into a number of "small" parts. There are other ways as well. These different ways of division, different ways of looking at the same problem, are not necessarily mutually exclusive. They can be, and often are, complementary to one another.

We will try a few different methods in this book while implementing (essentially) the same rock paper scissors apps.

8.2. Tasks

Breaking down a project into a series of smaller "tasks" is another way of "divide and conquer". These tasks really depend on how exactly we are going to build the software, including the choice of high-level designs and what not.

Here's a simple example, assuming that we are going to implement the program sort of "sequentially" following the flow chart example above.

- Create an empty/boilerplate program in a project folder.
- Work on the high level program structure.
- Display the game start message.
- Start a main loop (so that the user can play multiple rounds).
- Read a user's input.
- Convert it into a "hand" (e.g., Rock, Paper, or Scissors).
- Pick a computer hand (e.g., in a random manner).
- Determine whose hand wins.
- Print out the result.
- Once the game ends, display the game end message.

This is just an example, but we will end up doing more or less the same tasks for each of our three different implementations.

Chapter 9. Project Setup



There are many different ways to set up and manage a software project. We will introduce a certain (simple but rather formal) way in this book. For a small project like this, this kind of setup is not really required. But, for learning purposes, we will show one particular example. The readers can make any necessary changes to this baseline guideline. In fact, some, or even all, of these "setup steps" can be omitted.



This lesson provides some common things that the (professional) developers do when they start working on a new project. If you are only interested in learning Python syntax, or if you have done some Python development before, then you can skip this lesson.

9.1. Workspace

We use a computer for a variety of purposes. This is true even for professional software developers. Not many people have dedicated computers for software development only. So, it is rather important to "organize" your software projects on your machine.

Some people have a single top-level folder in their "home" directory and use each of

9.1. Workspace

its subfolders for one software project. There are many other options. The choice is yours. In this book, we will use a directory named *projects* under the home directory.

In Unix-like systems, you can use the *cd* command (without an argument).

```
$ cd ①
```

① *cd* stands for "change directory".

This will change the current directory to your home. You can verify this using the *pwd* command:

```
$ pwd ①  
/home/harry ②
```

① *pwd* stands for "print working directory".

② The username of the author's account on this computer is *harry* and its home folder is */home/harry*.

Then we can create a new folder named *projects* as follows:

```
$ mkdir projects ①  
$ cd projects
```

① *mkdir* is a command that "makes a directory".

As we have seen before, if you use a shell that is based on Bourne Shell, such as BASH, then you can do `mkdir projects && cd $_` or `mkdir projects; cd $_`, etc. At this point,

```
$ pwd  
/home/harry/projects
```

The *CWD*, current working directory, is *~/projects*. (The tilde symbol *~* represents the home folder of the current user. *cd*, without any arguments, is equivalent to *cd ~*. On Windows CMD (the Command program), you can use *cd \$HOME*, *md projects*, *cd projects*, and *cd* for *mkdir projects*, *cd projects*, and *pwd*, respectively.)

Next, let's create the project folder for our rock paper scissors program(s).

```
$ mkdir rps
$ cd rps
$ pwd
/home/harry/projects/rps
```

You can name the project folder any way you'd like. We have chosen the name *rps* because that seems like a pretty reasonable name for the rock paper scissors game. ☺ As we will see later, the choice of folder names for Python programs can be significant in some situations, but that is generally not the case for the "top-level" project folder (that is, if you follow the best practice).

9.2. Virtual Environments

Python is normally installed globally on a system (even on a multi-user system), and the third party libraries (or, "packages") are likewise stored at a common location(s) as well. As one can easily imagine, this *can* be a problem. Many libraries are updated over time and they have different versions and different, often mutually conflicting, dependencies.

If you work on multiple Python projects on the same computer (as most of us do), then some projects may depend on the same libraries but with different versions. Some programming languages have the concept of formal "projects", which lets you independently manage the library dependencies, among other things, "per project". Python does not have such a thing. (Python uses "projects" for packaging/publishing purposes, but that is slightly different.)

Instead, Python uses what is called the "virtual environments". A virtual

9.2. Virtual Environments

environment is an *isolated* (abstract) working space as far as Python is concerned. One can run, in a given virtual environment, a different version of Python interpreter and install different versions of the same libraries independently of what is available across the system and in other virtual environments.

You can use a few different virtual environments on your system and select an appropriate virtual environment for a given programming project. Or, alternatively, you can create and use one virtual environment per program/project. In this book, we will use the latter approach. As stated, for small projects like our rock paper scissors program (with few or no external library dependencies), it is not strictly necessary to use virtual environments. But, it is still a good practice, and we recommend you use it for all your (future) Python projects.

Historically, there have been a number of different incarnations of "virtual environments". Currently, however, many people use the standard library `venv`. We will use `venv` in this book. (Just keep in mind that there are other alternatives.)



If you use Python distributions like Anaconda, then they might have their own way of setting up virtual environments. We will not discuss Anaconda in this book, but it is mainly used for data science and machine learning projects,

In the project folder, *rps*, type this:

```
$ python -m venv venv
```

If you happen to have more than one versions of Python installed on your computer, then use the appropriate command (e.g., instead of `python`) to use a particular version of your choice. As indicated, the most recent version (3.10 as of this writing) is recommended.

The author uses the `python3` command. For instance,

```
$ python3 -m venv venv
```

As we have seen before, the `-m` flag executes the specified module, `venv` in this case. The last argument `venv` (or, `./venv`) is the folder that will be used by the `venv` module to store any virtual environment specific data. The folder name is arbitrary, but names like `venv` or `virtualenv`, or something similar, are commonly used. As stated, you can create a `venv` folder in a shared location as well (e.g., to be shared by multiple Python programs/projects).

Make sure that the command runs successfully, and that you have the `venv` subfolder in the current working directory (if you are following exactly what we are doing here). Then do the following, if you use a Bourne shell (like BASH or ZSH):

```
$ source venv/bin/activate
```

Or, if you use a flavor of C shell,

```
$ source venv/bin/activate.csh
```

The `source` Unix command executes the shell script `activate` or `activate.csh` in the current shell. The subfolder name `venv` in the relative path `({.}/)venv/bin/activate(.csh)` is the name of the folder that we just used while creating this virtual environment. After executing this command,

```
(venv) $
```

You can notice that the shell prompt has changed. It is now preceded by `(venv)`. As stated, the `$` symbol is used for the normal shell prompt in this book, and it may be different on your system (depending on your particular shell and its configurations). On the author's computer (using the BASH default settings), it looks like this:

9.3. Package Install

```
(venv) harry@dory:~/projects/rps$
```

Note that, now everybody uses the same command `python` regardless of what exact command they used to create a venv from the shell. For instance, in the author's computer,

```
(venv) $ python --version ①
Python 3.10.4
(venv) $
```

① Note that *python* is used, not *python3*.

You can exit the current virtual environment as follows:

```
(venv) $ deactivate
$
```

We will just use the shell prompt `$` for the rest of the book, for simplicity, even when we are in a particular Python virtual environment.



As stated, if you are on Windows and use CMD or PowerShell (e.g., instead of using a Linux terminal on WSL), then you may have to do some Web search to figure out what the corresponding commands are, in this step, as well as throughout this book.

9.3. Package Install

Python (third party) libraries are normally distributed as "PyPi packages". You will need a tool/module "pip" to install and manage these packages, either globally or in a particular virtual environment. Many Python distributions also come with the *pip* command line tool. In some platforms, you may need to separately install a CLI

version of *pip*. (*pip* is a Python standard library module, and it is always available with the standard Python interpreter.)

Now, if you have decided to use *venv* for your rock paper scissors project, then activate it.

```
$ . venv/bin/activate ①  
(venv) $
```

① In BASH, the `.` command is a synonym to the *source* command.

As stated, we will just use `$` for all different prompts, including `(venv) $`, from now on, for the sake of simplicity. Then, do the following:

```
$ python -m pip install -U pip autopep8 ①
```

① First, note that we are in the currently active virtual environment. Second, you can use the *pip* CLI command if you have it on your system. Otherwise, you can always do `python -m pip`.

The above command likely updates the *pip* module (e.g., in the currently active virtual environment) and installs the *autopep8* module (e.g., because it does not have it in the newly created virtual environment). We can also do it in two separate commands:

```
$ python -m pip install -U pip  
$ python -m pip install autopep8
```

The *autopep8* module is used for auto styling and formatting during development. We will discuss this further later in the book.

Note that these are more or less general dependencies (e.g., for all projects), and our rock paper scissors game (all three variations) does not have any third party library

dependencies specific to the game.

In general, however, you will end up using some external libraries, modules or package modules, and you will most likely use *pip*. The general syntax is the same, you specify the package name after the `-m pip install` flag, possibly with `-U` option.

```
$ python -m pip install -U SomePackage
```

9.4. Source Control System

Professional software developers use "source control systems" (aka "version control systems") to manage their source code. Programs are developed over time, and it is important to keep track of the changes made throughout the project.

Using a source control system is particularly important in a team environment where multiple developers concurrently work on the same software project. But, it has many other uses. If nothing else, it can be used as a file backup system (on steroid). If you lose your development computer, or if it crashes and becomes unrecoverable, etc., you can still retrieve your program source files (which is *the most valuable thing* to the developers ☺) depending on where your source code files are stored.

For beginners, and hobby programmers, it is not essential to use a version control system. For the rock paper scissors project in this book, for example, it is optional. However, if you are *serious* about doing any *serious* programming, now or in the future ☺, then we *highly* recommend that you start using a version control system. (At least, at some point in the future, not necessarily now.)

Traditionally, version control systems were based on the "client-server model". That is, you use your computer ("a client") to do programming, but you use the version control system on a remote "server" machine (e.g., owned/operated by your employer), which is shared by all developers working on the same project (or, across

the organization, etc.).

These days, however, the "peer to peer" version control systems are more popular. Two of the most widely used such systems are `git` and `mercurial`. We will use `git` in this book. A peer to peer source control system runs on each developer's computer, and, in theory, developers can "sync" their source code (of the same project) in various ways.

In practice, even for these peer-to-peer systems, it is most common to use a client-server model (although this terminology is not used for these systems). That is, we use one computer as a "server" and the rest as "clients" by convention. In fact, there are a number of companies that provide a "hosting service" or "cloud service" for `git` so that developers can use their service as "servers". GitHub is the most popular one. There are also other services like GitLab and BitBucket, etc.

For the project of this book, we will ignore this *client-server paradigm*. We will just use `git` on our computer and ignore the "remote server" part. As indicated, this provides all the benefits of using a source control system except that you cannot easily share your code on multiple computers (that is, if you use multiple laptops for development, etc.). Another thing to note is that if you lose your computer, you lose *everything* ☹ unless you have a separate backup.

You will first need to install the `git` program on your development computer(s). You might already have it on your computer. Try `git --version`, for instance. Otherwise, go to the official git website, [git download](https://git-scm.com/downloads) [https://git-scm.com/downloads], to download and install git. There are other distributions that can be installed with the platform specific package managers (e.g., dpkg/apt, yum/dnf, homebrew, etc.). The official git website always has the most recent version, but in many cases the "standard" distributions on your platform should be sufficient (although they might be a slightly older version).

Once `git` is installed, do the following from a terminal. This is a one-time setup.

```
git config --global user.name "your name"
```

9.4. Source Control System

```
git config --global user.email "your@email.address"
```

Use any name (like your real name ☺) and email address (which you don't mind using publicly). Whenever you "commit" the source code changes to git, it will record them with your name and email. (If you end up using a public git server and pushing your git repository later, then this data will become all public.)

Now `cd` to the `rps` directory that we just created, if you are not already there. Then create a file named `.gitignore`. For example, on a Unix-like platform,

```
$ touch .gitignore
```

The `touch` Unix command updates the timestamp of a file, if it exists. Otherwise, it creates a new file with the given name. Note the leading dot `.` in the name. The "dot files" do not show up through `ls` by default. You will need to do `ls -a` (typically `ls -la` for the "long" listing) to also list the dot files.

Then copy the content of the page, [gitignore/Python.gitignore](https://github.com/github/gitignore/blob/master/Python.gitignore) [https://github.com/github/gitignore/blob/master/Python.gitignore], to this file. The `.gitignore` files are used to exclude certain files from the version control system (like temporary files, or the build outputs, which need not, or should not, be stored in the git). You can have multiple (different) `.gitignore` files in one project, e.g., in different folders. For our purposes, and for small projects, the top level `.gitignore` file often suffices. Note that there is a line `venv/`, for instance, in this "standard/boilerplate" `.gitignore` file for Python. We are not going to check in the `venv` folder and its content/subfolders to the git repository.



You can use any text editor for this. But, if you have installed VS Code, then you can use it as an editor for all files, not just for the Python program files. When you use VSCode, it is more common to open an entire folder rather than just open a single file or a few files. As explained before, a folder (including its subfolders, etc.) is the simplest, and the most convenient, type of the "workspaces".

Make sure that you have the correct file edited in the correct folder:

```
$ ls -la
```

You can also check the content of the file:

```
$ cat .gitignore
```

Then, do the following:

```
$ git init
$ git add .
$ git commit -m "Initial commit"
```

Version control systems like `git` use what is called the "branches" to store multiple different "incarnations" of the same program. By default, git uses the name *master* for the first, or the "main", branch. We will only use the main *master* branch in this book.

This example uses three git commands, `init`, `add`, and `commit`. We will mention other git commands, as needed, throughout this book.

- The `git init` command initializes the current directory as a new "repository" (e.g., sort of like a database for storing file revision information, etc.). After running this command, you will notice that a new "hidden" directory is created, namely, `.git`.
- The `git add <files and folders>` command *stages* the recent changes (e.g., since the last commit). In git, checking files/changes to the repository is a two-step process. You will need to select the files/changes and stage them first using `git add` and then you can commit the staged files/changes. In this example, we do `git add .`, as in "stage all changes in the current directory, including all its subdirectories, if any". (Incidentally, the only change in this particular example is

9.4. Source Control System

the newly added *.gitignore* file.)

- Then, we commit the staged files/changes through the `git commit` command. The `-m` flag is used to specify the "git commit message", and it is sort of required. If you do not use the `-m` flag, git will automatically open a default editor app so that you can write a commit message. It is generally more convenient to provide the commit message in the command line than having to go through an extra editing unless you have a specially long commit message.

Imperative Programming

Even the smallest person can change the course of the future.

— Galadriel (The Lord of the Rings)

We went through some "extensive preparations" in the previous part for some real software development. ☺ We set up a new git repository, a development workspace, and a Python virtual environment, for our rock paper scissors project.

Although there is really no "standard process" for software development, and you do not have to follow any particular processes, it is always a good idea to add some order to otherwise very complex tasks. Programming is very easy to start, but it quickly becomes unmanageable once we start working on more complex problems.

Now that we have *everything* ready, all we need now is some real programs. ☺

In this part, we will start implementing the first version of the rock paper scissors program. "Imperative programming" is one of the oldest and the most widely used programming (since the early success of Fortran in the 1940s). It simply means that you achieve your goal using a series of statements. As stated earlier, all programming (in programming languages like Python) essentially boils down to imperative programming at its foundation regardless of whether you use other high-level programming paradigms like OOP.

If you are new to programming, then it is rather important to get the basics right. This part is divided into two lessons, a theory lesson and a "lab" for practice.

Now let's go!

Chapter 10. Main Project - Rock Paper Scissors



We mostly used the Python interpreter in the interactive mode throughout the beginning of this book. Now, let's take a look at a small "real" Python program. That is, a program written in a file.

As we will see, the development process is slightly different. Most of the time, you will most likely write code in files so that, among other things, they can be revised, and used more than once, if necessary. Python programs, or program files, are often called the "scripts".



Despite the common use of the word "program", Python does not create an "executable program". It is the Python Interpreter program that executes the Python scripts. As alluded before, however, this distinction is not important to us. We will continue to

use the term "programs" to refer to the Python scripts.

10.1. Rock Paper Scissors

The first rock paper scissors program that we will study in this lesson accepts an input from the user, "rock", "paper", or "scissors", and it prints out the result as to whether the user has won or not. For now, we will only play one hand, or one "round".

In Python, depending on how exactly we define what a "program" is, one file roughly corresponds to one program, an executable (aka "script") or a library (aka "module"). Or, both. In practice, a source code file may use the functionalities provided by other source code files, local and/or third-party, and a "program" may involve multiple Python source files. Large Python programs can use hundreds, if not thousands, of source code files, either directly or indirectly.

For now, let's start with the "one file - one program" paradigm. Even in a more general case, a program always starts with one file. Here's our first program for Rock Paper Scissors:

rps/main.py

```
1 import random
2
3
4 def to_string(hand: str) -> str:
5     if hand == "r":
6         return "Rock"
7     elif hand == "p":
8         return "Paper"
9     elif hand == "s":
10        return "Scissors"
11    else:
12        return ""
13
14
```

10.1. Rock Paper Scissors

```
15 # Read a user input and convert it into r, p, or s.
16 user_hand: str
17 u: str = input("Rock (r), Paper (p), or Scissors (s)? ").lower()
18 if u.startswith("r"):
19     user_hand = "r"
20 elif u.startswith("p"):
21     user_hand = "p"
22 elif u.startswith("s"):
23     user_hand = "s"
24 else:
25     user_hand = ""
26
27 # Randomly pick one of the strings, r, p, and s.
28 computer_hand: str
29 r: int = random.randint(0, 2)
30 if r == 0:
31     computer_hand = "r"
32 elif r == 1:
33     computer_hand = "p"
34 else:
35     computer_hand = "s"
36
37 print("You: " + to_string(user_hand) +
38       " -- Computer: " + to_string(computer_hand))
39
40
41 # Compare the two hands and print the result.
42 if user_hand == computer_hand:
43     print("Tie")
44 elif (user_hand == "r" and computer_hand == "s" or
45       user_hand == "p" and computer_hand == "r" or
46       user_hand == "s" and computer_hand == "p"):
47     print("You win")
48 else:
49     print("You lose")
```



The label indicates that this particular Python program file is

named `main.py`, and it is stored in a folder named `rps`, in a certain unspecified parent folder on the author's computer. (As we did together earlier, the precise location is `~/projects/rps` in this case. The absolute file path, on a particular computer, is not generally relevant when you are developing a Python program.)



You do not have to copy this code to get the "hands-on" experience. You will get to do this on your own in the next "lab session". 😊

Let's try running this program:

```
$ python main.py
Rock (r), Paper (p), or Scissors (s)? rock
You: Rock -- Computer: Scissors
You win
```

- ① The program printed out "Rock (r), Paper (p), or Scissors (s)? ", and the user typed in "rock" (and pressed Enter) in this example.

The program echoes back the player's hand along with its own hand ("scissors" in this case), "You: Rock — Computer: Scissors". It terminates after printing out the result "You win" verifying that "rock" beats "scissors" in the rock paper scissors game.

Note that we run a Python script by providing the file name after the python command, as a "command line argument", as in `python main.py`. (This is *one* of the ways to run Python code, if you'll remember. 😊)

So, how does this program work?

The first line is an "import statement", which imports "names" from other "modules" or "packages", e.g., the standard library `random` module in this case, so that those names (and their functionalities) can be used in our program. In this particular case, `import random`, we are only importing the module name, `random`, and other names

in this module will need to be "qualified" with the module name, e.g., using the prefix `random` as we do on line 29.



We have used the term "package" before, in a slightly different context. The third party libraries in Python are called packages, as in "PyPi packages", which you need to install on your machine (e.g., in a particular virtual environment) in order to be able to use them. The term `package` that we will mostly use for the rest of the book is a Python language construct, and it is sometimes called the `package module`. A PyPi package may include a Python program organized as a package module, among other things, but nonetheless they are distinct concepts.

The code from the line 4 to line 12 is a "function definition". We have seen something similar in an earlier lesson. This function, `to_string`, is a bit longer than the previous `hello` function, but you can easily notice that it has a similar structure, with the indentations and what not.

(Note that we are quickly going through the high-level structure of the program. You do not have to "understand" everything at this point. We will come back, throughout the book, to each of these important topics such as `import` statements and function definitions, etc.)

The "main part" of the program starts from line 15. First of all, notice the overall program text structure, as written. The lines are "grouped" in some way. Some are separated by one empty line and some by two empty lines. As stated, the Python interpreter (mostly) ignores empty lines, but they are for us and our fellow programmers.

One thing to note is that Python has no concept of "blocks" (e.g., using a pair of curly braces), as is common in other C-style languages. (Those languages are often called the "block-scoped languages", as we will discuss further later in the book.) In Python, there is a function. (And, there is a class, as we will see later.) And there is the rest. Functions (and classes) can be "nested". (That is, a function definition can include

another inner function definition, etc.) But, that's about it.

Hence, when writing Python programs, it is important to thoughtfully use empty lines to "group" related statements together *for readability*. You can easily imagine that a gap of one empty line and that of two empty lines have different significances. (E.g., the gap of two empty lines is bigger. 😊)

As stated, if you use the *autopep8* extension (or, something comparable) in your code editor (like VSCode), then it automatically enforces the standard formatting rules, known as "PEP 8".



Python is an (open-source) community-driven project, and the changes/improvements to the Python language and its standard libraries are made through what is known as the "Python Enhancement Proposals" (PEPs). The Python standard formatting rules were accepted/adopted as PEP 8.

For instance, there are always gaps of two empty lines between the definitions of functions (or, classes) and the rest, as mentioned before. Another example is, you should use the gaps sparingly in a single compound statement. You (almost) never use two line gaps within a compound statement.



Again, you do not have to "study" or memorize PEP 8. You just learn these rules, and conventions, by reading, and imitating, other people's (well-written) code.

The code in lines 15-25 reads a user input (as a string), using the builtin `input` function, and converts it into one of the strings that have special meanings in our program, namely `"r"`, `"p"`, or `"s"`. We will see how to make them "more special" later in the book. The line that starts with the hash symbol (`#`) is a program comment. Or, more precisely, the part of a line from the hash sign, if any, to the end of that line is a comment.

In this example, the comment in line 15 says that the next set of statements have

10.2. Import Statement

something to do with reading the "user input". Not super-useful ☺ (because that is almost obvious by reading the code). But, even in this trivial case, notice that the comment also plays the role of separating parts of a program (just like empty line gaps). Commenting is an art, which you will (most likely) learn through experience.

Just to be clear, there are different kinds of comments. This particular comment on line 15, on the one hand, describes the implementation, which may not be really needed in this case. But, on the other hand, it also describes (partly) how the program works. This may not be the best place for this, but programs/apps generally need documentations. "Public APIs" also need proper documentations. We will come back to this topic later in the book.

The lines 18 through 25 form a single compound statement as we can (more or less) guess based on the indentations. We will discuss the `if` statement shortly.

The next program segment, from line 27 to 35, generates a random number from a set of 0, 1, and 2, and it maps the number to one of the special strings "r", "p", and "s", for Rock, Paper, and Scissors, respectively. We use the function `randint` from the `random` module (line 29). And, we also use the conditional `if` statement here (lines 30-35).

Then we print out both hands, for the user's reference, to the terminal in lines 37 and 38. Notice that although the function call `print()` is a "simple statement", it is written in two physical lines. The argument of the `print` function (an expression) uses the function that we just defined in the beginning of the program, `to_string`.

Finally, in lines 41-49, the program concludes by printing out the result. If both hands are the same, it is a draw, or tie, Otherwise, the player either wins or loses according to the rock paper scissors game rules.

10.2. Import Statement

A module is a file containing Python definitions (e.g., functions and classes) and other statements. As stated, there is no difference between a Python script (e.g., which corresponds to an executable program, more or less) and a Python module

(e.g., which corresponds to a library in other programming languages, more or less).

It is the way we use the source files that differentiates them. Based on the intended use, we make slight adjustments (mostly as a convenience) when writing the code. Otherwise, at least in theory, a single Python source file can be used either as a *runnable* script or as a *library* module. Or, both.



There are also small differences at run time when a Python file is run as a script as opposed to when it is just "imported" as a module in other scripts. We will discuss this further in the following lessons.

We use the Python keyword `import`, in a script or module (which is one and the same thing), to import the definitions provided in another module.

You can import a module more than once in the same file. Although this use case is relatively rarely found in programs (the non-interactive mode) it can be useful in the interactive mode. Also it should be noted that an imported module can include other modules, etc., and a script may end up (directly and/or indirectly) importing the same module more than once.

The statements in an imported module, including all definitions, are executed only once when they are imported for the first time. Statements (that are not definitions) in a module (e.g., in a Python file that is *intended* to be used as a module but not as a script) are typically used to "initialize" the module.



You can use the `importlib.reload` function from the `importlib` module to reload a module that has changed since the last import. This comes in handy when you want to experiment with your code, e.g., in the interactive mode, during the development. Incidentally, we do not cover the "code debugging" in this book. When you are just starting to learn programming, the `print` function is your best friend. Print out *everything* in your program. 😊

There are a few different syntaxes for importing modules and other names. You can

10.2. Import Statement

either import a module or just import one or more (or, more or less all) names defined in a module.

In our example (line 1),

```
import random
```

The script, `main.py`, is importing the standard module `random`. Now we can use *all* names defined in the `random` module in our program (a script or a module). So, how do you know what kind of names and definitions are available in this particular module. Over time, with experience, you will gradually become more familiar with certain libraries that you tend to use more. But, for now, if you use an IDE or a code editor like VS Code, it tells you what names can be used after `random`. (with the dot). This is often called the "intellisense" (or, the "code intelligence", etc.).

Alternatively, we can always use the Python REPL. We used the `dir` function before. Let's try it here:

```
>>> import random ①
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST',
'SystemRandom', 'TWOPI', '_ONE', '_Sequence', '_Set', '__all__',
'__builtins__', '__cached__', '__doc__', '__file__', '__loader__',
'__name__', '__package__', '__spec__', '_accumulate', '_acos', '_bisect',
'_ceil', '_cos', '_e', '_exp', '_floor', '_index', '_inst', '_isfinite',
'_log', '_os', '_pi', '_random', '_repeat', '_sha512', '_sin', '_sqrt',
'_test', '_test_generator', '_urandom', '_warn', 'betavariate', 'choice',
'choices', 'expovariate', 'gammavariate', 'gauss', 'getrandbits',
'getstate', 'lognormvariate', 'normalvariate', 'paretovariate',
'randbytes', 'randint', 'random', 'randrange', 'sample', 'seed',
'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate',
'weibullvariate']
```

① Note that we need to `import random` first in this case even if it is part of the

"standard" library. There is a difference between "builtin" and "standard". So... what's the difference? 😊

As we can see, the `random` module includes (or, "exports") a few dozen names, including `randint`, `random`, and `choice`, etc. Now we can use all these names with the "dot syntax" (prefixed by the module name). How do you know what they are exactly? Again, you can use documentations, or Python REPL (e.g., `help()`). We will do this later when we discuss the `random.randint` function.



The "standard library" modules are (almost) always *available* in a Python interpreter, that is, without having to explicitly "install" them on a local development computer (or, within a virtual environment). The Python interpreter implementation that we use ("CPython") includes all the standard modules. We will still need to `import` them (or, their names) into the current script/module, however, in order to be able to use them.

The *builtin* types and functions, or methods, are *always* available no matter what since they are *built into* the Python interpreter implementation. No need to "import" them.

Another way to import a name(s) is to use the `from import` syntax. For instance,

```
from random import randint
```

This statement only imports the name `randint` from the `random` module, but nothing else. In this case, you can use the name directly without the module name qualifications. That is, in this example, you can use the name `randint` (which happens to be a function) instead of `random.randint`. One can import more than one names using this syntax. For example,

```
from random import randint, random
```

10.3. Function Definition

In this case, we can use both `randint` and `random` (both of which happen to be function names) in our script.

Alternatively, one can import "all" names in a module using the "wildcard syntax":

```
from random import *
```

What does "all" mean in a particular module is module-dependent. This is beyond the scope of this book, but *by default*, all names that do not start with the *underscore* `_` are importable through this wildcard import syntax.

10.3. Function Definition

A new function can be defined using the keyword `def` as we have seen before. In this rock paper scissors program, we create a function `to_string`, which takes a single argument of type `str` (string) and returns another `str` value. In the example code above, this function takes one of the "special" strings, `"r"`, `"p"`, or `"s"`, and it returns its "text representation".

```
def to_string(hand: str) -> str:
    pass # The function body goes here.
```

The text following the hash sign `#` is a comment, as we have seen before, and it is ignored by the Python interpreter. (One can see a trace of Python's origin from this comment syntax. Python was originally influenced by Shell scripting and other script languages like Perl.)

As stated, we informally use the type annotations in this book, primarily for our own benefits. This is equivalent to the following:

```
def to_string(hand):
    pass
```

①

- ① The function definition starts with keyword `def`, a function name and a parameter list, and a colon (`:`), followed by a function body (one `pass` statement, in this example).

The function parameter name `hand` (as well as the function name `to_string`) is arbitrary, but a good choice of names will make your programs more easily understandable by other people. The Python interpreter does not care what kind of names we use as long as they are syntactically valid. In this case, by using the name `hand`, we indicate that the function expects a "hand", likely one of "rock", "paper", and "scissors" or something similar, as an argument. We can also use "comments" (in a broad sense), as a documentation purposes, as we will see later.

The "function body" is indented "one level" (e.g., 4 spaces) with respect to the `def` line that ends with a colon `:`.

The `pass` statements in these examples are just placeholders. At least one statement is needed in the function body (e.g., the part after the colon, normally the indented part). Using two empty lines before and after the function definition (lines 2-3 and 13-14 in our example) is a recommended style (PEP8) as explained in the beginning of this lesson.

In our example program, the `to_string` function implementation includes *one* (compound) conditional statement, which comprises four "clauses" and which is written across eight physical lines.

```

4 def to_string(hand: str) -> str:           ①
5     if hand == "r":                         ②
6         return "Rock"
7     elif hand == "p":
8         return "Paper"
9     elif hand == "s":
10        return "Scissors"
11    else:
12        return ""

```

10.3. Function Definition

- ① The function `def` compound statement, which includes...
- ② the `if` compound statement, lines 5~12.

One thing to note about functions is that functions are "objects" in Python, just like everything else. When the Python interpreter "executes" a function `def` statement, it does not execute the statements in the function body. Instead, it creates a function object in memory, which can be used to "call" that function (e.g., using a pair of parentheses).

Just for fun, let's play with functions just a little bit. 😊

Any object in Python has "attributes". We have seen some of them for the builtin types and objects, e.g., using the `dir` function. Attributes have two kinds, the data attributes, which are often called the fields or properties or data members in other programming languages, and the function or method attributes, or methods for short.

For the user defined objects, such as "function objects", and "class objects" and "instance objects", as we will discuss later, we can add any attributes. We can do that even for the function objects.

Let's try the following in the Python REPL:

```
>>> def fn():                                ①
...     print("I am a function")
...
>>> fn                                        ②
<function fn at 0x7fda1f87d2d0>                ③
>>> id(fn), hex(id(fn))                      ④
(140574808593104, '0x7fda1f87d2d0')
>>>
>>> type(fn)                                 ⑤
<class 'function'>
>>> fn()                                      ⑥
I am a function
```

- ① A function definition, named `fn`.
- ② The value of `fn`.
- ③ This function object has no user-facing value. It just prints out the "internal value" that includes the object's memory location.
- ④ The `id` value of the `fn` happens to be the same as the memory address (in decimal).
- ⑤ The builtin `type` function that we are familiar with by now. ☺
- ⑥ We call this function with an argument list within parentheses, which happens to be empty in this example.

We have defined a function `fn`, which happens to be stored at a certain memory location (e.g., `0x7fda1f87d2d0`, in this particular execution). Note that *CPython* happens to use the memory address as the object's identity, but that is an implementation detail. (The builtin `hex` function returns a hexadecimal representation of a given `int`, as a string.)

The type of `fn` is `function`. We can *call this function*, and it prints out the text *I am a function* as expected.

This function object has certain attributes (coming from the type `function`):

```
>>> dir(fn)
['__annotations__', '__builtins__', '__call__', ... ]    ①
```

- ① Most of the attributes are omitted for brevity.

Now, interestingly, we can add any additional attributes to this object. For example,

```
>>> fn.a = 33
>>> fn.a
33
>>> def donut():
...     print("I am Homer Simpson. I want donuts.")
```

10.4. Comparison Operators

```
...  
>>> fn.f = donut  
>>> fn.f()  
I am Homer Simpson. I want donuts.
```

We added two extra (arbitrary) attributes, `a` and `f`, to this object. `a` is a data attribute and `f` is a method. If we list all attributes of this function object at this point,

```
>>> dir(fn)  
['__annotations__', '__builtins__', '__call__', ... , 'a', 'f']
```

It now includes the two extra attributes, `a` and `f`. If you are familiar with some statically typed programming languages, then this may all seem like a hocus pocus. *How can you do that with a function?* ☹️ But, this is *Python*. 😊



We will not go deep into the Python internals in this book, but notice the method attribute, `fn.__call__`. This is what makes the `fn` function object different from other objects like the number `3`. This attribute makes `fn` "callable". That is, `fn()` is a valid expression whereas `3()` is not.

Note that `donut` is just a name. Although it uses a different syntax to introduce the function name, e.g., through the `def` statement, and not through the assignment statement (which we learned in the first part of the book), it is nonetheless just a name.

10.4. Comparison Operators

We looked at the `bool` type, and the Boolean expressions that evaluate to `bool` values, in the introductory lessons. The "comparison operations" are Boolean expressions. Python includes all the usual (binary) comparison operators found in most programming languages, such as

- `<`: Less than
- `<=`: Less than or equal to
- `>`: Greater than
- `>=`: Greater than or equal to
- `==`: Equal to
- `!=`: Not equal to

These operators may be used between the objects of different types, and not just between the numeric objects. If a given operation is not supported for the two operands, then a `TypeError` exception is raised.



The word "binary" means, in this context, that the operator takes *two* operands.

Here are some examples (in the REPL):

```
>>> 2 < 3
True
>>> "aa" < "b"
True
>>> True < 3
True
>>> 2 < "3"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'int' and 'str'
```

In addition, Python includes other comparison operators such as `is` or `not is`, or `in` or `not in`, etc. We will not use these operators in this book. The readers are encouraged to look them up, if necessary.



Remember, you do not have to boil the ocean today. ☺

10.5. The `if` Statement

But, if you are an impatient type, then there are multiple ways to get this kind of information. Whatever way works for you, it is best to get into your own "routine" when it comes to looking up Python-related information. Some developers prefer just to use the Python REPL (say, instead of doing Web search).

You can get into an interactive help session by calling the `help()` function (without any arguments). Then, try typing `is` at the help prompt (e.g., after `"help> "`). Or, even just `>` (e.g., `"help> >"`). By the way, you can get a complete list of the Python keywords by typing "keywords" in the help session.

One of the interesting things about comparisons in Python is that these operators can be "chained". That is, a comparison like `-2 < 0 < 5` is a valid expression. This expression is equivalent to `-2 < 0 and 0 < 5`, where `and` is the Boolean AND operator in Python. If both operands of `and` are `True`, then the `and` expression yields `True`. Otherwise, it is `False`. Hence, the value of `-2 < 0 < 5` is `True` because both `-2 < 0` and `0 < 5` are `True`. (What about `1 != False == 0`? 😊)

Note that we have seen some similar behaviors before. For example, although the string concatenation is a binary operation, we were able to concatenate three (or more) strings (sort of) "at the same time" (more like, from left to right). This kind of behavior applies, in fact, to all arithmetic operations, for instance. The only caveat is that we will have to consider the "operator precedence rules". More on this later.



The Boolean `and` and `or` operators in Python are a bit tricky. We will discuss these operators further a bit later in this part.

10.5. The `if` Statement

The entire program of `rps/main.py` of this lesson essentially consists of three `if` statements. Even the `to_string` function is practically one `if` statement. 😊 The keywords `if`, and `elif` and `else`, are used to create *conditional* statements in Python. The `if` statement is a compound statement.

As we learned, a compound statement is a statement that includes one or more other (simple or compound) statements. A compound statement can, and generally does, span multiple physical lines, as in this example. The function `def` statement spans multiple lines. Likewise, the `if` statement in this `def` statement spans multiple lines. Notice the cascading indentations.

```
4 def to_string(hand: str) -> str:
5     if hand == "r":
6         return "Rock"
7     elif hand == "p":
8         return "Paper"
9     elif hand == "s":
10        return "Scissors"
11    else:
12        return ""
```

The lines that start with `if`, `elif`, and `else` (e.g., lines 5, 7, 9, and 11) are indented by "one level" (e.g., 4 spaces) because they are part of the `def` function statement, whose indentation level is zero, in this example. All `return` statement lines (lines 6, 8, 10, and 12) are indented by "two levels" (e.g., $4 * 2 = 8$ spaces) because they are part of the `if` statement, whose indentation level is one.



As stated before, Python allows more permissive indentations. But, most Python programmers follow the stricter indentation rules based on the discrete "levels" in a program file, as well as across all the programs they write, e.g., for consistency. The recommended style (e.g., PEP 8) for "one level" is 4 spaces.

As for the `if` statement in the `to_string` function definition,

```
if hand == "r":
    return "Rock"
elif hand == "p":
    return "Paper"
```

10.5. The `if` Statement

```
elif hand == "s":
    return "Scissors"
else:
    return ""
```

If the given argument `hand` is the same as `"r"`, that is, if `hand == "r"` evaluates to `True`, then the `if` clause is executed, which is `return "Rock"`. The `return` statement "returns" the given value to the caller of the function. In this case, the function call `to_string(hand)`, or `to_string("r")` since `hand == "r"`, will be evaluated to `"Rock"`, the function's return value. The `return` statement can also be used without a value, and in such a case, the return value is `None`.

When this function is called with `hand` that is not the same as `"r"`, the statements in the first `if` clause are ignored and the program control moves on to the next `elif` clause, if any is present. In this example, the `elif` clause includes a Boolean expression, `hand == "p"`. If this expression evaluates to `True`, that is, if the function is called with an argument `"p"`, then this particular `elif` clause is executed. In this case, that happens to include a single statement, `return "Paper"`, which terminates the function execution and returns the value `"Paper"` (to the caller).

The same with the following `elif` clause, `elif hand == "s": return "Scissors"`. Note that we read the `if` statement as "if true do this; otherwise do that" rather than "if false do that; otherwise ...". In some languages, there is an "unless" statement (e.g., in addition to "if"). In Python, it is always "if true do this; otherwise ...". If you want to write it as "if false do that; otherwise ...", then you will need to use the `not` operator, as we will see shortly.



Note that, unlike in many C-style languages, Python's `if` statement syntax does not require parentheses around the Boolean expressions in the `if` and `elif` lines. Instead, these lines terminate with colons.

The (optional) `else` clause in this `if` statement returns an empty string `""` if none of the previous expressions evaluates to `True`. That is, if the given argument `hand` is

none of `"r"`, `"p"`, and `"s"`, then the control passes to the final `else` clause, if one is present. If no `else` is found, this particular function will simply return (with value `None`) because there is no more statements after this `if` statement. The `return` statement is always implied when it reaches the end of a function.

In this particular example, we explicitly return an empty string instead of implicitly returning `None`. (And, that is what we "promise" (e.g., to the users of this function), by declaring it as `to_string(hand: str) → str`, and not in any other way.)

As we learned in the earlier lessons, the value of a *function call* expression is the function's return value.



Try calling this function `to_string(hand)`, e.g., in your head 😊, with a few different `hand` values, like `"a"`, `"bee"`, `"r"`, and `"See"`, and see which clause, out of these `if`, `elif`, `elif`, and `else`, is executed.

The `to_string` function definition is equivalent to the following:

```
def to_string(hand: str) -> str:
    text: str = ""

    if hand == "r":
        text = "Rock"
    elif hand == "p":
        text = "Paper"
    elif hand == "s":
        text = "Scissors"

    return text
```



How can you "prove", or at least verify, that these two function definitions are "equivalent"?

10.5. The `if` Statement

Note that an `if - elif - else` statement with one `elif` clause, for instance, is equivalent to two nested `if - else` statements. That is,

```
if a:
    print("a")
elif b:
    print("b")
else:
    print("c")
```

vs.

```
if a:
    print("a")
else:
    if b:
        print("b")
    else:
        print("c")
```

The above two statements are more or less equivalent to each other, for any two expressions, `a` and `b`. (Why do you think that is?) Note the indentation differences. It is generally preferred to use the non- or less-nested ("flatter") versions over the (more deeply) nested versions. (Remember the Zen of Python? 😊)



How can you verify that these two statements are equivalent?

Any expression in a Boolean context will have either `True` or `False`, only one of these two values. Hence, there are only four different possibilities as far as these `if` statements are concerned, `a == True - b == True`, `a == True - b == False`, `a == False - b == True`, and `a == False - b == False`. In all these four cases, these two statements behave exactly the same way. That is, both of them print out the same string, `"a"`, `"b"`, or `"c"`, in each of

these four cases. Hence, they are equivalent.



Now, can you re-write the **to_string** function using the nested **if-else** statements (without using **elif**)? Just for fun? 😊

The other three **if** statements work more or less the same way. We will leave it to the readers, as an exercise, to go through each **if** statement and understand how they work.

10.6. Builtin **input** Function

The statement on line 17 uses another builtin function **input**.

```
17 u = input("Rock (r), Paper (p), or Scissors (s)? ").lower()
```

This is sort of the "opposite" of the **print** function, which does the output. The **input** function handles the user input (from the terminal/console).

This function can be called with zero or one argument. If **input** is called with one argument, like **input(prompt)**, then the **prompt** argument is written to standard output first (without adding a trailing newline). The **input** function then reads a line from the input as a string, and it returns the string object (after stripping a trailing newline, if any).

For example, if you run the following program,

```
s = input("How old are you? ")  
print(type(s))
```

You will get the following prompt, and it waits for the user input.

10.6. Builtin `input` Function

```
How old are you?
```

If you input, say, `101`

```
How old are you? 101
```

Then, it will print out the type of the input value. The entire output may look like this:

```
How old are you? 101
<class 'str'>
```

As stated, the `input` function returns the user input value as a string *when a valid input* is provided. We will discuss some error handling in the next lesson.

The `input()` function call in the `rps/main.py` program (line 17) returns the user typed input value, and this value is assigned, or "bound", to a "variable" `u` (all in one statement). (We will discuss the builtin string `lower` method shortly.) Note that, as stated, you do not have to (pre-) declare a variable (or, in fact, any name) before its use in Python. A name is just an alias or reference to an (existing) object. Let's go over this in some more detail in the next section (with a plenty of repetitions as well in case you have been dozing off 😊).

In this particular example, however, it *appears* that the variable, `user_hand`, has been declared before its use, line 16.

```
16 user_hand: str
```

This is special. The sole purpose of this line is to give the type annotation to the variable, `user_hand`. In fact, as indicated before, declaring a variable without an initial value is not allowed in Python. The type annotation is an exception, sort of.

In this case, the variable `user_hand` is bound in multiple places (in code). Depending on the value of `u` (essentially, the value of `input()` function call), the variable might be *first* bound in lines 19, 21, 23, or 25. Hence, the type annotation, if needed, should be done before this `if` statement, not in each of these separate clauses.

10.7. Variables/Names

As briefly alluded before, in many C-style programming languages, the *variables* come first. They "hold" the values or references/pointers.

In Python, on the contrary, the *objects* come first. Variables, or names in general, are used to refer to objects. Objects can be immutable or mutable. (Hence, names in Python do not refer to values. They refer to the objects.)

These two different "interpretations" can lead to the same explanations in many scenarios, but that is not always the case. There are certain concepts that are hard for the beginners to understand in C-style languages, like "pass by reference". But, in the Python's (conceptual) framework, they, or at least some of them, become trivial.

Variables are also called the *references* in Python, but as stated, this term is (subtly) different from the same term used in other programming languages like C/C++, Java, C#, Go, Rust, or even Javascript.



That is a long list. (And, we haven't even started. 😊) That is why we often compare Python with these *C-style* languages in this book. They are much more prevalent. The readers can benefit from these comparisons, now or in the future, regardless of their programming background.

A name in Python can be *bound to*, or *unbound from*, an object. We have seen a couple of examples of a name binding earlier, that is, through the assignment and function definitions. A new name does not have to be declared before binding. In fact, there is no way in Python to just declare a new name without binding it to an existing object unlike in many other programming languages. (As stated, there is no

such thing as the "null pointer exception" in Python.)



In Python, there is a certain syntax that allows us to introduce a name into a "scope" without explicitly binding it to an object, for example, using the keywords like `global` and `nonlocal`. But, even then, the names have to refer to existing objects before use, e.g., at run time.



One other thing to note is that, you can just declare a variable and initialize it with `None` in Python. This is a valid assignment. But, it is really not the same as the variables being `null` in other C-style languages. `None` in Python is a valid *real* object unlike `null`. Furthermore, there are very few circumstances where you need to introduce variables with the initial `None` value when you program in Python. (You can say *that is not Pythonic*.)

Here's an assignment statement:

```
three = 3
```

`3` is an object (with a certain unique identity) which has a type, `int`, and a value `3`. (As mentioned, in some cases, the distinction between "objects" and "values" becomes somewhat blurry, especially for the immutable objects. This is because we often use their values to refer to the objects themselves, and not because these two concepts are the same.)

The object `3` is immutable (since its type is `int`, a built-in simple type). Through this assignment, the name `three`, which may or may not have existed before this statement, is now bound to, or refers to, the object `3`. An object can be referred to by more than one names.

```
tres = 3
```

Now, the names `three` and `tres` refer to the same object, `3`. As stated, an object can have zero, one, or more names at any given moment. A name can be unbound from and/or bound to a different object. For example, continuing with this example,

```
three = 5
```

This statement first unbinds the name `three` from the currently bound object `3`, and re-binds it to a different object `5`. Now `three` refers to `5`, and the object `3` has only one name `tres` at this point. (BTW, in this particular example, we realize that the choice of names like `three` and `tres` may not have been the best, e.g., because these same names may be used to refer to something else.)

We can also unbind a name without rebinding it to a different object. Python's builtin function `del` can be used for this purpose. For instance,

```
del(three)
```

This statement, e.g., a function call, unbinds the name `three` from the currently-bound object `5`. The name `three` no longer exists at this point (since a name cannot exist without a bound object in Python). If we try to use the name at this point, we will get an error, "`NameError`". E.g.,

```
NameError: name 'three' is not defined
```

Names can be part of an expression, and a name itself is an expression that evaluates to (the value of) the bound object.



Note that the `del` function merely unbinds the name, and it does not *delete* the object that the name refers to. If an object has no more names/references, then the object might be "garbage collected" by the Python interpreter. We will discuss Python's

garbage collection *some other time*.

As we discussed, the function `def` statements introduce the function names to the local scope, just like assignments. And, the function names are just names, bound to the function objects.

```
>>> def bart_simpson():
...     print("I didn't do it!")
...
>>> bart_simpson()
I didn't do it!
>>> el_barto = bart_simpson
>>> del(bart_simpson)
>>> bart_simpson()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'bart_simpson' is not defined
```

There is nothing special about the function name, `bart_simpson`. The builtin `del` function only deletes the name, and not the object referred to by the name. We can still call the function object using a new name `el_barto`:

```
>>> el_barto()
I didn't do it!
```

Although the original name `bart_simpson` is gone, the function object remains, and it can still be referenced by this new name, `el_barto`.

We explained the names and assignment earlier in the introductory lessons, and we will discuss more about the name binding throughout the rest of this book. There are a few more important, and basic, circumstances where we will need to pay more close attention. There are also (subtle) differences in the name bindings for immutable and mutable objects.

One other thing to note is that a name cannot be used before it is introduced to the program. A `def` statement, for example, lines 4 through 12 in our rock paper scissors program, introduces a new name, `to_string` in this example, in addition to providing the function definition.

If we try to call the function `to_string` prior to this statement, we will get a `NameError`. Note the order in which program is written in this example. We could not have put the function definition, lines 4-12, at the end of the program, for instance, because this function is used in the statement in lines 37 and 38.

This program uses the names, `to_string`, `u`, `user_hand`, `r`, and `computer_hand`. (The "function parameter" `hand` in line 4 is also a name, and it is different from the names `user_hand` and `computer_hand` used in lines 37-38, although they are related. We briefly touched on this in the introductory lessons, and we will discuss this further a bit later.) Notice that the general rule is observed for all these names that a name cannot be used before it is introduced/bound to an object. For example, the variable `u` is first bound to a certain value in line 17 and it is used in the statements in lines 18, 20, and 22, *all after the line 17*.

```

16 user_hand: str
17 u: str = input("Rock (r), Paper (p), or Scissors (s)? ").lower()
18 if u.startswith("r"):
19     user_hand = "r"
20 elif u.startswith("p"):
21     user_hand = "p"
22 elif u.startswith("s"):
23     user_hand = "s"
24 else:
25     user_hand = ""

```

10.8. Rules on Names

Python is rather unique in that you can use pretty much any (natural) languages for names, or "identifiers". In most programming languages, the names are limited to a

10.9. Naming Conventions

small set of letters from English (and numbers and a few different symbols). In Python, you can use your native language, for instance, to name variables and what not.

Although this is fantastic, and the readers are encouraged to try this out in their programs, who are not native English speakers, one thing to note is that it can potentially reduce the portability of the Python programs. At least in theory, you can share your Python programs with anybody in the world. But, not everybody can read, or write, all international characters ("Unicode") on their computers.

If you only use English, then all 52 lowercase and uppercase alphabets are allowed in a name. (Names are "case sensitive" in Python. That is, `Hello` and `hello` are two different names.) In addition, you can use all 10 digits and the underscore symbol, `_`, in names in Python (in English or otherwise). The names cannot start with digits (since numeric literals all start with at least one digit/number).

10.9. Naming Conventions

One can use any (random) sequence of valid letters as a name in Python programs. However, it is customary to use the names that include one or more (English) words. When you use more than one word in a name, there are a few commonly used conventions.

In one convention, the first letter of each word is capitalized except for the first word. For example, `fastElectricCar`. This is called the camel case (or, the lower camel case). A variation is using a capital letter even for the first word, e.g., `BlueSky`. This is called the Pascal case (or, the upper camel case).

In another popular naming convention, the words are concatenated with the underscores (`_`) in between. For example, `cruel_month`. This is called the snake case. In this convention, only lowercase (English) alphabets are typically used. (Or, all uppercase letters in some cases.)

In Python, we use the snake case names for variables and custom (user-defined) functions, among other things. On the other hand, we use the Pascal case names for

the user-defined types (which we will discuss later in the book). In the example code of `rps/main.py` we use the snake case names, `to_string`, `user_hand`, and `computer_hand`, for the function and the two variables.

Just to be clear, the Python interpreter does not care as long as all names in the program are syntactically valid. But, as you will see more as you do more programming in Python, these naming, and other stylistic, conventions are (almost strictly) followed by experienced developers. And, there is no reason for you not to. ☺ These conventions make the Python programs easier to read and understand.

As a general rule, we use shorter and concise names for the variables that "are used only for a short while". On the other hand, we use longer and more descriptive names for those that "live longer". (No precise definitions can be provided for these. ☺)

For example, the name `u` is used only within a small segment of the program, e.g., from line 17 to line 25. What `u` means is clear in this context, e.g., from the statement of line 17. That is, `u` is a name for the object that is just read from the `input()` function call. What `u` stands for, if any, is not that important. Any more descriptive name would not have increased the readability much.

The name `user_hand`, on the other hand, is used much more broadly, or longer, depending on whether we use space or time as a metaphor. In this example, `user_hand` is used from line 16 (or, more like line 18 or 19) and pretty much to the end of the program (line 49). In the last "segment" of the program (the `if` statement of lines 42-49), which is further down from where the name `user_hand` is first introduced (e.g., lines 16-25), it may be useful to easily recognize what this name means. In this case, it means the "user hand". Any other short or non-descriptive name like `xyz` would have made the program harder to understand.

This is a general rule. As stated, Python is not a *block-scoped* language, and the "longer range" names should have longer and/or more descriptive names, especially in large programs. This is also true for the "exported names" that are meant to be shared with other scripts and modules,

10.10. Scopes

This will be surprising, or even *shocking* 😲, to some readers who have experience in other C-style programming languages, but Python is not a "block scoped language". In other words, there are no "curly braces", or anything equivalent, in Python.

In the example code of this part, everything is in one *top level* scope (or, the "global" or "module" scope). The names, `to_string`, `u`, `user_hand`. etc. are all in one scope. The only exception in this example is the function parameter `hand`. A function or class definition, for instance, creates another scope.

The name, `user_hand`, for instance, seems to be declared in the `if` statement "block" (lines 18-25), that is, if you are coming from the C-style language background. (Again, we ignore the type annotation on line 16.) But, we use it outside this "block", e.g., in lines 42-49. That *should not* be possible!!

```
18 if u.startswith("r"):
19     user_hand = "r"
20 elif u.startswith("p"):
21     user_hand = "p"
22 elif u.startswith("s"):
23     user_hand = "s"
24 else:
25     user_hand = ""
```

```
42 if user_hand == computer_hand:
43     print("Tie")
44 elif (user_hand == "r" and computer_hand == "s" or
45       user_hand == "p" and computer_hand == "r" or
46       user_hand == "s" and computer_hand == "p"):
47     print("You win")
48 else:
49     print("You lose")
```

Well, in Python, it is possible. ☺ Python does not have the concept of the C-style *blocks*. You do not "declare" names (e.g., within a certain block). The only important rule to remember is that, as explained before, you cannot use a name before it is first "bound" to some object.

Python *does* have the scoping rules (albeit much simpler), and you cannot use a name bound in another scope. For example, the name `hand` (a function parameter) is available only in the context of the function definition (lines 4-12), or more accurately, during a function call, and it cannot be used outside this scope. The same applies to all other names that are first bound in the function scope. (This particular function, `to_string`, does not use any other names.)

```

4 def to_string(hand):
5     if hand == "r":
6         return "Rock"
7     elif hand == "p":
8         return "Paper"
9     elif hand == "s":
10        return "Scissors"
11    else:
12        return ""

```

As mentioned, there are two exceptions. We briefly mentioned this earlier. The keywords, `global` and `nonlocal`. We will not use them in this book, and we will leave it to the readers as an "exercise". ☺ When would you use `global`, and when would you use `nonlocal`?



An astute reader might have noticed that we just "sprinkle" certain concepts and terms all over this book without fully explaining them. ☺ *This is deliberate*. Some books have a section titled "what's next?" at the end. This book does that throughout the book. The readers should not feel, after finishing this book, that they are "done". It is just a beginning. And, these "sprinkles" are used to remind the readers of what they do not know. ☺

10.11. String Methods

The builtin `str` type has many builtin methods (and, other "attributes"). How do you find that out? Well, there is always your good ol' friend, the Python REPL. You can see everything there is to see about the `str` type by calling the `dir(str)` function.

This rock paper scissors program uses two of those methods, namely, `lower` and `startswith`. What are these methods?

```
>>> help(str.lower) ①

Help on method_descriptor:

lower(self, /)
    Return a copy of the string converted to lowercase.
(END) ②
```

- ① Every time you see the interactive shell prompt ">>> ", you know that we are in the Python REPL.
- ② Press "q" to go back to the REPL prompt.

```
>>> help(str.startswith)

Help on method_descriptor:

startswith(...)
    S.startswith(prefix[, start[, end]]) -> bool

    Return True if S starts with the specified prefix, False otherwise.
    With optional start, test S beginning at that position.
    With optional end, stop comparing S at that position.
    prefix can also be a tuple of strings to try.
(END)
```

There seem to be small differences in the way the docs are generated, but in the case

of `lower`, the syntax `lower(self, ...)`, with `self` as the first parameter, indicates that it is a method, not a function. (In this help doc, the slash `/` signifies that the parameters preceding it can only be used for the "positional only arguments". You can ignore this for now.)

In the case of `startswith`, the notation `S.startswith(...)` with the prefix `S` indicates that it is a method (e.g., defined on the type string).



You do not have to understand everything. The line, `Help on method_descriptor:`, for example, indicates that these are *builtin methods* (e.g., for *builtin types*), if you are curious. 😊

As a matter of fact, one of the most important skills to learn as a beginning programmer is to be able to *program without knowing everything*, or more accurately, be able to *program while knowing pretty much nothing*. 😊 In all seriousness 😊, this book teaches you just that. As the saying goes, "give a man a fish and you feed him for a day; teach a man to fish and you feed him for a lifetime."

If you don't understand the doc, that is perfectly all right. We learn from examples. In our first rock paper scissors program, we use the `lower` and `startswith` methods to convert a user input text to one of our "special" strings, `"r"`, `"p"`, and `"s"` (lines 17-25).

```
17 u = input("Rock (r), Paper (p), or Scissors (s)? ").lower()
18 if u.startswith("r"):
19     user_hand = "r"
20 elif u.startswith("p"):
21     user_hand = "p"
22 elif u.startswith("s"):
23     user_hand = "s"
24 else:
25     user_hand = ""
```

10.12. Random Module

For example, if the user inputs a string that starts with "R", "P", or "S", we convert it to a lowercase string, whose first letter ends up to be "r", "p", or "s", respectively. Then, the `if` statement, in effect, sets `user_hand` to one of these starting letters. If the user inputs a string that starts with "r", "p", or "s", `user_hand` ends up to be its starting letter. For any other (invalid) input strings, we just convert it into an empty string (in the last `else` clause of the conditional statement). The result is then referred to as `user_hand` (throughout the rest of the program), regardless of which value ends up being selected.

Note that we can use the `str.lower` method on the object returned by the `input` function call since `input()` returns a string object (if successful). Calling a function/method on the result of another function/method call is rather common because it can possibly reduce the amount of code (that is only temporary). In this example, we could have assigned the result of the `input()` function call to a name first, and used the `lower` method on that name (or the object referred to by the name). For instance,

```
h = input("Rock (r), Paper (p), or Scissors (s)? ")
u = h.lower()
```

Here, we introduced a new (temporary) variable `h` just to be able to refer to the object returned by the `input()` call.



In terms of the program execution, there is really not much difference between the two styles. When you add error handling, or when you need to do debugging, etc., there are some practical differences, and one style may be preferred over the other in some cases. We will briefly discuss error handling in the next lesson.

10.12. Random Module

`random` is a Python standard library module which implements a "pseudo-random number generator". We `imported` this module in the beginning of the program, as

we described earlier.



This is a style, to a large extent. You can use the `import` statements (just about) anywhere in a program (*before* the imported module/names are used). But, by convention, we generally put all `import` statements in the "header" part of a program, or in the beginning of a function definition, etc.

Since we `imported` the `random` module as `import random`, now we can use all names in the module in our program with the dot prefix, `random..` For example, the program `rps/main.py` uses the `random.randint` function.

```
>>> import random
>>> help(random.randint)

Help on method randint in module random:

randint(a, b) method of random.Random instance
    Return random integer in range [a, b], including both end points.
(END)
```

① Don't forget to `import random` first.

Note that the doc says that `randint` is a method, not a function, but that distinction is not important for us. We use `randint` as a function (defined in the module `random`).

According to the help doc, the `random.randint` function returns a random integer number, e.g., from a "uniform distribution" between the first and second arguments, including both ends. In our example, we will get one of `0`, `1`, and `2` (from `random.randint(0, 2)`) with (more or less) equal probabilities. That is, we will have a one-third chance of getting int `0`, and so forth. The actual numbers, e.g., `0`, `1`, and `2`, are not important. The program relies on the fact that we get three objects with equal probabilities.

10.13. Boolean Operators

We then map 0, 1, and 2 to "r", "p", and "s", respectively, using the `if` statement.

```
if r == 0:
    computer_hand = "r"
elif r == 1:
    computer_hand = "p"
else:
    computer_hand = "s"
```

Again, this particular mapping is somewhat arbitrary. We will see, in the next two different implementations, how this can be done differently.

10.13. Boolean Operators

Python supports binary Boolean operators, `and` and `or`, and a unary Boolean operator, `not`. These correspond to `&&`, `||`, and `~` in most C-style languages, respectively, but there are some (subtle but significant) differences.

The operator `not` yields `True` if its argument evaluates to `False`. Otherwise, it returns `False`. That is,

```
>>> not True
False
>>> not not not False
True
>>> not 3                                ①
False
>>> not ""                               ②
True
```

① The Boolean value of non-zero number is `True`.

② The Boolean value of an empty string is `False`.

Python's `and` and `or` operators are based on the logical AND and OR operators.

Roughly speaking, if both operands of the `and` operator are true, then it returns `True`. Otherwise, it returns `False`. If both operands of the `or` operator are false, then it returns `False`. Otherwise, the `or` operation returns `True`. There are a couple of (some significant) differences, however.

First, as in many other programming languages, Python's `and` and `or` operators "short circuit". That is, if the first operand of the `and` operator is false, then the second operand is *not* evaluated. This is because the value of this `and` expression will be `False` regardless of the value of the second operand. Likewise, if the first operand of the `or` operator is true, then the second operand is *not* evaluated since the value of the whole expression is already `True`.



For experienced programmers, this is so automatic that they don't even think about it. They will in fact have a hard time understanding this verbose explanation. However, for beginners, it is rather important to understand this "short circuiting" concept. The result of a program may turn out different depending on whether we evaluate a certain expression or not.

In fact, it is a common practice to use this "short circuiting" on purpose, e.g., to prevent certain expressions from being evaluated under certain conditions, or to have certain expression evaluated only when certain conditions are met, etc.

Second, unlike in many C-style languages, Python's `and` and `or` operators do *not* necessarily return `bool` values. As a matter of fact, none of the arguments for all Boolean operators in Python, `and`, `or`, and `not`, need not be `bool` values. In case of `not`, the result is always Boolean, either `True` or `False`.

For `and` and `or`, the results depend on the types of their arguments. They, in fact, return the last evaluated argument (when its boolean outcome is determined, taking the short-circuiting into account). It can sometimes have significant consequences. We will not discuss this any further in this book, but the readers should keep in mind that Python's boolean operators behave differently.

10.13. Boolean Operators

In our rock paper scissors program, the `elif` clause in the final `if` statement (lines 44-47) includes a Boolean expression that may potentially require up to 5 `and` or `or` binary operations. Why 5?

```
42 if user_hand == computer_hand:
43     print("Tie")
44 elif (user_hand == "r" and computer_hand == "s" or
45       user_hand == "p" and computer_hand == "r" or
46       user_hand == "s" and computer_hand == "p"):
47     print("You win")
48 else:
49     print("You lose")
```

First, an expression `A or B and C` is equivalent to `A or (B and C)`, where the expression in the parentheses is computed together. This is because `and` has a higher "operator precedence" than `or`. We briefly mentioned this before. When (unary or binary) operations are chained together, operations that have a higher precedence are computed first as if they are enclosed in parentheses. (All operators in Python are ordered, in several groups, according to their operator precedence.)

In our example, because of the difference in the operator precedence between `and` and `or`, the `if` statement can be written as follows:

```
if user_hand == computer_hand:
    print("Tie")
elif ((user_hand == "r" and computer_hand == "s") or
      (user_hand == "p" and computer_hand == "r") or
      (user_hand == "s" and computer_hand == "p")):
    print("You win")
else:
    print("You lose")
```

The boolean expression in the `elif` clause has the form of `X or Y or Z`, which is equivalent to `(X or Y) or Z`.

The first `and` expression (`X`) will be computed first. If it is `True`, then the whole expression (in the `elif` clause) will end up to be `True` and therefore the rest of the expression is not computed. In particular, `Y` and `Z` will not be evaluated. The two `or` expressions `((X or Y) or Z)` just short circuit, and the statement(s) in the suite, `print("You win")` in this case, is executed.

If the first `and` expression (`X`) evaluates to `False`, then the second `and` expression (`Y`) is evaluated next. And then we compute the `or` between these two values (`X or Y`). If this turns out to be `True`, e.g., because the second `and` expression (`X`) is `True`, then the second `or` operation (`True or Z`) again short-circuits, and the third `and` expression (`Z`) is not evaluated. It just executes the suite since the value of the overall expression is `True`.

If the value of the second `and` expression (`Y`) is `False`, then the value of the first `or` expression (`False or Y`) is `False`, and hence the third `and` expression (`Z`), and then the second `or` expression (`False or Z`), are evaluated. Depending on its value, which is the value `Z` in this case, the statements in the `elif` or the `else` clauses might be executed.



So, did you count that there could be up to 5 evaluations for this `elif` Boolean expression? 😊 Regardless, as stated, this will become natural to you and, most of the time, you do not have to go through a process like this when you program. This is just an illustration.

10.14. Lines in Python

Python programs are "line-based", so to speak. More or less. Contrast this with other C-style programming languages, in which a newline is just another white space. More or less. Spaces, tabs, and newlines are (syntactically) mostly interchangeable in those languages.

In Python, there is roughly one-to-one correspondence between a statement and a "logical line". A logical line can span one or more physical lines (e.g., separated by newlines). We have seen many such examples, including the function definitions

and the `if` statements (both of which are compound statements).

On the flip side, Python expressions and statements cannot be arbitrarily broken into multiple physical lines. For example, an arithmetic expression should be written in one line. `1 + 2` cannot be broken into two lines, like `1 +` in one line and `2` in the next. You can try `1 +` in the Python shell and see what happens. You cannot do `1` in one line and `+ 2` in another either. They become two separate expressions, which may or may not be syntactically valid in a given context.



In (most) C-style languages, it is the semicolon (`;`), not the newline, that terminates a statement. Hence the newline symbols, or line breaks, are not that special in those languages, in many circumstances.

In the last `if` statement of our program, the `elif` keyword needs to be followed by a Boolean expression that should be in one line (lines 44-46). Clearly, the expression is too long and it may make the program harder to read.

Interestingly, Python allows line breaks within a pair of "brackets". Any brackets. The sample code uses a pair of parentheses to "group" the expression (which is not really required in general), and it breaks down the Boolean expression into three lines. This is possible only because we use the extra parentheses.

This program includes another such example. The `print()` statement, lines 37-38, includes an argument that is a string concatenation expression.

```
37 print("You: " + to_string(user_hand) +  
38      " -- Computer: " + to_string(computer_hand))
```

Normally, the expression (with three `+`'s in this case) cannot be split into multiple lines. And yet, this happens to be inside a pair of parentheses (which is used for the function call), and hence it can be written in multiple (physical) lines.

As another example, lists or tuples can be written across multiple lines because they

are enclosed in some kind of *brackets*. For instance,

```
>>> [1,
... 2
... , 3]
[1, 2, 3]
```

Yes, that's "ugly". But it is syntactically valid, and it is *just* an illustration. 😊 Well, you can even do this:

```
>>> (1 +
... 2 +
... 3 +
... 4)
10
```

One thing to note is that, in this kind of situations, indentations are often not significant. For instance, in our program, the leading white spaces in line 38 and lines 45-46 do not follow the 4-space indentation rule. In fact, they are mostly ignored. (The white spaces may still play a role in separating multiple "tokens", e.g., between `or` in line 44 and `user_hand` in line 45. They have to be separated by at least one space, etc.)

10.15. Error Handling

Our first version of the rock paper scissors app is rather simple, and not much flexible. We will try a couple more different versions in the next two lessons, but they are not necessarily "better" than this simple program. They all have their pros and cons. The decisions we make while writing these programs can only be judged in the overall context (e.g., the business requirements, use cases, costs, etc.).

For example, if you are writing a rock paper scissors game just for yourself, then you know that, for instance, you will need to input one of "r" (for rock), "p" (for paper),

10.16. Putting It All Together

and "s" (for scissors). Neither "better user interface" nor "user-friendly error handling" will be needed. If an error occurs, you just play the game again. Clearly, this will not be the case when you are writing a program which will be used by other people.

The simple rock paper scissors program of this lesson gets the job done. No real error handling is needed, for instance. Having said that, you can still "break" the program. For instance,

```
$ python main.py
Rock (r), Paper (p), or Scissors (s)? Traceback (most recent call last):
  File "/home/harry/projects/rps/main.py", line 17, in <module>
    u = input("Rock (r), Paper (p), or Scissors (s)? ").lower()
EOFError
```

This particular error is a very unusual case. (Programmers often use the terms like "edge cases" or "corner cases" implying that they are not the common occurrences.) We are just including it here as an example of an unexpected error ("unexpected" in the sense that, for example, it is not obvious to get such an error just by reading the code). We will leave it to the readers to figure out how to get this kind of error. 😊



Well, if you have been paying attention throughout this book 😊, then all the clues are there.

10.16. Putting It All Together

Whoa! That was a lot of stuff. 😊

We learned how to define a new function in Python. A function definition consists of,

- The keyword `def` (as in "definition"), followed by
- A syntactically valid function name (e.g., in the snake case),
- A possibly empty (comma-separated) list of function parameters within a pair of

parentheses, with syntactically valid parameter names, if any,

- A colon `:`, and
- A function body (e.g., the indented part),

in this order.

The `to_string` function is defined as follows (omitting the type annotations):

```

4 def to_string(hand):
5     if hand == "r":
6         return "Rock"
7     elif hand == "p":
8         return "Paper"
9     elif hand == "s":
10        return "Scissors"
11    else:
12        return ""

```

The function body includes a single conditional compound statement comprising four `if/elif/else` clauses. We went through how to read this `if` statement, for a given argument `hand` (a `str`). In order to understand this function definition, you will need to know the general structure of the `if` statement, the equality comparison operator, and the `return` statement.

The `to_string` function does not play a central role in this program. This kind of functions are often called the "convenience" functions, "helper" functions, or "utility" functions, etc. The purpose of this function is to convert an internal symbol (e.g., `"r"`, `"p"`, or `"s"`) to the corresponding text form for output purposes. As we will see later, Python provides a "standard" way of creating the "string representation" of an object.

The "main script" starts from the part where we read the user input, lines 17-25:

```

17 u = input("Rock (r), Paper (p), or Scissors (s)? ").lower()

```

10.16. Putting It All Together

```
18 if u.startswith("r"):
19     user_hand = "r"
20 elif u.startswith("p"):
21     user_hand = "p"
22 elif u.startswith("s"):
23     user_hand = "s"
24 else:
25     user_hand = ""
```

Again, the type annotations have been omitted. Note that without a type annotation, the variable `user_hand` cannot have been declared alone without an initial value. Regardless, the scope of this variable is the entire program although it cannot be used until it is bound to some object. Where exactly that happens in this program is determined at run time (somewhere in this `if` statement).

This part of the program reads the user input and converts it into one of the special strings, `"r"`, `"p"`, and `"s"`. We use the `str.startswith` method to compare the first letter of the input string to one of those strings/characters. The expression `u.startswith("r")`, for instance, is more or less the same as `u[0] == "r"` using the index notation. We can use this latter expression as long as `u` has at least one letter. Otherwise, we will get an `IndexError` exception.

Since when the string `u` is empty, the desired value for either of these expressions is `False`, we can modify the Boolean expression as follows, `len(u) > 0 and u[0] == "r"`.

If `u` is an empty string, then the first operand of `and`, `len(u) > 0`, evaluates to `False`. And, due to the aforementioned short-circuiting rule, the second operand of `and`, `u[0] == "r"` will not be evaluated. Hence, there will be no `IndexError`. When `len(u) > 0`, `u` has at least one character, and hence there is no problem accessing its first character via `u[0]`.

In the next segment of the script, we generate a random hand using the `random` module's `randint` function.

```

29 r = randint(0, 2)
30 if r == 0:
31     computer_hand = "r"
32 elif r == 1:
33     computer_hand = "p"
34 else:
35     computer_hand = "s"

```

Note that, in this slightly altered version, we call the `randint()` function without the module prefix. This is allowed if we import the function name `randint` directly using the `from import` syntax. For example,

```
from random import randint
```



As to whether to import a module or only certain names in the module, there are no general rules. The author recommends, to the beginning Python programmers, to use the module import syntax, e.g., `import <mod>` (or, `from <pkg> import <mod>`), when importing "external" modules (e.g., from PyPi). The module name prefix in the imported names can help make the program more readable (other than the fact that it also reduces the chances of name collision).

On the flip side, the more direct import name syntax may be more preferred, e.g., `from <mod> import <name1>, <name2>`, when importing names from the "internal" modules (e.g., from the same project/program). Even the wildcard name import, e.g., `from <mod> import *`, may be used if there is no big chance of name conflicts and if it doesn't make the program too much harder to read.

This part of the program ends up with one of `"r"`, `"p"`, or `"s"` for the computer's hand. We use the `if` statement more or less in the same way as before to map an integer in the range, 0, 1, and 2, to one of the predefined strings, `r`, `p`, and `s`. As we

10.16. Putting It All Together

will see later, this can also be done using the new `match - case` statement syntax.

Note that we could have defined this branching logic (lines 29-35) in a separate function. The same can be said regarding the code that reads the user input and converts it to `r`, `p`, or `s` (lines 17-25). We will do this in the next lesson when we work on an alternative version of our rock paper scissors program.

Next, we display this information to the user/player:

```
37 print("You: " + to_string(user_hand) +  
38      " -- Computer: " + to_string(computer_hand))
```

Here we call our `to_string()` function twice, with two different arguments. We use the custom functions in the same way that we use the builtin functions. We use a pair of parentheses after the name of the function, and include the function arguments (e.g., `user_hand` or `computer_hand` in this case), if any, within the parentheses. A function call is an expression, and it evaluates to a value, of type `str` in this example.

When we call a function with certain arguments, the arguments (objects) are "shared" by the caller and the called function, for a lack of better word. Again, this is in contrast with the general viewpoint used in the C-style programming languages.

In C-style languages, when we call a function, we "pass" in an argument into the function, and depending on the argument type, the value of the argument may be copied ("pass by value") or its "reference" may be copied ("pass by reference").

In Python, there is only one object for a given argument. When we call a function, the callee function uses the parameter, a name/variable, to refer to the object. As explained earlier, the argument passing in Python has the same semantics as the assignment, or name binding.

In our program, we call the `to_string` function with `user_hand`, line 37. Let's suppose that `user_hand` happens to be `"r"` in this particular run. There is an object

in memory whose type is string and whose value is "r". The `user_hand` variable is just a name referring to this object.

Now, when we call `to_string()` with `user_hand`, the `to_string` function "takes over" (in a new scope), and it initializes its parameter, `hand`, with the object which `user_hand` points to, namely, the object "r" in this example. It is *the same object*. There is no "copying" or "value vs reference". Function parameters are just names.

Finally, now that we have both the player's hand and the computer's hand, we compare the two hands and decide on the winner, and we print out the result. Lines 42-49.

```
42 if user_hand == computer_hand:
43     print("Tie")
44 elif (user_hand == "r" and computer_hand == "s" or
45       user_hand == "p" and computer_hand == "r" or
46       user_hand == "s" and computer_hand == "p"):
47     print("You win")
48 else:
49     print("You lose")
```

Note the particular way we determine the win, tie, or loss in this program. If the two hands are the same, then it's a tie. If the player's hand is Rock and the computer's hand is Scissors, then the player wins. Likewise, if the player's hand is Paper and the computer's hand is Rock, or if the player's hand is Scissors and the computer's hand is Paper, then it's the player's win. *Otherwise*, it is the computer's win. We print out the result, and the program terminates.

We could have written this `if` statement in a number of different ways, but we picked this particular order in this program. What would have been alternative ways? What happens if the player inputs an invalid input, say, "t"? Let's further discuss this in the final section of this lesson.

10.17. Code Review

There's more than one way to skin a cat, as the saying goes. Likewise, there's more than one way to write a program to achieve the same goal.

When you write a program, there are a lot of choices. You end up making a lot of decisions (often even without realizing it). Your program will turn out differently depending on the series of decisions that you make. Sometimes you make "good" decisions, and sometimes you make "bad" decisions. Different people will (more than likely) end up with different programs for the same task.

It's often useful to have an extra pair of eyes to go through your implementations (and, designs). Based on the program that you have written, other people can provide some useful feedback. This is called the "code review".

Let's review together our first rock paper scissors program.

In this program, we defined a function, `to_string`, since (more or less) the same set of statements are used twice, one for printing out the player's hand and the other for printing out the computer's hand. It is generally a good practice to reduce the code duplications.

Functions can be useful even when they are used only once (in a program). Using functions helps modularize the program, and it helps separate the high level structures from the low level details.

In our example, for instance, the part that reads the user input (lines 17-25) could have been written as a separate function. The same with the part that generates the computer hand (lines 29-35) and the final part that determines the winner (lines 42-49). If we had done that, the "main" part of the program might have been simpler and more readable. Here's a (pseudo-code) example of the entire script:

```
# Some function definitions here...  
  
u = user_hand()
```

```

c = computer_hand()

if is_win(u, c):
    print("You win")
else:
    print("You lose")

```

Just to be clear, we are not saying that this alternative way of writing, for example, is "better". "Simple" often has big advantages, and there are always tradeoffs among different alternatives when you write a software. "Better" can only be defined in terms of all the variables that constrain the particular software development project. For our simple rock paper scissors program, for instance, is the modularity important? If so, how much? Is the code readability important? If so, to what extent? And so forth.

We will look at another implementation of the rock paper scissors game that is a bit more modular in [the next part](#). Besides the modularity, this *rps/main.py* program has some additional room for "improvement".



We all have blind spots. One of the most important functions of the code review is to look at the same things from different angles. The readers are encouraged to go through the example code and see if they can find anything "new". For instance, did we use any "hidden assumptions" in writing this program? What will happen if you let a friend play the game using this program? Etc.

One place where we can improve the program is the use of the "special" strings, "r", "p", or "s". As stated, they are arbitrary but they have special significance in the program. If you happen to write R instead of r, for instance, then the program might "break". (That is, it may not function as intended.)

Python has a few ways to alleviate such problems. We will discuss this throughout the rest of the book.

Another area where we can improve this program is the input handling. When a

10.17. Code Review

player inputs an "invalid" input, for example, the player just loses in this program. And, the program does not tell you exactly what happened.

First of all, let's look at the lines 24-25, the `else` clause of the input processing `if` statement. If an invalid input is supplied, we merely set the input to an empty string `"`, and because of the way we have written the final `if` statement, the user who provided an invalid input always loses.

One can easily see this by going through the `if - elif - else` clauses (lines 42-49) with an empty string for `user_hand`. When an invalid hand is inputted by the player, it falls to the final `else`, and hence he/she loses, regardless of the value of `computer_hand`. (Note that `computer_hand` is always valid since we generate it.)

```
42 if user_hand == computer_hand:
43     print("Tie")
44 elif (user_hand == "r" and computer_hand == "s" or
45       user_hand == "p" and computer_hand == "r" or
46       user_hand == "s" and computer_hand == "p"):
47     print("You win")
48 else:
49     print("You lose")
```

When an invalid user input is received, it can be handled in a number of different ways. For example, we can just declare the game as the user's loss and terminate the program (since there is no point of going through the rest of the code). Or, we can give the user a second chance, or a few additional chances, so that they can input a valid hand.

Furthermore, we can be more flexible, or more strict, in accepting the user input. We can add some input validation, and/or we can add some input data "cleansing", etc. For instance, currently if the player types "rrr", this is accepted as a rock (which is internally converted to `"r"`). As another example, if the player types `" rock"` (with a leading space), it fails to read it as a rock. (Did you realize that? 😊)

As stated, these are not always necessary. Whether any further implementation is

necessary is determined by the (implicit and explicit) requirements. It should be noted that *the more is not always "the better"*.

Another "improvement" that we can make to this program (if desired) is that we can let the player play more than one hand. The current implementation merely plays one "round" and terminates. We can alternatively let the player play a fixed number of rounds, or we can let the player play as long as they want. In the latter case, we will need to provide a way for the player to quit the game.

We will continue with this rock paper scissors program in the next two parts, and address some of these "issues".

Chapter 11. Lab 1 - Expressions and Statements

Not all those who wander are lost.

— Bilbo Baggins (The Lord of the Rings)

Reading, and learning from, other people's code is very important. Reading books like this is one way to get more exposure to the code written by (hopefully 😊) more experienced programmers.

Ultimately, however, programming is about doing. The "job" of a programmer is writing programs. Like learning to drive, for instance, the theories and the "second hand driving" go only so far. You yourself will need to get in the driver seat and drive. And practice.

We have a few "lab sessions" in this book so that you can get some "hands-on" experience with programming. They include a number of small coding exercises, mostly based on the main project, e.g., the rock paper scissors game.

You can do all this in the `rps` folder, or you can create a separate folder for each problem. But, it is easier to just use the same folder and use different file names for different scripts. You can reuse the development environment that we set up in the beginning, including `venv` and what not.

11.1. Echo

Write a Python script that reads a user input and prints out the result back to the user.

You can call the script `echo.py`, or any other name that you prefer. You can add the script to your git repository as follows:

```
git add .  
git commit -m "Created a new script echo.py"
```

Then, every time you make changes to the script file, you can do the following to commit changes into the repository, with appropriate commit messages:

```
git commit -am "Updated"
```

11.2. Dice Rolls

Write a Python program that generates a random number from 1 to 6 as if you are rolling a dice. Print out the result in the following format. "The outcome is x." where x is the random number generated.

Add the script file into git. You can do the same for all the scripts in the following exercises.

11.3. Is It Positive?

Write a function that takes an `int` argument and returns a `bool` depending on whether the given integer is bigger than zero or not. What would be a good name for a function like this?

Write a script that tests this function for the numbers, `-10`, `0`, `5`, and `20`. Print out the appropriate text to indicate whether a given number is positive or not.

11.4. To Uppercase

Write a function that takes a `str` argument and converts it into all uppercase letters. For this, you may have to search for an appropriate method defined in the `str` type. How would you go about doing that?

11.5. Random Letters

Write a program that gets a user input, converts it to uppercase using this function, and prints out the result to the console.

11.5. Random Letters

Write a function that picks two different random alphabets from the 26 lowercase letters. Write a script that uses/tests this function.

Python has builtin functions like `ord` and `chr`, which can be useful in problems like this.

11.6. Random Arithmetic

Write a function that

- Takes two `int` numbers,
- Picks a random operator, e.g., one of `+`, `-`, `*`, and `//`,
- Computes the expression comprising this operator and the two numbers, and
- Prints out the result in an expression form.

For example, given two arguments, `5` and `10`, and a randomly picked operator, `+`, the function will print out

```
5 + 10 = 15
```

Write a script that calls this function 5 times, with 5 different pairs of integers.

11.7. Can I Buy a Vowel?

Write a function that takes a single letter (English alphabet), and returns true if the given letter is a vowel. Otherwise it returns false.

Write a script that generates a random English alphabet and tests it whether it is a vowel or not. If it is a vowel, then print out the letter and **True**. If not, then print out the letter and **False**.

11.8. All True or Not

Write a function that does the following:

- It generates three random bool values, **True** or **False**.
- If all three values are the same, e.g., 3 **True**'s or 3 **False**'s, then it prints out "All true" or "All false" depending on their values. Otherwise, it prints out the number of True's and the number of False's, e.g., "2 true and 1 false", etc.

Write a script that calls this function 10 times.

11.9. Spade, Heart, Diamond, or Clubs

Write a program that asks the user for one of the four strings, "Spade", "Heart", "Diamond", or "Clubs". Read the user input, and based on the first letter of the input string, choose one of the four letters, "s", "h", "d", or "c".

Print out both the user input and the selected letter.

11.10. Random Suit

Write a program that randomly picks one of the four letters, "s", "h", "d", or "c".

11.11. The Same Suit Or Not

Use the code from the previous two exercises, and write a program that does the following:

- Read the user input, and convert it to one of the four letters. Assign it to a name.

11.12. Rock Paper Scissors

- Generate a random letter, out of the four, and assign it to another name.
- Compare the letters referred to by the two names. If the two letter are the same, then print out "The same suit. You win!". Otherwise, print out "Different suits. Try again.".

11.12. Rock Paper Scissors

Close the book. (That is, after reading this problem 😊)

Write a rock paper scissors program based on what you remember from the book. You can use more functions than we used in the sample program.

The main part of the program has three parts:

- Read a user input and convert it into **r**, **p**, or **s**.
- Randomly generate one of the strings, **r**, **p**, or **s**, as a computer hand.
- Compare the two hands and decide who wins.

Try running your program a few times, and make sure that it works as expected.

Procedural Programming

What are we holding on to, Sam?

That there's some good in this world, Mr. Frodo... and it's worth fighting for.

— Frodo and Sam (The Lord of the Rings)

In the lessons of the previous part, we studied one of the simplest implementations of the rock paper scissors game. We went through a lot of details digging into the example program. At the end of the day, however, the more important thing is the high level flow of the program. You read a user input, you generate a random hand, and then you compare them to decide who wins.

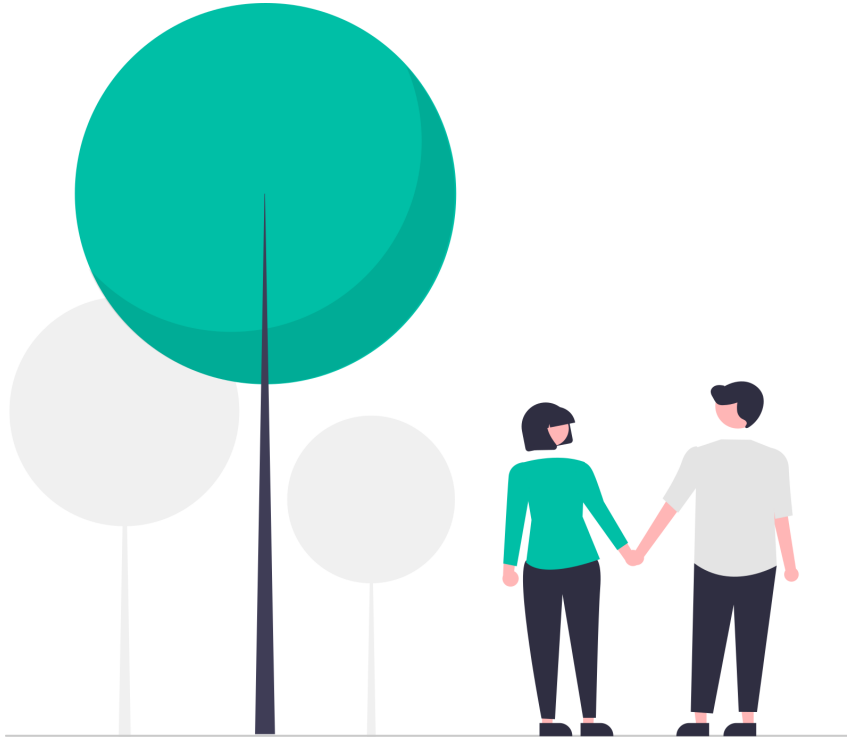
This kind of linear thinking is at the heart of the imperative programming. Virtually every program can be written this way regardless of whether that's the best way to solve a given problem.

Now, *can you write a program like that?* If you can, then you are ready for the new challenge. 😊

This part is titled "Procedural Programming". Procedural programming is an imperative programming style that uses procedures, or functions. These two terms, imperative and procedural programming, are pretty much synonymous, however.

As in the previous part, we will also start with a complete program, namely, the second version of the rock paper scissors game, and we will analyze each and every part of the program in the following lessons. This example program happens to be much longer and more complex, and it will take some effort to follow along. We recommend that you try to memorize some portions of the program while reading this book so that you don't have to flip back and forth too much. 😊

Chapter 12. Rock Paper Scissors - The Sequel



There are many different ways to do the same thing, and programming is no different. We can, in fact, write our rock paper scissors program rather differently. As stated earlier, for instance, we can make the program "more modular".

We can also improve on error handling. In this lesson, we will add an error handler when the user input is not a valid string, e.g., one of rock, paper, or scissors. While we are at it, let's try and improve the output as well. Improving input and output is the easiest way to make your program more "user friendly", so to speak.



We have the same setup as in the previous part, but in a different project folder named `rps2`.

Our second implementation of the rock paper scissors game is split into three scripts (or, modules), *game.py*, *rock_paper_scissors.py*, and *output.py*. The name of the "top level" folder (e.g., *rps2* in this example) does not play any significant role in Python.

The script file name, on the other hand, can play an important role, depending on the purpose/usage of the script file. The names of the *subfolders* that include other Python source code files are also significant, as we will see later in the book.



Knowledge is familiarity. This code sample might look strange and gibberish to you at first, depending on your background, but eventually they will all look trivial, and obvious, to you. Clearly, that does not happen automatically. You will need to make *a conscious effort* to "read" the code every time you see the code. We will go through this sample code multiple times throughout this part, and although it may be a bit inconvenient to refer to the code again and again (especially if you are using ebook), do not just flip the pages back and forth. Every time, try to "read" the (relevant part of) the code.

Now, let's start reading the code.

rps2/game.py

```
1 from typing import Final
2 from output import *
3 from rock_paper_scissors import play_a_round, WIN, LOSS
4
5 MAX_ROUNDS: Final[int] = 3
6
7
8 def play(max_rounds: int = MAX_ROUNDS):
9     """Play max_rounds of Rock Paper Scissors."""
10
11     start_banner()
12
13     wins: int = 0
```

```

14     for _ in range(max_rounds):
15         wlt = play_a_round()
16         print_result(wlt)
17         wins += 1 if wlt == WIN else 0
18         horz_bar()
19
20     end_banner(wins, max_rounds)
21
22
23 def print_result(wlt: str):
24     """Print a text corresponding to win/loss/tie."""
25
26     if wlt == WIN:
27         print("You win!")
28     elif wlt == LOSS:
29         print("You lose!")
30     else:
31         print("Tie!")
32
33
34 if __name__ == "__main__":
35     play()

```

rps2/rock_paper_scissors.py

```

1 import sys
2 import random
3
4 _R, _P, _S = 'r', 'p', 's'
5 WIN, LOSS, TIE = 'w', 'l', 't'
6
7
8 def _user_hand() -> str:
9     while True:
10         try:
11             u = input("Rock (r), Paper (p), or Scissors (s)? ").strip()
12             return u[0].lower()

```

```

13         except (EOFError, KeyboardInterrupt):
14             print("\nThanks for playing the game!")
15             sys.exit()
16         except:
17             print(f"Invalid input, {u}. Try again")
18             continue
19
20
21 def _computer_hand() -> str:
22     return random.choice((_R, _P, _S))
23
24
25 def _determine_win(u: str, c: str) -> str:
26     if c == u:
27         return TIE
28     if u == _R and c == _S or u == _P and c == _R or u == _S and c ==
    _P:
29         return WIN
30     return LOSS
31
32
33 def play_a_round() -> str:
34     u = _user_hand()
35     c = _computer_hand()
36     print(f"You -- {_to_str(u)} vs {_to_str(c)} -- Computer")
37
38     return _determine_win(u, c)
39
40
41 def _to_str(h: str) -> str:
42     if h == _R:
43         return "Rock"
44     elif h == _P:
45         return "Paper"
46     elif h == _S:
47         return "Scissors"
48     else:
49         return ""

```

```

50
51
52 if __name__ == "__main__":
53     play_a_round()

```

rps2/output.py

```

1 def horz_bar():
2     print("+" + "-+" * 20)
3
4
5 def start_banner():
6     horz_bar()
7     print("Let's play Rock Paper Scissors!")
8     horz_bar()
9
10
11 def end_banner(wins: int = 0, rounds: int = 0):
12     print("Thanks for playing Rock Paper Scissors!")
13     if rounds > 0:
14         print(f">>> You won {wins} rounds out of {rounds}. <<<")
15     horz_bar()

```

The "main script" in this example (which is context dependent, if you will) is *rps2/game.py*, which imports the `rock_paper_scissors` and `output` modules. As stated, `game` can also be used as a *module*, and *rock_paper_scissors.py* can also be used as a runnable script, etc.

The *output.py* file does not include any "real statements" (e.g., with side effects), other than the function definitions, and the import statement, etc., and hence it will produce no real meaningful result when we run it as a script.

This new program looks rather different from the *rps/main.py* in the previous lesson. But, they are more or less the same program. They do more or less the same thing.

One thing to note is that, unlike the spoken languages, which are "one dimensional",

the programming languages have "two dimensional" structures (e.g., nesting and what not). Computer programs are "visual". You do not read a program like reading a novel, for instance. You do not read a program word by word sequentially, from left to right, and then from top to bottom. You read programs "visually". This is especially true for the Python programs. Do you notice the different levels of indentations, and the empty lines, etc. in the program *game.py*, for instance? (It will take a bit of practice for the beginning programmers. ☺)



Not all programs (or, a single code file) will fit into a single screen, and you will need to scroll up and down and "sequentially" read the program. But even then, you always try to see the entire program visually (at least, in your mind ☺).

Just by looking at the script, *rps2/game.py*, we can see that it has three "parts" (roughly speaking, excluding the `import` statements). The first part (line 5) corresponds to a variable declaration (or, a name binding).

The second part (lines 8-31) comprises two compound statements, namely, the function definitions, `play` and `print_result`. The `play` function, for instance, in turn comprises what appears to be multiple "parts". The middle part includes another compound statement starting with the keyword `for`. Although we have never seen the `for` statement before (in this book), we can easily see that it is a single compound statement (lines 14-18), e.g., based on the indentations, etc.

These two functions, `play` and `print_result`, also include, strangely ☺, stand-alone string expressions (e.g., triple quote strings) in lines 9 and 24. This is strange because, as we learned earlier, an expression that is used as a statement by itself is evaluated, but its result is ignored by the Python interpreter (in the non-interactive mode). So, what are these long string expressions doing here? What is the point? ☺

Finally, the program includes another segment that is an `if` compound statement (lines 34-35). If the `if` condition turns out to be `True` (in some way), then we call the `play()` function that we have just defined. Otherwise, nothing happens.

The `play_a_round` function, used in the `for - in` statement inside the `play`

function (line 15), is defined in a different file, *rock_paper_scissors.py*. On the other hand, the rest of the functions used in the main script, `start_banner`, `end_banner`, and `horz_bar`, are defined in another file, *output.py*.

Note the wildcard `from import` syntax (line 2). The declaration effectively imports all names defined in the `output` module, which comprises 3 function definitions. The `rock_paper_scissors` module includes a few variable definitions, function definitions, and finally another `if` statement (lines 52-53) that resembles the one in the main script. The `if` clause of this statement also includes the same Boolean expression `__name__ == "__main__"`.

As with *rps2/game.py*, we can also see that the script, *rps2/rock_paper_scissors.py*, has more or less three "parts" simply by looking at code, and its overall structure. The first part (lines 4-5) corresponds to name bindings, or variable definitions.

The second part (lines 8-49), which comprises the bulk of the script, includes a number of function definitions. Each function `def` in turn comprises one or more "parts". The `_user_hand` function, for instance, includes another compound statement starting with the keyword `while` (lines 9-18), which again we have not seen before in this book. But, you can easily guess that it is a single compound statement, again, based on the indentations, and so forth. Inside the `while` statement, there appears to be another structure (lines 10-18). The same with other function definitions.

We have made a few changes in this new rock paper scissors program relative to [our first version](#).

First of all, we now print out "banners" in the beginning and end of the program run (e.g., `start_banner` and `end_banner` functions in *output.py*). This is mostly "cosmetic", which will (likely) make the user experience (slightly) better. 😊

Second, we now use constant variables for the internal symbols, `"r"`, `"p"`, and `"s"` (line 4 in *rock_paper_scissors.py*). As stated, this kind of change can reduce the chance of any accidental programming errors.

Also, we now give the player multiple chances to provide a valid input (the `_user_hand` function in `rock_paper_scissors.py`), using a `while` loop. In addition, this function includes some "error handling" using Python's `try - except` statement. (Again, pay attention to the indentations.)

In the `_computer_hand` function (lines 21-22), we use a different function `choice` from the same `random` module. This function takes an argument of a sequence type, e.g., a `tuple` in this example, line 22. The function `random.choice` is used basically for the same purpose as before, namely, to generate a random hand for the computer player.

The `play_a_round` function essentially includes all the main functionalities of the original rock paper scissors program from the previous part. It reads the user's hand using the `_user_hand` function (line 34), it generates the computer's hand using the `_computer_hand` function (line 35), and it prints out the hands using the `print` function and the f-string expression (line 36). Then, the outcome is determined using two `if` statements in the `_determine_win` function (lines 25-30).

Let's try running this program (e.g., the main script `game.py`). As we have seen before, there are multiple ways to run a Python script, for example, using `-c` or `-i` flags. Or, by `importing` the script module within the interactive shell.

The most common way is, however, to use the script file name in the command line, as we have been mostly doing so far.

```
$ python game.py
+-----+
Let's play Rock Paper Scissors!
+-----+
Rock (r), Paper (p), or Scissors (s)? rock
You -- Rock vs Paper -- Computer
You lose!
...
+-----+
Thanks for playing Rock Paper Scissors!
```

```
+---+---+---+---+---+---+---+---+---+---+
$
```

There is an even "better" way (although we have noted that there is no such thing as "better", in the abstract scale, in programming. ☺). Every Python script is a "module", as we discuss in this lesson, and throughout this book. We can run a Python script, or a Python module, as follows:

```
$ python -m game
+---+---+---+---+---+---+---+---+---+---+
Let's play Rock Paper Scissors!
+---+---+---+---+---+---+---+---+---+---+
Rock (r), Paper (p), or Scissors (s)? r
You -- Rock vs Scissors -- Computer
You win!
...
+---+---+---+---+---+---+---+---+---+---+
Thanks for playing Rock Paper Scissors!
+---+---+---+---+---+---+---+---+---+---+
$
```

Note that we use the `-m` flag followed by the module name "game" (not the file name), as we have done a few times before.

There are not much differences between the two styles, `python <mod>.py` and `python -m <mod>`, but there are still some (subtle) differences that are significant enough (in the author's view). This is especially true if your "program" contains more than one source files, in different subfolders, etc.

We recommend that you use the latter syntax while developing your multi-file Python programs (unless you are working on simple (throwaway) programs). In this book, we will mostly use this syntax using the `-m` flag for the (*runnable and importable*) modules, moving forward, and the command line argument syntax for the (*non-importable*) scripts. We will discuss this further a bit later.

As we will see in the next and final lesson, the common practice is to create *Python programs* as (sharable) modules or packages unless they are (throw-away) scripts with limited uses, or unless they are just single file scripts used for specific purposes.

In the `play` function of the main script *game.py* (lines 8-20), we repeat the game play, `play_a_round()`, 3 times (lines 14-18). We will go over the relevant Python grammar next, including the `for` and `while` statements.

12.1. Python Modules

A module is a file containing Python definitions and other statements, (some of) which can be used in other modules and scripts (through Python's import system).

As stated, as far as Python is concerned, a single Python file is *both a script and a module*. It is the developer's decision how they are going to use their Python program file, that is, whether to use it as a (runnable) script only, as an (importable) module only, or as both. Although it is not an irreversible decision and it does not require a "lifetime commitment" ☺, how you develop a Python module/script depends on how you are going to use it.

Although the Python grammar does not clearly distinguish it, there are two kinds of statements in Python. One kind of statements are mainly used to define variables, functions, and classes (that we will discuss later), etc., which are primarily for the Python interpreter (so that it can execute other statements).

The other kind of statements are those that have side effects or that "do something". In some sense, these are the meat of a Python program. ☺ This distinction is not black and white, however.

Here's a general guideline for the beginning Python programmers. (As you gain more experience, and develop your own insight, you will start to have your own opinions. ☺)

When you create a module, that is, when you write a program that will only be used as a module, you primarily include definitions so that they can be used in other

12.1. Python Modules

modules and scripts. (Let's call this a "module module".)

The names that are *intended* to be shared with other programs start with non-underscore letters (e.g., English alphabets). All other "hidden" names should start with underscores (_). This is because Python's wildcard import syntax `from <mod> import *` does not import names that start with underscores by default.

It should be noted, however, that Python does not have a module-level access control. All names in a module can be used in other programs if the module can be located and it can be installed/imported. That is, there is no "public" vs "private" names in the Python modules.

In a "module module", any executable, non-definition statements should be used for the "initialization" purposes only. As stated, a module can be `imported` more than once in a program, and these statements in the same module are run for the first time `import` only. In fact, we generally recommend not to use this kind of statements in a "module module". It is a relatively rare use case that a module requires any specific initialization steps during an import.

Note that this kind of modules should have unique, descriptive, reasonably long, and snake-case module names, to avoid the name collision. The best practice is, however, to create packages, or "package modules", for this kind of modules, as we will discuss shortly. (As stated, the term "package", as well as many other terms we use in programming, can refer to different things depending on the context.)



In Python, *modules* and *classes*, as we will see later, have some overlapping use cases when they are used for the code organization purposes. In many circumstances, it is generally better to use classes, and class instances, rather than modules when they require any initializations.

The second kind of Python program files are the ones that are to be used only as executable scripts. (Let's call them "script modules".) In this kind of modules, or scripts, we primarily use executable statements to "do something". All other definition statements are included in a script so that they can be used in other

statements in the script. In some cases, it may be useful to put these definitions in separate modules or packages. But, that is not usually necessary.

For this kind of script modules, following the standard naming conventions is not that important. In particular, you do not have to use the underscore-prefix name rule since, in effect, everything is "private" in the script modules. (Nonetheless, it is still a good practice to follow the snake-case and Pascal-case naming conventions, for consistency.)

The same with the module/script names (e.g., file names). Any name will do. However, it is often a good practice to use the common file name conventions on a specific platform. On Unix-like systems, for example, the "kebab case" names (e.g., words separated by dashes, as in "change-all-names.sh") are often used for files. As with the variable names in a program, the scripts that are to be used multiple times, over a longer time period, or to be shared with other people should use longer, more descriptive names whereas shorter, non-descriptive, names suffice for the scripts that have a shorter life span (e.g., for one time use).

Now there are this third kind of modules that are *intended* to be used both as an importable module and as a runnable script. (Let's call this a "combo module" in this book. *Would you like a French fries with that?* ☺) The combo modules should follow the same rules/conventions as those of the module modules. As stated, the initialization statements should be sparingly used, if any. In addition, the script statements should be "guarded" inside the idiomatic `if __name__ == "__main__"` statement.

As stated, every (importable) module has a module name, and its name is, by default, the name of the script file (excluding the file extension). When a module is used as a script (e.g., not imported as a module), however, the (internal) module name is always `__main__`.

Everything in Python is an object. Modules are also objects in Python. Modules have attributes and other methods. One of the module's predefined attributes is `__name__`, which is initialized with its module name (e.g., by the Python interpreter). Hence, this Boolean expression `__name__ == "__main__"` will evaluate to `True`

12.1. Python Modules

only if the module is run as a "script" (e.g., as opposed to being imported via the `import` statement).

Hence, for the combo modules, it is idiomatic to put all executable script statements, which are not for the module initialization, within this kind of conditional statement. In many cases, although it is not entirely necessary, we sometimes use this idiom even for the "script modules".

As an example, let's suppose that we have a python module named "my_math.py":

rps2/my_math.py

```
def add(a: int, b: int) -> int:
    return a + b

if __name__ == "__main__":
    a, b = 1, 2
    sum = add(a, b)
    print(f"sum = {sum}")
```

If we run this module/script, we will get the following output:

```
$ python -m my_math
sum = 3
```

On the other hand, if we import this module in another script/module, then we can use the `add` function in that module. For instance,

rps2/add-driver.py

```
import my_math

if __name__ == "__main__":
    a, b = 1, 2
    sum = my_math.add(a, b)
```

```
print(f"sum = {sum}")
```

```
$ python add-driver.py
sum = 3
```

The `if __name__ == "__main__"` guard is not necessary in this case, and we could have just put these three statements at the top-level without the `if`. (Note that the name `add-driver` is not a valid module name in Python since it includes the characters (e.g., a dash `-`) that are not allowed in the identifiers.) In practice, for Python files that are only to be used as scripts do not generally use this `if` statement. For example,

rps2/add-driver-2.py

```
import my_math

a, b = 1, 2
sum = my_math.add(a, b)
print(f"sum = {sum}")
```

The `rps2/game.py` file, in our (second version of) rock paper scissors app, includes this idiomatic `if` statement, lines 34-35.

```
34 if __name__ == "__main__":
35     play()
```

This particular `if` statement comprises only one statement, `play()`. The `play` function is also "exported", so to speak, since its name does not start with an underscore. This `game` module can be imported, possibly (locally) in other modules/scripts, and the `play` function can be used in those modules/scripts.

The `rps2/rock_paper_scissors` module also includes this idiomatic guard statement, lines 52-53. This `if` statement again happens to include only one

statement, which is a function call, `play_a_round()`. The `play_a_round` function (and, other functions and variables) can be imported from other modules/scripts as well.

Both of these files can be run as scripts. These files can also be used as *importable* modules without any undesirable side effects.

In general, a Python "program" might comprise more than one (local) files and subfolders, and some files may be intended to be used solely as scripts, some files may be intended to be used solely as modules, and some other files may be intended to be used for both purposes.

12.2. Python Packages

For the modules that are to be shared, it is a good practice to use packages. The packages add additional namespaces, and their use thereby reduces the chance of module name collisions.

A package in Python is a folder that contains one or more other Python modules (e.g., Python source code files) and/or other packages (e.g., other folders).

The packages are a special kind of modules, and they can be nested. When you refer to a module (or submodule) in a package, we use the "dotted module name" syntax. All folder names in a path (its parent and all ancestors), up to the package folder, are prepended, separated by dots (`.`). For example, a module `airplane` (corresponding to a file `space/airplane.py`) within a package `space` (corresponding to a directory `space`) may be referred to as `space.airplane`. (Well, airplanes do not fly into space, but that's beside the point. 😊)

The submodules of a package, e.g., the modules within the package, can be imported as `import <full-mod-path>` as before, e.g., `import space.airplane`, or it can be imported as `from <parent-pkg> import <mod-name>`, e.g., `from space import airplane`. These two syntaxes are equivalent. As indicated, packages can be nested, to an arbitrary depth. However, it is relatively rare to see the packages that are more than two levels deep.

Note, again, the flexibility of the Python rules. It is up to you, the programmer, to decide whether to use a Python file as a (standalone) module or as part of a package. Any Python file must reside in a certain folder on a file system (which may be under a certain other folder, etc.), and hence you can always use it as part of a certain package (corresponding to one of the ancestor folders in the path).

As with the module vs script comparison, we generally pick one convention (e.g., one top-level package folder, if any) and stick to that particular choice for a particular project. (Note that there can be subtle differences in terms of the import path, etc., and hence changing from one usage to another may require some code changes, as we discuss below.)

It used to be the case that a package required a special file named `__init__.py` in the package folder (and, its subpackage folders, etc.) *even if it is empty*. That is no longer the case, however.

A package is an object (just like everything else in Python), and you can customize its properties using certain (predefined) attributes, etc., in this `__init__.py` file, but we will not discuss this in this book. We recommend that the Python beginners stick to a certain (simple) convention(s) and do not go astray too much beyond the "modern standard convention": A Python file corresponds to a (regular) module/script, and a folder corresponds to a package module. No special configuration is needed, in most cases.

As stated before, it is a common practice to make any module that is to be (widely) shared a package module because packages provide nested namespaces. Package names are generally short, and they do not tend to include any underscores (or, dashes or any other characters that are not allowed in Python identifiers).

In our example of the rock paper scissors 2, we have three modules (that are more or less "independent" of each other, in terms of the package structure), `game`, `output`, and `rock_paper_scissors` in a single folder. We can alternatively treat them as submodules of a single package. How do we do it?

We just treat the parent folder of these three modules (`rps2` in this example) as a

12.2. Python Packages

package. For instance, we just `cd` to one folder up, and run the package module as follows:

```
$ pwd
/home/harry/projects/rps2
$ cd ..
$ python -m rps2.game
```

Note the module name. It is `rps2.game`, and not just `game`. Now `rps2` is a package, and `rps2.game` is a submodule of this package.

In order for this to work, however, the `import` statements need to be modified. For instance, instead of

```
2 from output import *
3 from rock_paper_scissors import play_a_round
```

in `game.py`, lines 2-3, we will need to use the correct package-based module names, as follows:

```
from rps2.output import *
from rps2.rock_paper_scissors import play_a_round
```

This syntax of using the (full) dotted module names is known as the "absolute import". The import statements are different depending on whether we treat `rps2` as a package or not. In the modules that are part of the same package, we can also use the "relative import" syntax. For example,

```
from .output import *
from .rock_paper_scissors import play_a_round
```

The dot `.`, e.g., in `.output`, represents the current package/folder. The two dots `..` in this syntax represents the parent package/folder, etc. It should be noted, however, that since the name of the main script module is always `"__main__"`, the modules intended for use as (runnable) script modules, even within packages, cannot use the relative import syntax. (They will work if the module is used as an imported module, but they will not if it is run as a script.)

12.3. Tuple Unpacking

Let's take a look at the assignment statement, line 4, in the `rock_paper_scissors` module.

```
4 _R, _P, _S = 'r', 'p', 's'
```

As we learned in the beginning, the right hand side of this statement is an *expression list*. It includes three expressions, separated by commas, each of which is (trivially) evaluated to its own (string) value, `'r'`, `'p'`, and `'s'`. The value of an expression list is a tuple. In this case, it is `('r', 'p', 's')`. Now we are assigning this (single) tuple value to three variables on the left hand side, `_R`, `_P`, and `_S`. This is called the "unpacking" in Python (or, "destructuring" or "deconstructing", etc.).

This statement is (more or less) equivalent to the following:

```
_R, _P, _S = ('r', 'p', 's')
```

or

```
temp = 'r', 'p', 's'  
_R = temp[0]  
_P = temp[1]  
_S = temp[2]
```

12.3. Tuple Unpacking

As long as the tuple has the same number of items as the number of the variables on the left hand side, this (one-liner) unpacking will succeed (which is slightly more concise and easier to read).

After executing this statement, line 4, the variables `_R`, `_P`, and `_S` will be bound to `'r'`, `'p'`, and `'s'`, respectively. The same can be said about the three variables, `WIN`, `LOSS`, and `TIE`, which are bound to the three objects, `'w'`, `'l'`, and `'t'`, respectively. (Line 5.)

Unpacking, or sequence unpacking, generally applies to all sequence types such as lists and strings. For example, for a list,

```
a, b, c = [1, 2, 3]
```

This statement assigns values `1`, `2`, and `3` to the variables `a`, `b`, and `c`, respectively. And, for a string,

```
a, b, c = "joy"
```

What would be the values of `a`, `b`, and `c` after executing this statement?

One thing to note is that the type annotation becomes a bit cumbersome in the "multiple assignment" statements.

For example, we added an annotation `Final[int]` for the name `MAX_ROUNDS` in line 5 of `rps2/game.py`.

```
MAX_ROUNDS: Final[int] = 3
```

This indicates that the variable `MAX_ROUNDS` is of the type `int` and it is not intended to be changed throughout the execution of the program. Other programming languages have constructs like `final`, `const`, or `readonly`, etc. In Python, it is

primarily a convention. When a program, or a programmer, uses all caps names, they are intended to be "constants". The type annotations provide additional clues to other third party tools.

The variables `_R`, `_P`, and `_S` in `rps2/rock_paper_scissors.py` are of the `str` type and they are also semantically constants. Hence they can be annotated as follows:

```
_R: Final[str]
_P: Final[str]
_S: Final[str]
_R, _P, _S = 'r', 'p', 's'
```

Note that, in this case, we had to write these variables separately before their use, just for the purposes of type annotations. (As noted, variables cannot be declared like this without the initial values in Python. The type annotations are exceptions, as we saw in the previous part.) In many cases, this kind of type annotations make the code less readable in practice, and hence it is not recommended. (And, it beats the purpose of using the concise syntax of "multi variable assignment".)

If the type hint is really important, then we can just use three separate statements in cases like this. For example,

```
_R: Final[str] = 'r'
_P: Final[str] = 'p'
_S: Final[str] = 's'
```

As we will see in the next lesson, this type of constants can be "better" represented using Python's `Enum` types.



The type annotations for variables may not always be needed, e.g., in contrast with the type annotations for functions. In many cases, the type of a variable can be easily inferred based on its initial value. This is known as the "type inference". If a variable is initially

bound to a string value, then the variable *might* be of the `str` type, or something broader than `str`.

The typing support in Python is currently evolving, just like other aspects of the language, and we may end up with a somewhat different typing system in the future. One thing worthwhile to repeat and emphasize here is that the typing puts more constraints to the usage of the Python language. For instance, in Python, a type is associated with an object. Variables are just references or aliases to objects. A variable which refers to an object of one type at one point can be made to refer to another object of a different type. That is perfectly valid in Python. In the "type annotated world", however, we tend to use a variable to refer to the objects of the same type.

We have seen before some similar constraints that the typing system puts on the Python programming style. For example, typing on lists encourages the use of the "homogenous lists".

12.4. Function Definitions

We learned how to define a simple function in the lessons of the previous part. We will discuss a little bit more on the function definitions in this lesson.

In Python, a function can be defined with zero, one, or more parameters, more or less in a certain order. For some parameters, when the function is called, the corresponding arguments need to be provided in the exactly the same positions as they are defined in the parameter list (e.g., as in C and some other C-style languages). These are called the "positional only parameters" in Python.

For some parameters, they do not have the fixed positions in the parameter list, and the corresponding arguments need to be provided using the `parameter=value` syntax (known as the "keyword arguments"). These are called the "keyword only parameters".

For all other parameters, they have fixed positions and they can be used either with the positional argument syntax (in the corresponding positions) or with the keyword argument syntax. When you define a function with non-zero parameters, they belong to this category by default.

In order to define positional only parameters, we use a separator `/` (forward slash). Any parameter preceding this optional separator is positional-only. To define keyword only parameters, on the other hand, we use a separator `*` (asterisk). The parameters following this optional separator, if any, are keyword-only. If both are present, `/` should come before `*`.

Here's an example:

```
def are_you_being(name: str, /, title: str, *, repressed: bool) -> bool:
    """Do you get the Monty Python joke? :)"
    return True
```

This function can be called in one of the following two ways:

```
ans1 = are_you_being("Anonymous", "Peasant", repressed = True)
ans2 = are_you_being("Arthur", title = "Knight", repressed = False)
```

As stated, most parameters are, by default, both positional and keyword, like the `title` parameter in this example. The parameter `name` is, however, positional only since it comes before `/`, and because it is defined in the first position in the parameter list, it can only be used in the first position in the function call. On the other hand, `repressed` is a keyword only parameter since it is placed after `*`, and it can only be used as a keyword argument, as illustrated in the example here.

Note that, when you call a function, regardless of which categories its parameters belong to, keyword arguments cannot come before any positional arguments. The order of the keyword arguments (in the argument list) is not important in a function call (as long as they come after all positional arguments), but they must match all

12.4. Function Definitions

non-optional keyword parameters, e.g., one for one, as defined in the function. Likewise, all *non-optional* positional arguments must be provided in their corresponding positions, that is, before any keyword arguments.



It all sounds way too complicated, but there are only two rules. When you define a function, the order of the parameters are (i) any optional positional parameters followed by `/`, (ii) the normal (positional or keyword) parameters, and (iii) any optional keyword-only parameters after `*`. When you call a function, the order is (a) any positional arguments and then (b) any keyword arguments.

Note that we use the terms "parameters" and "arguments" slightly differently here. Can you tell the difference from the contexts in which they are used? If not, no worries. These two terms are mostly interchangeable unless you are really picky. 😊

Python does not support "function overloading". That is, functions with the same name (regardless of the differences in the parameter list) cannot be defined more than once in the same scope. However, some function parameters may be made optional in Python by providing their "default values". Hence, the same function may be called possibly with different sets of arguments.

Here's an example:

```
def are_you_learning(subject: str = "programming", using_books: bool =
False) -> bool:
    pass
```

The `using_books` parameter has the default value, `False`. That is, if the value is not provided in a function call, the value is set to `False`. Likewise, if the value of `subject` is not provided, then its value is set to the string `"programming"`.

Since both parameters, `subject` and `using_books`, can be used as positional or keyword arguments and since they both have default argument values, this function

can be called in one of the following seven ways:

```
ans1 = are_you_learning()                                ①
ans2 = are_you_learning("design")                        ②
ans2 = are_you_learning(subject = "design")
ans3 = are_you_learning(using_books = True)             ③
ans4 = are_you_learning("web development", True)        ④
ans4 = are_you_learning("web development", using_books = True)
ans4 = are_you_learning(subject = "web development", using_books = True)
```

- ① This call is equivalent to `are_you_learning("programming", False)`.
- ② The calls of this line and the following line are equivalent to `are_you_learning("design", False)`.
- ③ This call is equivalent to `are_you_learning("programming", True)`.
- ④ The calls of this line and the following two lines are all equivalent to one another.

In our rock paper scissors program, there are two functions that use the optional parameters, `game.play` and `output.end_banner`.

```
8 def play(max_rounds = MAX_ROUNDS):
9     ...
```

```
11 def end_banner(wins = 0, rounds = 0):
12     ...
```

In the case of `play` in the `game` module, for example, the function can be called with one argument, e.g., `play(5)`, in which case we play the specified number of rounds, e.g., 5 times. If this function is called without an argument, e.g., `play()`, then the default value, `game.MAX_ROUNDS`, is used, which is currently hard-coded to 3.

One thing to note is that, unlike in many C-style languages which support optional parameters, the default value of an optional parameter, which is an expression, is

evaluated in Python when the function definition statement is executed, and not every time the function is called. This can have some interesting implications, especially when the default value argument is of a mutable type. This topic is, however, beyond the scope of this book.



Python supports functions with a variable number of parameters, generally known as the "vararg" functions. Python uses the syntax `*args` and `**kwargs` for the "var-positional" and "var-keyword" parameters, respectively. Again, the parameter names are conventional. It's the `*` and `**` prefixes in the parameter names that make them special. For more information, please refer to the official Python reference.

12.5. Function `def` with Type Annotations

There are two kinds of functions in Python, that is, if you have been paying attention ☺, those that return a value and those that do not (or, that return `None`).

All five functions, `_user_hand`, `_computer_hand`, `_determine_win`, `play_a_round`, and `_to_str`, in the `rock_paper_scissors` module happen to return values of the `str` type.

On the other hand, none of the functions in the `game` and `output` modules returns a (non-`None`) value. When a function does not return a value, we can omit the type annotations for the return value. That is, `def f(): pass` is the same as `def f() → None: pass` as far as the typing goes.

Functions in Python do not have to return a value of a single fixed type. In fact, the same function may return a value in certain cases and may not return any values in certain other cases. This may come as a surprise to the readers who have some exposure to the statically and strongly typed programming languages, but that is how it works in most dynamically and loosely typed languages such as Python and Javascript. (And, PHP, Ruby, Perl, Clojure, ...)

In this book, we use the type annotations as a way to put some constraints on Python's "enormous freedom". Note that this freedom, or power, if you will, comes at a price. That is, Python programs tend to have more bugs, and they are harder to maintain over time. It is *generally* more difficult to write a large software system using Python (or, using other dynamically and loosely typed languages).



Python typing does support what is called the "union types", as well as the generic `Optional` type, etc., which can be used in this context. In general, however, the author does not recommend the readers to use the *heavy* typing constructs like generics. Unfortunately, they completely destroy the simplicity and beauty of Python. If you really need the full benefit of static typing, then just use a statically typed language (like Go or Rust) instead of mutilating Python. 😊

The following function, as written, does not have any real implementation, but it does return a string value (always an empty string `""`), which is consistent with the type annotations.

```
def _determine_win(u: str, c: str) -> str:
    return ""
```



As indicated a couple of times before, the Python interpreter does not care about the type annotations. The way we use them in this book, they are solely for us, programmers, sort of as a mental note.

The type annotations for functions impose certain constraints (for us). We are indicating that, for instance, we intend to accept two `str` values and return a `str` value from this function no matter what, through this function signature, `_determine_win(u: str, c: str) -> str`. It makes (large) Python programs generally easier to read (as long as we stick to our promises, expressed as the type annotations). And, eventually, if you decide to do so, you can use other tools to more strictly impose the typing constraints during the development.

12.6. Expression Statements

The `_determine_win` function in `rps2/rock_paper_scissors.py` assumes that it is called with two `str` arguments, and it, in turn, returns a `str` value (and, not any other types, including `None`). We can easily see this from its implementation:

```
25 def _determine_win(u: str, c: str) -> str:
26     if c == u:                                ①
27         return TIE                            ②
28     if u == _R and c == _S or u == _P and c == _R or u == _S and c ==
    _P:
29         return WIN                            ③
30     return LOSS                              ④
```

- ① This is (supposed to be) a string comparison. A "law-abiding citizen" should call this function with two string arguments since that is what the function signature says. Note that this is strictly an honor system (unless we use an extra type checking tool).
- ② Since `TIE` is defined to be a string (line 5), it return a `str` if `c == u`.
- ③ Likewise, it return a `str` if the Boolean condition holds in this `if` statement.
- ④ In all other cases, it still return a `str`.

12.6. Expression Statements

As mentioned earlier (too many times ☺), in Python, an expression can be used as a statement by itself. In fact, *any* expression (or, even an expression list) written in a line by itself is syntactically a statement. (People coming from the C-style or other imperative languages might find this surprising, in which only a few expression types such as function calls can be used as statements. But, this is Python. ☺)

For example, the following is a *valid* Python script.

```
print("hello")
"hello hello"
```

```
print("hmm no echo")
```

This comprises three statements, each of which is also an expression. The first and third lines are function calls, and the second line is just a string literal. The Python interpreter, in the non-interactive mode, or while running a script, evaluates each expression (in an "expression statement"), and it *ignores the result*.

The expressions in the first and third lines evaluate to `None` (because the `print` function returns nothing), and the value of the second line expression is the string itself. *They are all ignored.*

(Now is the good time to remind yourself how the Python interpreter works differently in the interactive and non-interactive modes.)

As mentioned earlier, however, evaluating expressions might have some "side effects". In particular, function calls tend to have side effects. This is especially true for the functions that do not return any value (since there is no point of calling a function that neither returns any (useful) value nor has any (useful) side effects).

The `print` function happens to have a side effect, which the reader should be familiar with at this point. (So, what is the side effect of calling `print()`? ☺).

If you run this script (e.g., after saving it to a file, etc.), then you will get the following output.

```
hello
hmm no echo
```

Hmm... whatever happened to the `"hello hello"` expression? Well, the Python interpreter evaluated it and just ignored its result. Sadly, no echo. ☹ We ended up wasting precious CPU cycles for nothing. ☹



In case it is not obvious, when the author uses smileys (and, sometimes even when there are no explicit smileys ☺), it means

that he is not entirely serious about the statement(s) that he has just made. 😊 He will try to use smileys more sparingly from now on. 😊

In theory, we can just "liter" all kinds of expressions (with no side effects) throughout a Python script, and the script will run just as intended, ignoring the "junks". That will be *really funny*. And, *annoying*. 😊

Fortunately, most Python programmers do not do such (useless) things. As a matter of fact, many programmers (even "experienced" programmers) do not even know that this is generally possible. Lucky for us!



Now, you are an "advanced" Python programmer now that you know one more "secret" of Python 😊 BTW, did the author mention that he was not being totally serious when he used smileys? 😊

12.7. Doc Strings

There is an exception, however. (There is always an exception. 😊)

Python does not have the "multiline comments". The hash symbol `#` starts a comment that ends at the end of the (physical) line. Most *C-style* languages support multiline comments `/* ... */`, which can span one or more lines.

One of the common uses of the expression statements which do not have any side effects is using a string literal as a kind of "doc comment", as is generally known. (This is called the "doc string" in Python since it is a string, not a comment, as in many other programming languages.) Since the Python interpreter eventually ignores the string literals, when they are used as stand-alone statements, this kind of string literals have "meanings" to the programmers (and other tools) only. As stated, this is a very common practice. In fact, it is more or less a part of the Python language.

For illustration, we only added the doc strings to the two functions in `rps2/game.py`. Generally, we will want to add the doc strings for all functions and

classes, especially those which are intended to be shared.

In this `game.py` script, the string literal, `"""Play ..."""`, on line 9 is the doc string for the function `play`. Likewise, the string literal `"""Print ..."""`, on line 24 is the doc string for the function `print_result`.

Note that these strings are part of the function definitions. (That is, they are inside the `def` statements.) These doc string literals are often written over multiple lines (e.g., using the triple quote long strings instead of simple single or double quote strings). For instance,

```
def play(max_rounds: int = MAX_ROUNDS):
    """Play max_rounds of Rock Paper Scissors.

    After the game,
    it prints out the "total wins" and "total rounds".
    """
    # The implementation goes here...
```

Note the empty line after the first "summary" line. This is a convention. The doc string generally includes one liner at the top, and it can optionally include more detailed description after a one-empty-line gap.

One thing to note is that, as before, the double quotes (`"`), or single quotes (`'`), are not escaped in the triple quote strings. A sequence of three or more consecutive double quotes (or, a sequence of three or more single quotes in a triple quote string using single quotes) still needs to be "escaped" in some way. (How? Why do you need "escaping" in this case? How would you escape a three or more matching quote character sequence within a triple quoted string?)

Let's try and view our own documentations in the REPL:

```
>>> import game
>>> help(game.play)
Help on function play in module game:
```

12.8. Ellipsis (...)

```
play(max_rounds: int = 3)
    Play max_rounds of Rock Paper Scissors.
(END)
>>> help(game.print_result)
Help on function print_result in module game:

print_result(wlt: str)
    Print a text corresponding to win/loss/tie.
(END)
```

- ① Again, don't forget to `import game` first in the interactive mode. As we briefly mentioned before, this is a way to quickly experiment/test the ideas, in the Python REPL, while working on a script (e.g., in a file).
- ② You may have to press "q" at this point to go back to the REPL prompt.
- ③ The same here.

If somebody else uses your module, then they can also view your API docs in the same way.

12.8. Ellipsis (...)

There is another common use case where an "unnecessary" expression (with no side effects) is used as a statement by itself. As stated, some compound Python statements syntactically require at least one statement in certain positions. We can just put no-op expression statements in those positions.

For example,

```
def play(max_rounds):
    'To be implemented'
```

This is perfectly valid. At least, syntactically. We have seen the `pass` statement used for this purpose. One other commonly used placeholder is an Ellipsis value, `...` (three

dots). Ellipsis is a value of the `ellipsis` type, whose only value is the Ellipsis. (`...` is an `ellipsis` literal).

```
>>> ...
Ellipsis
>>> type(...)
<class 'ellipsis'>
```

There are a few places where the Ellipsis is used, which we will not discuss in this book. But it can also be used just as an expression statement that does nothing, that is, as a placeholder. For instance,

```
def play(max_rounds):
    ...
```

Or,

```
x = input()
if len(x) > 0:
    ...
else:
    ...
```

Many people use the Ellipsis instead of the `pass` statement as temporary placeholders during the development. You can go either way, but just remember that "being consistent" is always a good thing.

12.9. `random.choice()`

The standard library `random` module includes a number of different functions for generating random numbers or sequences. We used the `random.randint` function in the earlier lessons, to generate a random integer. We then mapped the generated

12.9. `random.choice()`

integer to one of the three hands, rock, paper, and scissors.

One of the more convenient functions to use in this context is `random.choice`. Let's take a look at the documentation.

```
>>> import random
>>> help(random.choice)
Help on method choice in module random:

choice(seq) method of random.Random instance
    Choose a random element from a non-empty sequence.
(END)
```



As stated before, many of the functions in the `random` module, including `randint` and `choice`, are both functions and methods. This is a detail that is not very important to us at this point of learning.

The `choice` function takes an argument of a sequence type, and it returns a random element from the sequence. This is the perfect function for our purpose. We just need to pick a random hand out of three.

This function is used in the implementation of `_computer_hand` in the `rock_paper_scissors` module (line 22).

```
def _computer_hand():
    return random.choice((_R, _P, _S))
```

Note that we use a tuple of three string values, `(_R, _P, _S)`, as an argument to the `choice` function. We could have used a list instead. We could have even used a string. For example, the call `random.choice("rps")` would return one of the three strings, or characters, `"r"`, `"p"`, and `"s"`, with (more or less) equal probabilities.

Both tuples and strings are immutable types. If we had used a list, e.g., `[_R, _P, _S]`, instead, then we would have probably created a name for it, and reused it throughout the program. For example,

```
rps = [_R, _P, _S]
def _computer_hand() -> str:
    return random.choice(rps)
```

Otherwise, a new list object (albeit with the same items) would be created every time we call the `_computer_hand()` function. (Why do you think that would be the case? Why do you think, in general, that would not be a very good thing? ☺)

12.10. Sequence Replication

Python's builtin operators are often "polymorphic" in the sense that they behave differently depending on their operand types. (And, many of these operators (e.g., arithmetic and comparison operators) can be "overloaded" for the user-defined types, which we will *briefly* mention in the next lesson.)

For example, the star operator `*` is used for multiplication between the numeric arguments. But, the same operator works differently when one of the arguments is a sequence and the other is an `int`. (An "operator" is just a function that provides a special syntax for calling.)

For example, the `horz_bar` function, defined in lines 1-2 in `rps2/output.py`, includes the following expression as the `print` function argument:

```
"+" + "-+" * 20
```

This expression has three operands with two operators, `+` and `*`. Because of the operator precedence rule, the expression `"-+" * 20` will be computed first. (Note: an operator overloading does *not* change the operator precedence of the given operator.)

12.11. f-String Expressions

The left hand side operand of `*` is a string `"-"` (of length 2), and the right hand side operand is an integer `20`. What does this mean? When a sequence (e.g., a string as in this example) is multiplied by an `int`, the sequence is "replicated" by the specified number.

For instance,

```
>>> "hello " * 3
'hello hello hello '
```

The order is not important. You can swap the operands from both sides, and you will get the same result. That is, `3 * "hello "` would have evaluated to the same value.

So, what would be the result of calling `print("+ " + "-" * 20)`? (Well, we already have seen the answer. 😊)

We can also "replicate" tuples or lists, or other sequence type objects. For example,

```
>>> [1, 2] * 4
[1, 2, 1, 2, 1, 2, 1, 2]
```

How about `10000 * ("hello", 33.3)`? And, how about `("hello", 21) * 3.3`? Easy, right? 😊



If the answers to these questions are not obvious to you, then you can always try them in the REPL. 😊

12.11. f-String Expressions

The formal name for the "f-string" is the "formatted string literal". Despite its name, however, it is not a constant literal. The f-string lets you include expressions in the string literal syntax, but overall it is an expression (which is evaluated at run time).

Syntactically, an f-string is a string literal prefixed with `f` or `F`. Either a one quote or triple quote string literal can be used (using either the single quotes or the double quotes). The "embedded expressions" are written as `{expression}`, with a pair of curly braces. These expressions are replaced with their values *at run time*. The f-strings are also called the "string interpolations" or "interpolated strings", and most modern programming languages support similar syntaxes.

For example,

```
name = "Joe"
greeting = f"Hello {name}!"
```

The value of `greeting` will be, at this point, `"Hello Joe!"`, after executing these two statements. This could have been constructed using the string concatenations, which we learned earlier. That is, the second statement in the above example is equivalent to the following:

```
greeting = "Hello " + name + "!"
```

Although somewhat convoluted, one can even use the literals in the `{expression}` (since a literal is an expression). For instance,

```
this_year = f"{2021}"
```

The variable `this_year` is, or refers to, a string object `"2021"` (although the expression `2021` is of type `int`). This f-string expression effectively converted a given number to a string. In fact, this is more or less the same as the following, using the builtin `str` constructor function:

```
this_year = str(2021)
```

12.11. f-String Expressions

The `str` function, which we have not used before (in this book), works just like other "constructor functions", e.g., `int`, `bool`, or `list`, `tuple`, etc.

Here's an even more convoluted example: 😊

```
farewell = f'{"Bye bye world!"}'
```

This f-string on the right hand side of the assignment evaluates to `"Bye bye world!"`, and `farewell` now refers to this string object `"Bye bye world!"`. Note that we used the same string object in the `{expression}`. The f-string merely returned the same object. 😊



Note the use of the single quotes (') in the double quoted string (a string literal or f-string) in this example. One can also include the double quotes, without escaping, in a single quoted string.

In the rock paper scissors program earlier, the expression, `f">>> You won {wins} rounds out of {rounds}. <<<"` (line 14 of `rps2/output.py`), evaluates to different strings, depending on the values of `wins` and `rounds`, at run time.

```
def end_banner(wins: int = 0, rounds: int = 0):
    print("Thanks for playing Rock Paper Scissors!")
    if rounds > 0:
        print(f">>> You won {wins} rounds out of {rounds}. <<<")
    horz_bar()
```

The f-string expressions tend to be more readable than other alternatives like the string concatenations or the `str.format` function, which we do not cover in this book. (You can always do `help(str.format)` or do a Web search, to learn more, *now that you know such a method exists*. 😊)



Python's help docs seem to need some more examples. As stated, Python is an open-source project, and you can contribute to the

project, for example, by improving on the current documentations (which *everybody* uses). You can create a "pull request" once you have made some updates. ☺

12.12. Conditional Expressions

We have been using the `if - elif - else` conditional statement throughout this book. The `if` statement is a statement. Duh! ☺ Python also supports conditional expressions, through the `if - else` expression syntax.

(What's the difference between the expression and the statement again? ☺)

For instance,

```
family = input("What is your family name? ")
is_reptile = True if family == "Python" else False
```

Note that, in the simplest usage, the `if` expression includes three separate (sub-) expressions, e.g., `<v1> if <exp> else <v2>`, where `<exp>` is a Boolean expression that returns `True` or `False`. (Or, any expression will be evaluated to a `bool` value in this context.) If this Boolean value turns out to be true, then the overall value of the `if` expression is `<v1>`. Otherwise, the value of the expression is `<v2>`.

Unlike in the case of the `if` statement, the `else` part is always required in the `if` expression.

Python's `if` expression corresponds to the ternary operator `? :` in other C-style languages. An expression `<v1> if <exp> else <v2>` in Python is roughly equivalent to `<exp> ? <v1> : <v2>` in those languages.

Note that *only two* of these three (sub-) expressions are evaluated regardless of the value of `<exp>`. The condition `<exp>` is always evaluated first, and if it true, then `<v1>` evaluated, but not `<v2>`. Otherwise, `<v2>` (after `<exp>`) is evaluated, but not `<v1>`.

12.13. "States"

As emphasized a few times before, expressions are more versatile than statements. For example, an `if` condition expression, but not an `if` statement, can be used in an `f-string` expression.

```
family = "Guinea Pig"
print(f"You are {'a reptile' if family == 'Python' else 'nobody'}!")
```

This will print out `You are nobody!` 😊

12.13. "States"

Any (non-trivial) program maintains its "internal states" while the program is running. As a matter of fact, the "states" is one of most important components of the imperative programming. (On the other hand, it is the complete opposite in the functional programming. The *states* is, in fact, a "dirty word" in the pure functional programming. 😊)

In our rock paper scissors program version 2, we keep track of the number of wins, for example, using an internal variable, `wins`, in the `play` function of the `game` module.

```
wins = 0
for _ in range(max_rounds):
    ...
    wins += 1 if wlt == WIN else 0
    ...
```

This is what we called the "variable-centric view" earlier. The `wins` variable may be bound to different objects through the iteration. These objects literally *come and go*. We are mostly interested in the variable `wins`.



And, to "go" can mean something more serious, or ominous, to these objects since they can be garbage-collected out of their existence. ☹

We do not discuss the garbage collection in this book. For beginners, it is not crucial to understand how the garbage collector works in Python. The important thing to note is that memory is automatically managed by Python, and you do not have to worry about "freeing memory", etc.

The operator `+=` is called an "augmented arithmetic operator". There is one for each arithmetic operator, e.g., `*=`, `-=`, `/=`, etc.

The statement `wins += 1` (yes, it is a statement, not an expression) is equivalent to

```
wins = wins + 1
```

How does this statement work? As we saw earlier, in an assignment, the expression on the right hand side is evaluated first. Then, the result object (somewhere in memory) is bound to the name on the left hand side. In this case, when we evaluate the RHS expression, the value of `wins` might be, say, `2`, and the value of the expression is then `3 (= 2 + 1)`. The same name `wins` is then re-bound to this (new) object `3`. Hence, effectively, the value of the variable `wins` has increased by `1` after executing this statement. (More accurately, `wins` is bound to two different objects before and after the statement.)

The statement `wins += 1` works more or less the same way (although it could be *slightly* more efficient in terms of memory usage). The value of the variable `wins` ends up being increased by `1` after executing this statement.

In our example,

```
17 wins += 1 if wlt == WIN else 0
```

We do not always add a fixed number `1`. Instead, we add `1` or `0` depending on whether the player has won or not, using the `if - else` expression, `1 if wlt == WIN else 0`, that we just discussed.

12.14. For Range Loop

Note that this statement is the same as

```
wins += (1 if wlt == WIN else 0)
```

Or, even

```
delta = 1 if wlt == WIN else 0
wins += delta
```

In this augmented assignment, line 17 of *rps2/game.py*, the original `wins` will need to be evaluated first before evaluating the RHS expression. Then, `wins` is updated again after computing the RHS expression as a whole (e.g., like `delta` above). (This is generally true for the augmented arithmetic statements.)

We will discuss the `for` and `while` loops next.

12.14. For Range Loop

The `for` loop compound statement in Python is a little bit different from its counterpart in C. (Other C-style languages support a number of different variations to the classic `for`.) Python's `for - in` *always* loops over a sequence. For example,


```
>>> for reptile in ["python", "boa", "lizard"]:
...     print(f"{reptile} is very tasty.")
... 
```

This will repeatedly execute the body of the `for` statement (or, the "suite") with the loop variable `reptile` replaced by each of the elements in the given sequence (after the `in` keyword). In this example, the given sequence is a list of three elements, and hence the suite, the single `print()` statement in this case, is executed three times, with `reptile = "python"`, `reptile = "boa"`, and `reptile = "lizard"`, in this order. Here's a sample output:

```
python is very tasty.
boa is very tasty.
lizard is very tasty.
```

Note that the variable `reptile` is bound to each of the items in the list, through the iteration. And, not to the "index" values like `0`, `1`, ...



As a full disclosure, the author has never tasted reptiles in his life. 

Since a string is a sequence in Python, we can also iterate over a string using the `for in` loop. For instance,

```
>>> for c in "hello santa claus":
...     print(c if c != ' ' else '', end='')
... 
```

This will execute the (`print()` statement) 17 times. (Because `len("hello santa claus")` is 17.) In each iteration, the value `c` will be one of the characters (i.e., one letter strings, `"h"`, `"e"`, `"l"`, ...) in the given string. Therefore the output will be something like this:

```
...
hellosantaclaus>>>
```

Note that we use the `if` conditional expression that we just learned earlier to remove the spaces from the given string. The keyword argument `end=''` directs that the `print()` function should not add anything after each call. (By default, `print()` adds a newline.) In this example output, there is no newline even after the last character. (The Python shell prompt `">>> "` is printed on the same line.)

One of the most common use cases of the loops is iterating over an integer sequence.

12.14. For Range Loop

Python's `for - in` statement can be used for iterating over an integer sequence using the builtin `range` function.

```
>>> help(range)
class range(object)
|   range(stop) -> range object
|   range(start, stop[, step]) -> range object
|
|   Return an object that produces a sequence of integers from start
(inclusive)
|   to stop (exclusive) by step.  range(i, j) produces i, i+1, i+2, ...,
j-1.
|   start defaults to 0, and stop is omitted!  range(4) produces 0, 1, 2,
3.
|   These are exactly the valid indices for a list of 4 elements.
|   When step is given, it specifies the increment (or decrement).
|
...
(END)
```

The `range` function is used to generate an integer sequence. You can call this function with two or three `int` arguments to specify a beginning (inclusive) and an end (exclusive), and optionally an increment (or, a decrement, if `step` is negative). The `step` argument cannot be zero. Its value is `1` by default. That is, `range(start, stop)` is equivalent to `range(start, stop, 1)`.

Note that, even though Python does not support the "function overloading", the builtin functions are special. The `range` function can also be called with only one argument. In such a case, the call `range(stop)` is equivalent to `range(0, stop)`. (This is often called the "polymorphism", in a broad sense.)

In the `play` function in the `rps2/game.py`, we repeat playing the rock paper scissors rounds for `max_rounds` times, using `for _ in range(max_rounds)`, lines 14-18.

```
for _ in range(max_rounds):
```

...

This is, as indicated, a very idiomatic, or "Pythonic", way of iteration, which roughly corresponds to the use of the classic C-style `for` in other languages.

Note that, in this example, we do not use the loop variable in the loop body, and just use the conventional "throwaway" variable name `_`. (As stated, the name `_`, which is by the way a valid identifier according to the Python grammar, has no special meanings in programs, unlike in the interactive mode. It is just a convention to use `_` for the unused variables which are otherwise grammatically required.)

Let's look at a simple example. How do you add all integer numbers from 1 to 100? This is one of the classic beginner's exercises when they start learning programming.

```
_sum = 0
for i in range(1, 100 + 1):
    _sum += i
print(f"sum (1-100) = {_sum}")
```

This `for` iterates from `i = 1` to `i = 100`, and adds the number `i` to the `_sum` variable, ending up adding all these numbers to `_sum`



In practice, we would not do this. There is a closed mathematical formula for the sum of the (consecutive) integers. In fact, Python has a builtin function `sum` that does this. That is why we are using the variable name `_sum`. What will happen if we use the variable `sum` instead? (This is a "trick" question. 😊)



We will not discuss the `enumerate` function in this book. But, for completeness, the readers are encouraged to look this up, if interested. Using `enumerate`, Python's `for` can be used like the C-style `for`, e.g., when the "index loop variable" is needed.

12.15. While Loop

Many different (imperative) programming languages support multiple different ways of iteration or "looping" (which indicates how important "repetitions" are in programming). For example, many C-style languages support both `for` and `while` loops (among others). In such languages, there is a big overlap in the use cases between `for` and `while`.

In Python, the overlap is relatively small (although many loops can still be implemented in either way using `for` or `while`).

As stated, the `for` loop primarily iterates over a sequence (including a range). On the other hand, the `while` loop is used for repeated executions *while* a given Boolean expression is `True`. (Pun intended. 😊) Here's an example:

```
>>> year, age = 0, 30
>>> while year <= age:
...     print(f"Python is now {year} years old.")
...     year += 1
... 
```

This will print out the sentence `Python is now <x> years old.` for 31 times, from *Python is now 0 years old.* to *Python is now 30 years old.*, as long as the expression `year <= age` remains true.

The `while` Boolean expression, `year <= age` in this case, is evaluated and tested through every iteration, including the very first time. Hence, the body of the `while` statement (or, the *suite*) may not be executed at all, not even once, depending on the initial value of the Boolean expression.

Note that the loops (using either `for` or `while`) can be "nested". That is, the body of a loop can include another loop, and so on. (This is a simple corollary to the more general fact that a compound statement in Python can include another compound statement.)



A Python (a snake) in the wild typically lives 20 ~ 30 years, depending on the species. Some Pythons live over 40 years (e.g., in the zoo). What about Python the Programming Language? How long will it live? 😊

In the `play` function in the `rps2/game.py`, the `for _ in range(max_rounds)` loop can also be implemented using `while`. For instance,

```
counter = 0
while counter < max_rounds:
    counter += 1
    # do whatever you need to do here
```

In this particular example, the use of `for in range` seems a bit simpler. But, in many cases, `for in` and `while` are interchangeable, as stated above.

The `while` statement (as well as the `for` statement) can optionally include a trailing `else` clause. When the `while` Boolean expression evaluates to `False`, the statements in this `else` clause is executed before terminating the loop. The `while` statement, therefore, may look like this in general:

```
while <expression>:
    pass # While suite
else:
    pass # Else suite
```



We do not use `else` in the context of `for` or `while` in this book. As stated, although we do not cover every feature of Python in this book, we briefly mention certain terms/concepts so that the readers can continue learning *after finishing this book*. 😊

The `while` statement can also be used for an "infinite loop". In the `_user_hand`

12.15. While Loop

function of the `rock_paper_scissor` module, lines 9-18, we use the "infinite loop":

```
9 while True:
10     ...
```

This is an infinite loop as far as the `while` Boolean expression is concerned since the Boolean expression, `True`, always evaluates to `True`. Clearly, a program that does not terminate, e.g., without doing anything useful, will not be, well, very useful.

Python, just like many other imperative programming languages, supports the `break` statement. The `break` simple statement can be used to *break* out of a loop (either `for` or `while`). Note that if it is called inside the nested loops, it only breaks out of the innermost loop where this statement is executed.

Note that when the loop ends with `break`, rather than through the normal iteration, the `else` clause, if present, is not executed. This is true for both `for` and `while` loops.

In our rock paper scissors program, we use the `return` statement (line 12) to break out of the (indefinite) `while` loop (e.g., when there is no error in the user input). Since this loop is inside a function definition, the `return` statement effectively terminates the loop (even inside a nested loop).

```
def _user_hand():
    while True:
        ...
        return u[0].lower() ①
    ...
```

- ① If the user types in a valid input, then we need not continue with the loop. We just return this valid input to the caller of this function. The statement like `while True: break` or `while True: return` is a common idiom.

Another way to end a loop in the middle is essentially to terminate the program. By

terminating the program, the iteration will be stopped, along with everything else. It is not as terrible as it may sound though. ☺ For small programs, when you run into an issue, exiting the program is often the simplest and easiest solution.

In this example, we deliberately terminate the program when the user inputs EOFError (e.g., Ctrl+D) or KeyboardInterrupt (e.g., Ctrl+C) signals, lines 13-15. We will discuss the "error handling" shortly. But, the function `sys.exit`, from the standard library `sys` module, terminates the program regardless of which part of the program is currently being executed (e.g., even if it is inside a loop).

The `continue` statement stops executing the rest of the body of the loop, and it goes back to the beginning of the loop (e.g., possibly for another iteration). In case of the `while` loop, it amounts to testing the Boolean expression again. And, depending on its value, there can be another iteration, or the loop can terminate and it goes to the next statement immediately following the `while` compound statement.

In case of the `for` loop, the iteration moves to the next element in the sequence, if any.

12.16. Error Handling

In the programming world, an "error" does not necessarily mean a real error, as in "somebody made a mistake". Errors are often part of the normal execution of any (non-trivial) program.

Errors, or aka exceptions, in Python are handled via a special mechanism, e.g., outside the normal function call and return framework. This is called the exception handling framework. Not all programming languages support exception handling. Most notable exceptions (no pun intended ☺) are C and Go, among others. In those languages, they simply use the regular call chain to propagate any error messages to the caller, and to its caller, and so on.

Python supports the more or less typical exception handling syntax via `try - except - finally`. This "`try` compound statement" in Python can also include another optional `else` clause, after the last `except` clause and before `finally`, if

12.16. Error Handling

present.

In many languages that support exception handling, exceptions are "thrown". That is, they **throw** an exception as if it is a hot potato or a baseball on fire. 😊 In Python, on the other hand, we **raise** an exception just like we raise a red flag when something happens that requires our special attention. That is precisely the point of using exceptions, say, instead of using the regular call return chain. Exceptions need to be handled with care. 😊

Here's an example of "raising an exception":

```
>>> raise ValueError("oops")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: oops
```

Or, more interestingly,

```
>>> if False < True:
...     raise Exception("Really?")
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
Exception: Really?
```

Of course, this is a silly example. **True** is always bigger than **False** (really? why? 😊), and hence this **if** statement will always raise an exception.

When an exception is "thrown", you can "catch" it, if you need to. In most languages, it is **try - catch**. On the other hand, in Python, it is **try - except**.

If any statement in the **try** clause (including any function calls) raises an exception, then you can handle the (particular) exception in one or more **except** clauses. Here's the general syntax:

```

try:
    ...
except <exception_exp_1>:      ①
    ...
except <exception_exp_2>:      ②
    ...
except:                        ③
    ...
else:
    ...
finally:
    ...

```

- ① An expression that evaluates to an `Exception`` type.
- ② A `try` statement can have zero, one, or more `except` clauses.
- ③ The "catch all" clause.

The `except` clause can specify a particular error type which it intends to handle. Most of the exception/error types in Python inherit from, or are the subtypes of, the `Exception` type. You can also create your own custom exception type by defining a subclass of `Exception`. (We will not discuss this in this book, but see later lessons for "subtyping".) Some system exceptions inherit from `BaseException` instead.

You can use at most one wildcard `except` clause without specifying a particular exception. In this case, it is the same as (`except Exception:`). (But, it does not catch the special system exceptions based on `BaseException`. They need to be explicitly handled.)

The broadest implicit `except` clause should be the last one in the `except` chain. Otherwise, a `SyntaxError` will be raised. In general, `except` with the more narrower exceptions should be placed before the `except` catching more broader exceptions.

If an exception object with a particular type is raised within the `try` clause, and if

12.16. Error Handling

there is an `except` clause that specifies that exception type (or, any of its superclasses, up to `Exception`), then the program control goes to the *first* `except` clause that handles that exception, without going through the rest of the statements in the `try` suite. Note that no more than one `except` clause will be matched in a given `try` statement.

An optional wildcard `else` clause can be used after the last `except` clause, if any. If no exception is raised in the `try` part, then the statements in the `else` suite will be executed, unless the program control leaves the `try` suite prematurely, e.g., through `return`, `break`, or `continue` statements.

On the other hand, the `finally` clause, if present, e.g., after any `except` or `else` clause, is *always* executed regardless of whether an exception is raised, or whether any `except` clause is invoked, before leaving the `try` statement. The `finally` clause is often used for "cleanup".

Syntactically, at least one `except` or `finally` clause is needed in the `try` statement. Otherwise, all other clauses are optional. That is, the following statement is valid,

```
try:
    sum = 100 + unknown_name
except:
    print("Somebody must have raised a red flag!")
```

So is the following:

```
try:
    ...
finally:
    print("Error or not, here we go.")
```

If an exception/error is raised, and none of the specified `except` clauses matches that exception in a given `try` statement, the exception propagates upward through

the call chain (after executing the statements in the `finally` suite, if present, but without executing the rest of the code in the given scope). If none of the code in the upstream call chain handles the exception, the program terminates with the given error. That is, the program "crashes". ☹️ (The Python interpreter in the interactive mode, however, catches all exceptions/errors.)

Note that the statements in the `else` and `finally` suites can raise an exception, and this exception is not handled by the `except` clauses, if any, of the current `try` statement. It will propagate upstream.

As with `import` statements, the keyword `as` can be used to give a name to an exception object that is matched with a particular `except` clause. For instance,

```
try:
    sum = 100 + unknown_name
except NameError as ex:
    print(ex)
```

This code sample will print out the following output:

```
name 'unknown_name' is not defined
```

An exception can be "re-raised" (as in a poker game ☺️). For example,

```
>>> try:
...     sum = 100 + unknown_name
... except:
...     raise
...
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: name 'unknown_name' is not defined
>>> ①
```

12.16. Error Handling

- ① The Python REPL handles the error, and it does not crash. It waits for the next user command.

In the context where an active **Exception** is present, the **raise** statement re-raises the current exception/error. This example code is more or less equivalent to the following, which explicitly raises the current exception.

```
>>> try:
...     sum = 100 + unknown_name
... except NameError as ex:
...     raise ex
...
Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
  File "<stdin>", line 2, in <module>
NameError: name 'unknown_name' is not defined
```

In the rock paper scissors game, version 2, we use the **try** statement to catch any possible errors, and other signals, from the statements (e.g., the **input** function call) within a **while** loop (lines 10-18, **rock_paper_scissors.py**).

```
10 try:
11     u = input("Rock (r), Paper (p), or Scissors (s)? ").strip()
12     return u[0].lower()
13 except (KeyboardInterrupt, EOFError):
14     print("\nThanks for playing the game!")
15     sys.exit()
16 except:
17     print(f"Invalid input, {u}. Try again")
18     continue
```

Any Python program can raise a **KeyboardInterrupt** exception e.g., when the user presses Ctrl+C (on Unix-like platforms) while the program is running. **KeyboardInterrupt** is one of the special system exceptions, as mentioned, and it

inherits from `BaseException`, and not from `Exception`.

The builtin `input` function raises an `EOFError` exception, when its input includes no valid string (not even an empty string). For example, when the user just presses Ctrl+D (on Unix-like platforms) at the input prompt, the `input()` function will raise `EOFError`.

Note that this is the way that the `input` function is designed/implemented. It is part of the public API. Alternatively, the designers of the Python language could have decided to return an invalid value like `None` instead of raising an error in this kind of situations. (But, they didn't.)

Likewise, when you write a Python program, you have a choice. Sometimes using the exception handling framework is good, and sometimes it is not. You do not always have to use exceptions in error-like situations. In general, however, for input/output handling and for network related functions, for example, it is often a good practice to raise an exception(s) in unpredictable situations.

In our program, we handle these two errors, or more like the "signals", in the same way. The tuple-like syntax after the keyword `except` matches any of the exceptions specified in the tuple. (The order is not important.) In this particular case, we just terminate the program by calling `sys.exit()`. That is, we handle the user action that leads to one of these exceptions as the user's explicit intention to terminate the program.

Alternatively, we could have handled these exceptions differently, possibly leading to different behaviors of the program. It is largely a design decision (e.g., based on the requirements, etc.).

For all other types of errors, in our rock paper scissors program, we just ignore the error, and give the user another chance. For example, when the user presses Enter without inputting any text, the `input` function will return an empty string. When we try to access the (non-existent) first element of the string (`u[0]`), Python will raise an `IndexError` exception.

12.17. Putting It All Together

It is often easier to read, and understand, the code if we read a program starting from the "top" (or, the head, starting point). But, in general, it is not always obvious where the top is in the large Python programs (with many source files). ☺ In fact, there may be more than one "top" in a collection of Python program files.

In some programming languages and runtimes, there is a single entry point to the program (e.g., the `main()` function). Python does not work that way.

Python executes any (single) script that is provided to the interpreter, and that script is the "top". And, that is the "main script". Everything else that is imported into this script, either directly or indirectly, becomes a part of the program.

Python starts running the program by first executing the first statement in the script, and it goes from there. In most cases, the control moves more or less from top to bottom in the script.

In our program, either `game.py` or `rock_paper_scissors.py` can be used as a starting script. If we run `game.py`, we will end up playing multiple rounds. On the other hand, `rock_paper_scissors.py` only does a single round and it quits.

Let's use the `game.py` script for illustration here. The script essentially consists of one (compound) statement, the `if` statement in lines 34-35, excluding other `import` statements, name bindings, and definition statements, etc., which do not actually perform any visible actions (e.g., to the external world).

```
34 if __name__ == "__main__":  
35     play()
```

Since the runtime module name is always `__main__` when the Python file is run as a script, and hence the expression `__name__ == "__main__"` always evaluates to `True`, and the `game.py` script will just run one (simple) statement, `play()`. The script will then terminate once the `play()` function call returns.

The `play` function has the following signature: `play(max_rounds: int = MAX_ROUNDS)`. It takes one optional `int` argument (named `max_rounds`), and if it is called without an argument, then the default value `MAX_ROUNDS` is used, which is currently hard-coded to `3`.

```
5 MAX_ROUNDS: Final[int] = 3
```

Note the naming convention, as we explained earlier. All caps generally indicates that the variable is a constant, and its value will not change. (In other words, we will not use this variable `MAX_ROUNDS` to refer to anything else but this initially assigned object, `3`.) We also use the type annotation `Final[int]` to indicate (to ourselves and/or to any static type checking tools) that it is indeed a constant.

The `play` function starts with a docstring (line 9), which is generally a good practice, and it consists of four statements (excluding the docstring): A function call, `start_banner()`, an introduction of a new name `wins`, which is initialized with an `int` object `0`, a compound `for` statement (lines 14-18), and finally another function call, `end_banner()`, which takes two arguments, `wins` and `max_rounds`.

```
def play(max_rounds = MAX_ROUNDS):
    start_banner()

    wins: int = 0
    for _ in range(max_rounds):
        ...

    end_banner(wins, max_rounds)
```

The Python interpreter executes these four statements, in this order, from top to bottom, within this function (that is, when this function is *called*). Note that, since the statements in the `for` statement, for instance, call other functions, etc., the program execution is not strictly linear. Nonetheless, you can easily convince yourself that Python programs are "structured" (in some way).



When we use the "top down" metaphor, we do not mean from line 1 to line 100, for instance. In this example, the `play` function is at a "higher" level than `start_banner` or `play_a_round` functions since `play` calls these other functions. We roughly see the program flow from top to bottom (or, sometimes, from front to back). This kind of strict top-down structure is not completely required in Python. For example, one function can call another function which calls back the same function, etc. In general, however, we strongly recommend the (beginning) programmers to always stick to this kind of top-down structure. As stated, a (large) software is an *extremely complex system*. This kind of "discipline", if you will, will make your software systems much more manageable.

Starting from the `start_banner()` function call (line 11), this function is defined in the `output` module. It prints out three lines of text, two of which are the "horizontal bars", as defined in the `horz_bar` function.

```
1 def horz_bar():
2     print("+" + "-+" * 20)
```

The argument of the `print()` function is a string expression, which we went through earlier. Because the star operator `*` has a higher precedence, this expression is the same as `"+" + ("-" * 20)`. Note that this expression is really a constant, and it can be simply replaced with `"+-+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+"`.

The next statement in the `play` function is an assignment.

```
13 wins = 0
```

Note that there is a one-line gap above this statement, but there is no empty lines below this statement. This (informally) indicates that the name `wins` is sort of a part

of the next grouped statements.

There are two kinds of styles (and, the whole gamut in between) when "declaring" (and initializing) new variables in a program (even in Python). One style is putting all variables in the beginning of a program, in the beginning of a function, or in the beginning of a block (in the C-style languages), etc. Another style is declaring a new variable closest to where it is used, e.g., just before its first use, etc.

In this example, we introduce the name `wins` just before it is used in the `for` statement body.

The main logic of the `play` function is in the `for` loop, lines 14-18:

```
14 for _ in range(max_rounds):
15     wlt = play_a_round()
16     print_result(wlt)
17     wins += 1 if wlt == WIN else 0
18     horz_bar()
```

The range function call `range(max_rounds)` generates an `int` sequence from 0 to `max_rounds - 1`. Hence, we will end up repeating the body of the `for` loop for `max_rounds` times.

In each iteration within the `for` loop, we play a round of rock paper scissors, by calling the `play_a_round()` function from the `rock_paper_scissors` module (line 15), and prints out the result using the `print_result` function, which is defined later in the program, lines 23-31. Note that the name `play_a_round` has been imported using the `from import` statement (line 3) at the top of the script.



We stated before that you cannot use a name that has not been already declared/bound. And yet, we call the function `print_result()` (line 16) *before* it is defined (lines 23-31). How is that possible?

This is because a function definition (e.g., of `play`) does not actually

12.17. Putting It All Together

execute the statements in the body. The function definition creates a function object in memory and introduces its name, the function name, to the program. By the time we really execute the program by calling `play()` (line 35), which subsequently calls `print_result()`, the function name `print_result` has been already bound to the function object, defined by the code in lines 23-31 (which is before line 35).

The `wins` variable that we initialized to `0`, just before the `for` loop starts, is then used to keep track of the number of wins.

```
17 wins += 1 if wlt == WIN else 0
```

This statement, which we discussed earlier at some length, is also equivalent to the following, using the `if` statement:

```
if wlt == WIN:
    wins += 1
else:
    wins += 0
```

Since adding zero happens to be a no-op operation, it can be simplified as follows:

```
if wlt == WIN:
    wins += 1
```

In this particular case, the `if` statement version looks a bit simpler, and easier to read, than the original `if` expression version. But, in general, it is often the opposite. Using the (one-liner) `if` expression, if syntactically allowed, can make the programs typically shorter, and easier to read. And, as stated, there are places where statements are not allowed, syntactically.

The loop continues, after printing out the "horizontal bar" as a separator between the rounds (line 18). The `for` iterates for `max_rounds` times, and it finally calls the `end_banner` function (line 20), again defined in the `output` module, to print out the final result. Note that all (non-underscore) names from this module have been imported using the wildcard import (line 2).

Then it returns to the caller, which is this `game.py` script itself, since there is no more statement in the `play` function. Since, likewise, there is no more statement after the statement `play()` in this script, the Python interpreter stops the execution of the script, and the program/process terminates.

The `end_banner` function prints out the "game end" message. It is defined with two (optional) parameters, `wins` and `rounds`, whose default values are both `0`. If a non-zero `rounds` is provided, then the "stats", e.g., the total wins (`wins`) and the total number of rounds played (which is effectively `max_rounds` in this example), is also displayed.

```

11 def end_banner(wins: int = 0, rounds: int = 0):
12     print("Thanks for playing Rock Paper Scissors!")
13     if rounds > 0:
14         print(f">>> You won {wins} rounds out of {rounds}. <<<")
15     horz_bar()

```

Note that we are assuming either that neither of the arguments is provided or that both are provided. This is reasonable. However, since Python function parameters, by default (and, as defined in this program), can be used as positional or as keyword arguments, if the `rounds` argument is provided as a keyword argument without `wins`, then the program may output potentially incorrect stats. One way to prevent this from happening is to make both parameters positional-only, as explained earlier. For instance,

```

def end_banner(wins: int = 0, rounds: int = 0, /):
    ...

```

12.17. Putting It All Together

This way, you cannot accidentally supply `rounds` without `wins`.

Alternatively, we can change the conditional logic to make sure that the stats is printed out only when both `wins` and `rounds` are provided. (How would you do that? This may require some "messy" changes. Just a little. 😊)

Another way is to combine the two parameters into one tuple parameter so that one value without the other cannot be provided in a function call. (Again, how would you do that? 😊)

Now going back to the `for` loop of the `play` function, the following two statements (lines 15-16) are the "core part" of each iteration:

```
15 wlt = play_a_round()  
16 print_result(wlt)
```

It lets the user play a round of rock paper scissors, and then it prints out its result. (The (arbitrary) name `wlt` stands for "win-loss-tie".) The `play_a_round` function in the `rock_paper_scissor` module returns one of the predefined constants, `rock_paper_scissor.WIN`, `rock_paper_scissor.LOSS`, and `rock_paper_scissor.TIE`, depending on the outcome.

The `print_result` function simply prints out different texts based on the round result. For this, we use the good ol' `if - elif - else` statement (lines 26-31). In the next lesson, we will introduce an alternative way of doing this.

Now the `rock_paper_scissors.play_a_round` function implements the rock paper scissors game logic. This function (lines 33-38 of `rock_paper_scissors.py`) roughly corresponds to the (entire) program that we wrote in the previous lesson (without iterations).

```
33 def play_a_round():  
34     u = _user_hand()  
35     c = _computer_hand()
```

```

36     print(f"You -- {_to_str(u)} vs {_to_str(c)} -- Computer")
37
38     return _determine_win(u, c)

```

The whole function is only 6 lines (including one empty line). But, it captures the essence of the core logic. It reads the user hand by calling the `_user_hand()` function (line 34), it generates a random computer hand by calling the `_computer_hand()` function (line 35), and it prints out both hands for user reference (line 36). Then, it determines whose hand wins, by calling the `_determine_win` function, and it returns the result to the caller (line 38), that is, the `play` function in the `game.py` script/module.

The implementation of the `_user_hand` function includes some minimal error handling, as we discussed earlier.

```

def _user_hand():
    while True:
        try:
            ...
        except (EOFError, KeyboardInterrupt):
            ...
        except:
            ...

```

First of all, the whole function body consists of one `while True` statement. As we learned earlier, you can break out of a loop using `break` statements, or `return` statements (if the loop is inside a function), among other ways.

The `while` statement in turn consists of one statement, a `try` statement (lines 10-18).



Can you see the "structure" based on the indentations and what not? As stated, Python programs are "structured", more so than those written in other languages.

12.17. Putting It All Together

For the `try - except` statement, the "main" part is the `try` clause. If an exception occurs within the `try` suite, then the specified exceptions will be caught in the matching `except` clause.

The `try` suite includes two statements (lines 11-12),

```
11 u = input("Rock (r), Paper (p), or Scissors (s)? ").strip()
12 return u[0].lower()
```

As we learned earlier, the `input` function returns the user input as a string. The builtin `str.strip` method removes the leading and trailing spaces from a string, if any. When the user did not input any real text (after "strip'ing"), the indexing expression `u[0]` will raise an error. This error is caught in the catch-all `except` clause, lines 16-18, and we simply `continue` through the iteration in such a case. This will effectively end up re-executing the `input` statement (line 11).

In this example, we have an `except` clause that matches specific exceptions, i.e., `EOFError` and `KeyboardInterrupt` (lines 13-15).

In such a case, we treat them as the user's request to end the game (e.g., before the normal game end) and we exit the program by calling `sys.exit()` (after printing out the game-end message).

```
13 except (EOFError, KeyboardInterrupt):
14     print("\nThanks for playing the game!")
15     sys.exit()
```

For all other errors, including the `IndexError`, we simply ignore them, and ask the user again for input through the `continue` statement, lines 16-18, e.g., after printing out a simple error message.

```
16 except:
17     print(f"Invalid input, {u}. Try again")
```

```
18     continue
```

If there have been no "errors", and if the user inputted a "valid input" (any non-empty text is a valid input in this program), then it returns the first character of the inputted text (e.g., `u[0]`), line 12, after converting it to the lower case (using the builtin `str.lower` method).

```
10 try:
11     u = input("Rock (r), Paper (p), or Scissors (s)? ").strip()
12     return u[0].lower()
```

As indicated, we could have added some input validation, and if the user has inputted an input not corresponding to rock, paper, and scissors, we could have just declared the round as the user's loss. But, often, this kind of "optimizations" are not that critical.

As for the rest of the functions that are called from `play_a_round`, we will leave it to the readers to go through them as an exercise. We covered all the essential points already.

12.18. Code Review

As indicated, error handling is not always "required". If you are the only one who is going to use the program and you know that you are supposed to input a certain valid string, for instance, then there is no reason to add any additional error handling.

We added some basic error handling in this second version of the rock paper scissors game, especially around the input processing. There is, however, still further room for "improvement". For example, currently, if the player inputs a random string like `A`, the program continues, but it eventually ends as the user's loss. We can add further input validation, that is, if that is deemed necessary, or desired.

12.18. Code Review

The new implementation of this part is "more modular". In particular, we used three files, and made each Python code file more like a module than a script. We will continue in the next part, where we will discuss some basic OOP programming techniques.

Chapter 13. Lab 2 - Functions, Loops, and More

All we have to decide is what to do with the time that is given to us.

— Gandalf (The Lord of the Rings)

As stated, the "lab sessions" are optional.

If you want to do (some of) the exercises in this lab and/or follow along with the development of the rock paper scissors game version 2, then create a new folder, e.g., named *rps2*, and do the same setup as before, including creating a new *venv*, etc. Again, the readers are encouraged to use git, or other source control system, at least on you machine. For practice, if nothing else. You can use the same repository from the lab 1, or you can create a new one.



There are (almost) always more than one ways to do the same thing, and there is no single "correct" answer for each of the following exercises. You will need to verify your solution (to make sure that it "works" 😊), but don't worry about coming up with the "best" solution.

13.1. Sum

One of the simplest problems which a beginning programmer is to do is adding integer numbers from 1 to *N*. It's like a rite of passage. 😊 We have seen an example in an earlier lesson.

In this exercise, we will write a function that takes an integer argument, *n*, and returns a sum from 0 to *n* (inclusive) if *n* \geq 0. There is a twist, however. 😊 If *n* $<$ 0, then this function should return a sum from *n* to 0, which will result in a negative

number.

Write a script that tests this function for a few different `n`'s, e.g., `0`, `5`, `10`, `-5`, etc.

13.2. Product

Although Python has a builtin function `sum`, it does not have a similar builtin function for products/multiplications.

Create a function that takes one integer argument `n` and multiplies all integers in the `range(1, n + 1)`, using the `for range` loop. The function returns the result as `int`.

Create another function that takes an argument of a sequence type, e.g., a list of `ints`, and returns the product of all the numbers in the list.

Test these two function with a number of different inputs, e.g., different `n` values and different `int` lists.

13.3. Filtered Sum

Although we do not discuss in this book, Python has a "membership test operator", `in`, which returns true if the first argument is an item of the second argument (of a collection type). It return false otherwise. We can also use `not in` to test the opposite condition. Try `help("in")` in the REPL to get more information.

Write a function that takes two arguments, an int `n` and a list of ints `excluded`, and returns an int:

- `n` is a non-negative integer. It is a "positional only" parameter.
- The function sums up all numbers from `0` to `n`.
- The second argument is optional, but if it is provided, then the numbers in the list are to be excluded in the summation. Make this a "keyword only" parameter.

For example, given `n = 5` and `excluded = [3, 4, 10]`, the function should return the sum `1 + 2 + 5`, which is 8.

13.4. Singular vs Plural Nouns

When you count things in English, the case of one item is somewhat special. For example, we might say "two monkeys", "three monkeys", "twelve monkeys", or even "zero monkeys". But, when we have only one monkey, we say "one monkey".

In the rock paper scissors program of the preceding lesson, we print out the number of wins using the f-string expression, in line 14 of the `output` module:

```
print(f">>> You won {wins} rounds out of {rounds}. <<<")
```

This will print out `>>> You won 1 rounds ...` when `wins == 1`, which is not correct grammatically. ☹ In practice, this kind of optimization is rarely important, but as an exercise, change this `print()` statement in some way so that it prints out the correct noun forms, singular vs plural, depending on the number of `wins`.

13.5. Power Operator

Although we have not discussed in the book, Python has another operator, `**`, which is the same as the builtin function `pow`. This is called the "power operator", and it does the exponentiation operation. For example, `2 ** 3` is the same as `2 * 2 * 2`, which evaluates to 8, and `3 ** 2` is equivalent to `3 * 3`, which is 9.

Write a "string power" function that takes two arguments, `base` of the `string` type and `exp` of the `int` type, and returns a string that is equivalent to `base * (len(base) ** exp)`. The argument `exp` needs to be non-negative.

For example, given `base="pi"` and `exp=3`, the function should return `"pipipipipipipi"`.

13.6. Tuple Parameter

Test this function with one-, two-, three-, and four- character strings, e.g., "h", "he", "hel", and "hell", for a few different exponents like 0, 1, 2, and 3.

13.6. Tuple Parameter

The `end_banner` function defined in the module `output` in the earlier lesson takes two optional arguments, `wins` and `rounds`.

```
def end_banner(wins: int = 0, rounds: int = 0):  
    ...
```

As discussed in the lesson, these two need to be provided as a unit. Supplying one without the other may not make sense. A better way to do this might be either using a custom type (which can be an overkill in situations like this) or just using a tuple argument.

Redefine, and re-implement, the `end_banner` function to take one parameter of an `(int, int)` tuple type. Also update the call syntax in the `game` module (line 20) accordingly.

13.7. Reverse a List

The `list` type has a builtin method `reverse`, which does an "in place" reverse on the given list. For example,

```
>>> a = [1, 2, 3, 4]  
>>> a  
[1, 2, 3, 4]  
>>> a.reverse()  
>>> a  
[4, 3, 2, 1]
```

Note that the object (which the name `a` refers to) has changed "in place".

We can also create a new list with a reverse order using slicing, which we discussed in the introductory lessons in the beginning of the book. For example,

```
>>> a = [1, 2, 3, 4]
>>> a
[1, 2, 3, 4]
>>> r = a[::-1]
>>> r
[4, 3, 2, 1]
```

The third (optional) argument in slicing is the "step". The negative step slices the list from right to left.

Now, without using the builtin list method, `reverse`, write a function that takes a list as an argument and reverses the given list in-place.

For example, the function signature might look like this:

```
def reverse_in_place(arr: typing.List[int]):
    pass
```

You can do it with a `for in` loop or using a `while` loop. Do it both ways. 😊

Now, implement a function which does the same thing, but does not change the given list argument. Instead it returns a new list with the items reversed.

```
def produce_reversed_list(arr: typing.List[int]) -> typing.List[int]:
    pass
```

Python has a builtin function `reversed` for this, which works for both mutable and immutable types. We discussed this in the beginning of this book.

13.8. Rock Paper Scissors

Do this exercise without using the `reversed` function or the aforementioned slice operation.

13.8. Rock Paper Scissors

Close the book, after reading this problem. ☺

Write a rock paper scissors program based on what you remember from the previous lesson. The program includes several core functions across a few different modules:

- A function that reads a user input and converts it into a constant, `r`, `p`, or `s`.
- A function that randomly generates one of these constants, `r`, `p`, or `s`, as a computer hand.
- A function that compares two hands and decides which hand wins.
- A function (`f1`) that plays one round, e.g., (1) reads a user hand, (2) generates a random computer hand, and (3) decides who wins.
- A function (`f2`) that plays a fixed number of rounds and prints out the total number of wins and the total number of rounds.
- Write a "main script" that calls one of these two functions, `f1` or `f2`.

Try running your main script a few times, and make sure that it works as expected.

13.9. How Many Rounds?

Now open the book again, that is, if you haven't already done so. ☺

Modify the program that you just wrote so that, instead of playing a fixed number of rounds, you ask the player how many rounds they want to play, at the beginning of the game.

13.10. Best of Seven

Modify your rock paper scissors program so that, this time, you end up playing no more than 7 rounds. Once the winner is determined, based on the wins/losses/ties up to that point, you do not need to play the rest of the rounds. This is often called "the best of X", for instance.

For example, let's suppose that we have played 5 rounds so far and the player has won 3 rounds, and there have been 2 ties. There is no way for the computer to win at this point with only 2 rounds remaining. Hence, we do not have to play the rest 2 rounds since the overall game winner is already determined.

It will require some thinking. As stated in the beginning, some programming problems like this have little to do with your Python knowledge. You need to *solve the problem*.

Object Oriented Programming

Your time will come. You will face the same Evil, and you will defeat it.

— Arwen (The Lord of the Rings)

In the previous lessons across two parts, we implemented the rock paper scissors game in two different ways. The first one was "simpler" (which is a virtue). The second one was "more modular" (which is also a good thing).

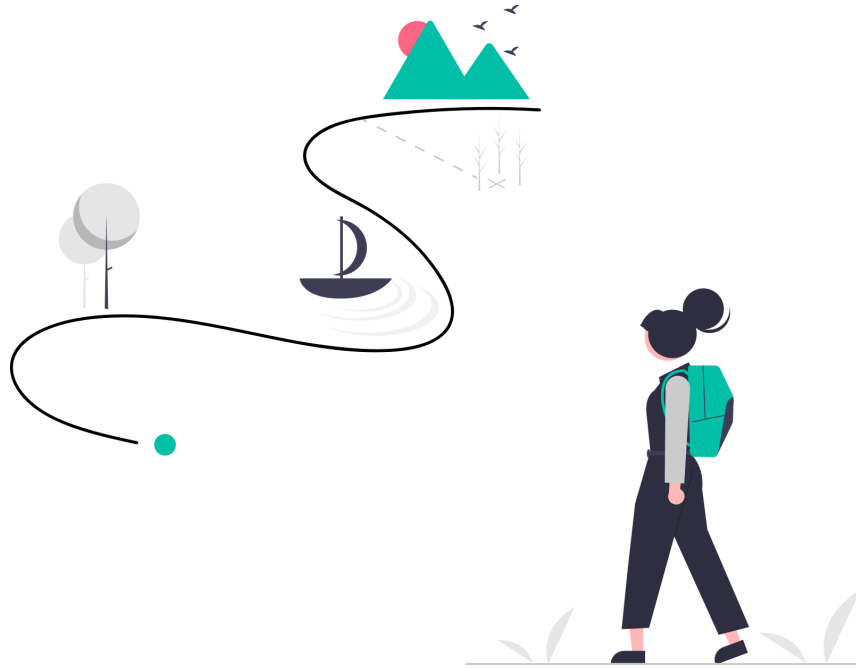
Obviously, it's not our goal to become an expert in rock paper scissors. 😊 But, let's try implementing it one more time, using different techniques. In particular, we will use the "object oriented programming" style to implement the same game.

Python is not an object oriented programming language like C++, Java, or C#. However, it supports OOP reasonably well, using `class` and what not. In Python, `class` is an ad-hoc addition to the language. As we will learn in this part, there are some (subtle) differences in the way Python's OOP works compared to more traditional OOP styles.

Although it is not essential, OOP clearly provides certain benefits in certain situations, and it is a useful thing have in your toolbox. The lessons in this part will provide a solid foundation on OOP when you program in Python.

As in the previous two parts, we start with a complete program in the first lesson of this part, and go through each of the important pieces in the program in the following lessons.

Chapter 14. Rock Paper Scissors - The Finale



We will take one more stab at implementing the rock paper scissors game in Python. In some sense, we are "progressing". We will end up using some of the "more advanced features" of the Python programming language in this new implementation. As stated, we will use some (basic) "object oriented programming" (OOP) styles. As emphasized before, however, this new program is not, in some absolute scale, "better" than the previous two implementations.

As for the OOP, in particular, you do not always have to use the object oriented styles. In fact, the OOP paradigms are way too much used (or, abused) in the modern programming even when the benefits that they provide are minimal. Remember, simplicity always has a premium.

"Everything should be made as simple as possible, but not simpler." -Albert Einstein

Classes in Python, as in many other languages that support OOP, are used to *organize the code*, broadly speaking. We use functions to group together some related statements (e.g., so that they can be made easily re-usable, etc.). Likewise, we use classes to group together some related functions and other "states".

Functions are primarily for capturing certain "actions" (e.g., in terms of their computations and side effects, etc.). On the other hand, classes encapsulate both data and actions (e.g., actions on the data).

More specifically, as we will see later throughout this final part, the `class` statement is used to create custom types. (In fact, all types are classes in Python.) A `type/class` prescribes data and actions (for an object of that type). More specifically, it is sort of a template to create instance objects of the type.



There are some (subtle) differences in the terminologies used in Python and other OOP programming languages. In general, the term "object" is used to refer to an instance of a class in OOP. In Python, an "object" is something that is stored in memory (e.g., as used by the Python interpreter). As stated, a Python object is associated with a value ("data") and some other "metadata" such as an identity and a type, etc. In Python, *everything is an object*. A function is an object. A class is an object. When referring to an object or instance of a class/type in the OOP sense, for example, Python uses more specific terms like the "instance object". We will discuss this further later in this lesson.

Our new implementation, which uses OOP, comprises 6 source files across two folders (with one being a subfolder of the other). Unlike in the previous examples, we make clear distinctions between modules/packages and scripts (for illustration). That is, the main script, `main.py` will only be used as a script, e.g., to run the program, whereas all other source code files in `rps` belong to one package module `rps`, and they are only intended to be used as (importable) modules, and not as scripts. The particular project/container folder name `rps3` in this example is not significant, as explained earlier.

As before, let's go through the complete program first. Code reading is a lost art. When we learn programming we primarily focus on writing than reading. In fact, it's a lot easier to write a program than read somebody else's program, especially when you are a beginner. (And, everybody is a beginner, in some respects. ☺)

First, here's the "main script":

rps3/main.py

```
1 import rps.game as game
2
3 g = game.Game()
4 g.start()
```

The `rps` package includes five submodules. Pay attention to the *high-level structures* than details. As emphasized, we generally read the code *from top to bottom*. (Although we did not explicitly mention, besides the caller-callee relationships, the lower-indentation structures are generally considered "higher" than those with bigger indentations.)

rps3/rps/game.py

```
1 import typing
2 import rps.reader as reader
3 import rps.rand as rand
4 from rps.hand import compare_hands
5 from rps.result import WinOrLose
6
7
8 MAX_ROUNDS: typing.Final[int] = 5
9
10
11 class Game:
12     """Game encapsulates the rock paper scissors game functionalities.
13
14     The "main function", start(), plays 5 rounds by default.
```

```

15     """
16
17     def __init__(self, num_rounds: int = MAX_ROUNDS):
18         self._wins = 0
19         self._losses = 0
20         self._ties = 0
21         self._num_rounds = num_rounds
22
23     def start(self):
24         """The "main" function. It starts the game."""
25         self._banner()
26         self._loop()
27         self._end_game()
28
29     def _banner(self):
30         print("""\
31 -----
32 Welcome to Rock Paper Scissors!
33 Type X or Q to end the game.
34 -----\
35 """)
36
37     def _end_round(self):
38         print(f"""\
39 Your wins: {self._wins}, losses: {self._losses} \
40 out of {self._wins + self._losses + self._ties} rounds
41 -----\
42 """)
43
44     def _end_game(self):
45         print(f"""\
46 Thanks for playing Rock, Paper, Scissors!!
47 Your final score:
48 Wins: {self._wins}, Losses: {self._losses}, \
49 Total rounds: {self._wins + self._losses + self._ties}.
50 -----\
51 """)
52

```

```

53     def _win_or_lose(cmp: int) -> WinOrLose:
54         match cmp:
55             case 1:
56                 return WinOrLose.WIN
57             case (-1):
58                 return WinOrLose.LOSE
59             case _:
60                 return WinOrLose.TIE
61
62     def _loop(self):
63         for _ in range(self._num_rounds):
64             player_hand = reader.read_hand()
65             if not player_hand:
66                 return
67
68             computer_hand = rand.random_hand()
69
70             print(f"Your hand: {player_hand},",
71                 f"computer hand: {computer_hand} ->",
72                 sep=" ",
73                 end=" ")
74
75             cmp = compare_hands(player_hand, computer_hand)
76             wol = Game._win_or_lose(cmp)
77             match wol:
78                 case WinOrLose.WIN:
79                     self._wins += 1
80                     print("You win!")
81                 case WinOrLose.LOSE:
82                     self._losses += 1
83                     print("You lose!")
84                 case _:
85                     self._ties += 1
86                     print("Tie.")
87
88             self._end_round()

```

rps3/rps/rand.py

```
1 import random
2 from rps.hand import Hand
3
4
5 def random_hand() -> Hand:
6     return random.choice((Hand.ROCK, Hand.PAPER, Hand.SCISSORS))
```

rps3/rps/reader.py

```
1 from typing import Optional
2 from rps.hand import Hand
3
4
5 def read_hand() -> Optional[Hand]:
6     while True:
7         try:
8             i = input("Rock (r), Paper (p), or Scissors (s)? ").lower()
9             if i.startswith("q") or i.startswith("x"):
10                 return
11
12             return _parse_hand(i[0])
13         except (EOFError, KeyboardInterrupt):
14             return
15         except:
16             print("Invalid input. Try again.")
17             continue
18
19
20 def _parse_hand(s: str) -> Hand:
21     match s:
22         case "r":
23             return Hand.ROCK
24         case "p":
25             return Hand.PAPER
26         case "s":
```

```
27         return Hand.SCISSORS
28     case _:
29         raise ValueError("Invalid hand")
```

rps3/rps/hand.py

```
1  from enum import Enum
2
3
4  class Hand(Enum):
5      ROCK, PAPER, SCISSORS = "r", "p", "s"
6
7      def __str__(self) -> str:
8          match self:
9              case Hand.ROCK:
10                 return "Rock"
11              case Hand.PAPER:
12                 return "Paper"
13              case Hand.SCISSORS:
14                 return "Scissors"
15              case _:
16                 raise ValueError
17
18
19  def compare_hands(h1: Hand, h2: Hand) -> int:
20      match(h1, h2):
21          case (Hand.ROCK, Hand.SCISSORS) | (Hand.PAPER, Hand.ROCK) |
22            (Hand.SCISSORS, Hand.PAPER):
23              return 1
24          case (Hand.SCISSORS, Hand.ROCK) | (Hand.ROCK, Hand.PAPER) |
25            (Hand.PAPER, Hand.SCISSORS):
26              return -1
27          case _:
28              return 0
```

rps3/rps/result.py

```
1 from enum import Enum
2
3
4 class WinOrLose(Enum):
5     WIN, LOSE, TIE = 1, -1, 0
6
7     def __str__(self) -> str:
8         match self:
9             case WinOrLose.WIN:
10                 return "Win"
11             case WinOrLose.LOSE:
12                 return "Lose"
13             case WinOrLose.TIE:
14                 return "Tie"
15             case _:
16                 raise ValueError
```

Let's start from the "top", the main script *rps3/main.py*. The main script (which we will use to run the rock paper scissors program) is essentially a one-liner. It creates an instance of a custom type `Game`, e.g., by "calling" it, `Game()`, and calls a "method", `start()`, on this instance. The type `Game` is imported from a module, `rps.game`.



Even without looking at the rest of the program, we can say this much from this file alone. How? 😊

Since we create an object and use it once, the name is not really necessary. We could have called the `start()` method as follows:

```
game.Game().start()
```

As one can easily guess, all the program logic is contained in the `start` method of the `Game` type.



All user-defined types are mutable by default. Or, more precisely, the Python interpreter treats the user-defined types as mutable. In Python, there is no way to create truly immutable custom types (similar to the builtin immutable types like `int`).

The type `rps.Game` is defined using a Python keyword `class`. For example,

```
11 class Game:
12     ...
```

Or, optionally, the class name can be followed by a pair of parentheses:

```
class Game():
    ...
```

As indicated before, by convention, user-defined types use the PascalCase names (aka UpperCamelCase). Just like functions, there is no such thing as "private" classes in Python. But, one can use the leading underscores to exclude them from the wildcard module import.

The `Game` class, defined in `rps/game.py`, includes a number of *methods* (e.g., the functions defined *within* a `class`, *indented*) and a few other *data attributes* (which you cannot easily see from the structure alone). The "main" method, `start`, calls a few internal, or *private*, methods (again, "private" only by convention), of which the `_loop` method (lines 62-88) includes the main rock paper scissors game logic. This method essentially follows the same logic that we used in the previous two programs.

This new rock paper scissors game implementation uses some Python features that we have not used before, besides `class` and its related OOP features.

Most notably, it uses the new `match` statement (new as of the 3.10 release), which is more or less comparable to the `switch` statement in C, and some other pattern

14.1. Modules and Packages

matching expressions/statements in other modern programming languages. It took over 30 years for Python to finally adopt this feature. 😊

Furthermore, we use Python's "enums" in this new version (e.g., `Hand` and `WinOrLose`), which replaces the string constants that we used in the previous implementations. We also introduce the important concept of the "Boolean context". All expressions in Python have their "truth values", and it is one of the unique characteristics of Python compared to other (C-style, C-descendent) programming languages. (But, the C language itself is an exception. 😊 In fact, Python is somewhat similar to C in this regards.)

Finally, although it is not the core part of this new program, we will also discuss the "union types" in this lesson, which is again a relatively new addition to the Python typing system. The commonly used `Optional` type can be viewed as a special case of the union types.



It goes without saying that Python has been influenced by many other programming languages. Although Python is not strictly a C-style language, and it has a number of fundamental differences from C, in terms of function calling conventions, block scoping, etc., one can still see C's (huge) influence in various aspects of Python. Python is also most likely influenced by Unix Shell, Perl, C/C++, Java, and other functional programming languages like Haskell. We will not discuss much of the functional programming styles in Python in this book, however.

On the flip side, Python has influenced many other programming languages as well. Most notably, the (newer) languages like Go, Rust, and Julia, among others.

14.1. Modules and Packages

In Python, a "module" (e.g., a Python code file) is a fundamental unit for organizing and sharing the code, as we explained in the earlier lessons. A Python module can

include variables, function definitions, and custom type definitions (e.g., classes and enums), among other things, which can be potentially used in other Python programs. Modules are objects, and they can be customized by updating their attributes (which we do not discuss in this book).

A package is a special kind of module, and it is a collection of one or more modules, typically but not always, from a folder in a filesystem. In our new rock paper scissors implementation, we use the `rps` package that includes all rock paper scissors related modules.

Modules, including package modules, and the names defined in those modules, can be imported using the `import` or `from import` syntax, which we have been using from the very beginning of this book. The imported names can be used in our program just like they are declared in the local namespace.

We can also use "aliases" for the imported names. This is done using the Python keyword `as`. For example,

```
>>> import random as rnd
>>> rnd.randint(1, 7)
1
```

In this example, we import the standard library `random` module as `rnd`. And, we use the name `rnd` just like we use the original name `random`. For instance, we call the `rnd.randint()` function in this case. One thing to note is that, since we have imported the random module `as rnd`, the name `random` is not available at this point. That is, you cannot use the module name `random` after importing it `as <some other name>`.

If you want to look up the `randint` function in the help doc, you also do this:

```
>>> help(rnd.randint)
```

14.1. Modules and Packages

Likewise, we can also use "as <alias>" when we import the names within a module via the `from import` statements. For example,

```
>>> from random import choice as pick_one ①
>>> pick_one(["M", "T", "W", "Th", "F"])
'M'
```

① We used the `choice` method in the previous part.

One of the common reasons why we want to use the name aliases is to use shorter or easier-to-user names, e.g., when the original names are too long or too difficult to type, or when the imported names are frequently used, etc. In some cases, some libraries have commonly-used aliases. For instance, it is typical to `import numpy as np` and `import pandas as pd`, etc. (These are two of the most commonly used libraries in data science and machine learning.)

But, more importantly, we often use the name aliases to avoid name collisions. For example, in the program that we are working on, we may have used the name, `choice`, e.g., for our custom function. In that case, we cannot use the same name from the `random` module (without fully qualifying it). The above example illustrates how to avoid the name collisions.



We never explicitly stated it, but one cannot use the same name in Python for two or more different things, in a given scope. It is sort of obvious 😊 since names cannot refer to more than one objects at the same time. Every time a name is bound to an object, it is unbound from the old referenced object, if any.

Unlike in many other C-style programming languages, where the namespaces are separated based on the types or other categories, etc., the names in Python are all in one namespace corresponding to each scope. That is, you cannot use the same name both for a function and for a variable. At the end of the day, everything is just an *object* in Python. 😊

14.2. Backslashes

Python was originally influenced, among other things, by Unix shelling scripting, and there are still some remnants of the shell scripting heritage, including the fact that the Python's comment uses that of shell scripts (`#`), as a trivial example. One other thing is the fact that Python is a line-based programming language, which is most suitable for scripting (e.g., unlike C or other C-descendent languages).

Lines are a critical component of the physical structure of a Python program. For long (physical) lines, one can use brackets so that they can be broken over multiple lines (which was explained in the first part). One other way to do this is to use the Unix line continuation symbol. A backslash `\` plus a newline character indicates that the given line is "continued" to the next line (e.g., in a text file). They essentially form a single physical line.



It can be rather confusing at times, but unfortunately we sometimes use the same English words to mean (slightly) different things. The term *line* in Python often refers to a "logical line". That is, an `if` statement, even if it is written over multiple lines in a program file, is a single *logical line*. When a line is broken over into multiple lines in a file, e.g., using this escape character, backslash + newline, those may still be viewed as a single *physical line*.

You cannot generally see white spaces (although you can do that in certain programmer's editors and IDEs, including VS Code). You cannot easily distinguish a backslash + newline and a backslash + other white space, for instance. Hence, it is generally not recommended to break lines using this syntax.

Here's an example:

```
>>> apple = 1 + \  
... 2 + 3  
>>> apple
```

Here, the statement `apple = 1 + 2 + 3` is really in one physical line as far as the Python language syntax is concerned. This "trick" can be used just about anywhere when it makes sense. You cannot, however, break a single name/identifier into multiple lines this way, for instance.

But you can use this syntax within string literals. For example, you can write a short string literal over multiple lines.

```
>>> "hello \  
... world"  
'hello world'
```

This is just a one-line string as far as Python is concerned, not a true multiline string literal.

We do not use this syntax in any of the examples in this book (except for the above two), and, as stated, this usage is not recommended (unless *absolutely* necessary ☺). For long strings, one can always use string concatenations over a series of short strings that are written in multiple lines (e.g., grouped together using brackets, etc.).

Likewise, this backslash + newline character can also be used to "escape" (real) newlines in the long string literals in Python. Hence, for example, one can effectively use a long string literal to include single line text that spans multiple lines (in the code file).

```
>>> """Hello  
... World and \  
... the Pythonites!"""  
'Hello\nWorld and the Pythonites!'
```

Note that there is no real newline (`\n`) between the `"World and "` and `"the`

`Pythonites!`". Normally, newlines in a long string literal is part of the literal, but they can be escaped away this away.

In the `game` module, we use this syntax to make the string literals more readable, or just a bit prettier ☺, e.g., lines 30-35, 38-42, and 45-51. Without the first backslash + newline in each string, for instance, the text may not have looked completely "aligned". For example, without the backslashes, the `print` function in the `_banner` method (lines 30-35) would have looked like this:

```
print("""-----
Welcome to Rock Paper Scissors!
Type X or Q to end the game.
-----""")
```

14.3. Optional and Union Types

As stated earlier, it is not uncommon for a function in the dynamically typed languages like Python to return values of two different types or more. The same variable can be used to refer to objects of different types. This kind of use cases are relatively rare in the strongly typed imperative programming languages.

The formal type systems (e.g., from mathematics and the functional programming languages) use what is known as the "union types" in such situations. A union type is a combination of two or more types which can be used in alternative circumstances. This is a subset of the broader category known as the "algebraic types".

A tuple type is sort of like a "multiplication", or logical *AND*, of its element types (as in "algebra"). For example, a tuple of `int` and `str` has a type of `int times str`, formally represented as `typing.Tuple[int, str]` in the Python typing framework. Likewise, a union type is sort of like an "addition", or logical *OR*, of multiple types.

A union type in Python typing can be represented by the new 3.10 syntax using the vertical bars, or using the `typing.Union` class. For instance,

14.3. Optional and Union Types

```
screen_size: str | typing.Tuple[int, int] = (860, 640)
screen_size = "FHD"    # ~ (1920, 1080)
```

This annotation `str | typing.Tuple[int, int]` is equivalent to `typing.Union[str, typing.Tuple[int, int]]`.

Unlike in the case of the tuple types, a single item union type is the same as its item type. That is, `typing.Union[int]` is the same as `int`, for instance. Union types are like sets of types: Their orders are not important, and all their element types have to be unique. For example, `int | str` is the same `str | int` as far as typing is concerned

One of the most common (generic) types, which are widely used in the statically typed languages (imperative or functional), is the so called "option", or "maybe", types. The Python typing system has the corresponding type in the `typing` module, namely, `typing.Optional[T]`. This type annotation is equivalent to `T | None`, for any type `T`. That is, for example, a function that returns a type `Optional[str]` may return an object of the string type or none at all.



Again, Python does not have "generics" (or, parameterized types). Generics is only needed in the statically typed programming languages. Through `typing`, however, we are adding some additional layers to Python's dynamic type system so that we can use some static type checking at the "build time" (that is, if we choose to do so). As briefly alluded before, however, we do not recommend the readers to go overboard with complex typing constructs like generics in Python.

BTW, why do we quote the "build time"? ☺ Python does not normally require "building" (or, "compiling", etc.). Everything is done at run time. However, you can always use other build tools during the development, such as the static type checkers.

In the third incarnation of our rock paper scissors program, the `read_hand` function of the `reader` module (lines 5-17) returns `Optional[Hand]`, where `Hand` is a custom `enum` type as we will see shortly.

```
5 def read_hand() -> typing.Optional[Hand]:
6     ...
```

This type annotation indicates that the `read_hand` function may return a valid value of type `Hand` in certain situations, or it may not return any value in certain other situations. In this example, when we catch any `EOFError` or `KeyboardInterrupt` exception, we just return `None` from this function (lines 13-14). (Note that `return`, without any argument, is equivalent to `return None`.)

```
def read_hand() -> Optional[Hand]:
    while True:
        try:
            ...
        except (EOFError, KeyboardInterrupt):
            return
    # ...
```

On the contrary, in the implementation of the `_parse_hand` function (lines 20-29), we just raise an exception in certain situations, e.g., instead of returning `None`. This is mostly an (API) design decision.

```
def _parse_hand(s: str) -> Hand:
    match s:
        # ...
        case _:
            raise ValueError("Invalid hand")
```

In this example, we annotate the `_parse_hand` function to return `Hand`. This is

because the function returns a value of the `Hand` type in all normal return cases. We do not generally annotate the exceptions in Python.

14.4. Boolean Context

We have used the `if` statements before. In fact, a lot. ☺ As indicated, a Boolean, or logical, expression follows the `if` keyword. Likewise, a Boolean expression follows each `elif` keyword as well.

In the sample code of this lesson, the `_loop` function of the `game` module uses an `if` statement, lines 65-66. This `if` statement includes a `not` expression on line 65.

```
64 player_hand = reader.read_hand()  
65 if not player_hand:  
66     return
```

As briefly alluded earlier, Python is rather different from many modern programming languages when it comes to handling Boolean values. (As stated, a Boolean or logical value represents a binary state, either true or false.) In Python, `bool` is a subtype of `int`, a numerical type. Furthermore, any expression in Python can be potentially interpreted as a Boolean value depending on the context.

For instance, the `if` statement expects Boolean expressions in certain places (e.g., right after `if` or `elif`) as stated. Likewise, the Boolean operators, `not`, `and`, and `or`, in Python expect Boolean expressions as their arguments.

In the example, however, the argument of the unary `not` operator, `player_hand`, is not a Boolean expression. In fact, it is of the `Hand` type or `None`. (as we can see from line 64, and the type annotations of the `reader.read_hand` function, line 5 of `rps/reader.py`).

In Python, because Boolean expressions are expected in these places, they are interpreted as Boolean values rather than other types. This is known as the "boolean context" in Python. Any expression in those places will be evaluated to either `True`

or `False` (even if it is not a `bool` expression). This is called the "truth value" of the expression.

The "boolean context" is one of the most important concepts in Python, especially for beginners and the people with experience with some other programming languages. Even a `list` in Python, for instance, can have a Boolean value depending on the context. The truth value of an empty list is `False`. All other non-empty lists are evaluated to `True` in the Boolean context.

Here are some examples:

```
>>> "I'm true" if 555 else "I'm false"
"I'm true"
>>> "I'm true" if 0.0 else "I'm false"
"I'm false"
>>> "I'm true" if "false" else "I'm false" # trick question :)
"I'm true"
>>> "I'm true" if "" else "I'm false"
"I'm false"
>>> "I'm true" if (False,) else "I'm false" # another trick question
"I'm true"
>>> "I'm true" if () else "I'm false"
"I'm false"
>>> "I'm true" if [0] else "I'm false"      # another one ;) ;)
"I'm true"
>>> "I'm true" if [] else "I'm false"
"I'm false"
>>>
```

Note that the *truth value* of an expression is the same as the value from calling the `bool()` constructor function on that expression.

```
>>> bool(555), bool("false"), bool((False,)), bool([0])
(True, True, True, True)
>>> bool(0.0), bool(""), bool(()), bool([])
```

```
(False, False, False, False)
```



As emphasized before, Python is a loosely typed language, and the "contexts" are important. For example, a certain non-numerical expression may be evaluated to a number in certain contexts. A certain other expression may throw an error instead, however. The Boolean context is universal. *Any* expression can be evaluated to **True** or **False** in the Boolean context.

Just to be clear, the fact that Python is *loosely typed* does not mean that the types are not important. It's quite the contrary. Types are not always obvious in Python (unlike in other statically typed languages), but they are nonetheless critical in Python programs. In fact, many of the runtime errors you will get as a beginner, and even as an experienced Python programmer, are type-related.

As stated, we use the "type annotations" in this book, which can help you avoid some common type-related errors. Consider it as a "training wheel". You may decide to do away with the Python typing after reading this book, or you may decide otherwise. Either way, you will end up paying more attention to the types.

14.5. Recursion

This is a bit of a digression, but here's an interesting exercise: How would you compute the length of a list object without using the builtin **len** function?

One way to do this is to count the items of a given list while iterating over them. We can use the **for** loop for this.

```
def length_by_iteration(seq):
    length = 0
    for _ in seq:
```

```

        length += 1
    return length

```

This is essentially a simpler version of the `sum` function that we implemented in the previous lab. In this case, since there are `len(seq)` items in the list `seq`, the `for` will iterate over that many times. After the loop, the `length` variable will have been incremented by `len(seq)` times.

Note that we use the underscore discard variable `_` since the actual values of the list items are not important. Each of them all counts as `1`.

Another way to do this is to use the "recursion". Recursion is one of the most important concepts in programming. A function, or any of the statements in its body, can call other functions. A function can likewise call itself, either directly or indirectly. This is called the recursion. Typically, a function that directly calls itself in its implementation is called a recursive function.

Note that if a function calls itself, directly or indirectly, this called function, the same function, will again end up calling itself, and then again, and again. Implementing recursion incorrectly is one of the easiest ways to get a "stack overflow error". ☹

Recursions are one of the core techniques in functional programming. Even in imperative programming, recursions generally play important roles (although they are not as much used in practice). A lot of algorithms can be more naturally, and more intuitively, expressed using recursions.

For instance, the `length` function in the above exercise can be written using recursion. Here's an example:

```

def length_by_recursion(seq):
    if not seq:                                ①
        return 0
    else:                                       ②
        return 1 + length_by_recursion(seq[1:])

```

14.5. Recursion

- ① The "base case". See below.
- ② The "recursive case". Ditto.

This can be written even in one line using the `if` conditional expression:

```
def length_by_recursion2(seq):  
    return 0 if not seq else 1 + length_by_recursion2(seq[1:])
```

It takes a bit of getting used to if you are new to computer programming, but it will eventually become very natural to you. Recursion is a very powerful tool, not just in (functional) programming but as a way of solving problems, in general.

Recursion has a "formula". (Lucky for us, the beginners. 😊) Any recursive algorithm has (1) a "base case", and (2) a "recursive case".

The *general* recursive case deals with a case for size `n` when we know the answer to a bit smaller problem, that is, for `n - 1`. (The "size" can mean many different things.)

For example, what is the sum of all numbers between `0` and `1_000_000`? If, that is, *IF*, we know the sum of the numbers between `0` and `999_999`, then we will know the answer to the original problem. How? If the answer to the smaller problem is, say, `N`, then the sum of the numbers from `0` to `1_000_000` is `N + 1_000_000`. Why? 😊



Again, we are not in a race. 😊 Take your time and make sure that you understand this. (Or, you can skip this section for now, and come back later. We do not use recursion in our rock paper scissors program.)

BTW, we can use the underscores `_` in numeric literals in Python, just like we use commas in numbers in daily life. (At least, in the U.S. This convention may be different in Europe, and elsewhere.) A number literal will still have to start with a digit. If it starts with an underscore, that is not a numeric literal. Python will treat it as a

name/identifier.

The *special* base case deals with a case when the problem becomes so small that the answer becomes trivial. That is, for example, what is the sum of a single number `0`? Or, what is the sum of the numbers from `0` to `1`? The answers are trivially `0` and `1`, respectively.

The base case generally provides the condition for terminating the recursion. If the recursive case is incorrectly implemented and the recursion never reaches the base case, then you will likely end up with an infinite recursion. That is, a stack overflow error. Likewise, if the base case is incorrectly implemented, and it does not terminate the recursion, you can also fall into an infinite recursion.

When we deal with a "list" (or, any sequence type) in programming, it is often convenient to view the list as a combination of the "head" item and the rest (which we generally call the "tail"). For example,

```
seq = [1, 3, 5, 7]
head = 1
tail = [3, 5, 7]
```

Now, adding/prepending `head` in front of the `tail` in some way will result in more or less the same list as `seq` (in terms of their items and values). Note, however, that this division will not work when the list is empty. You will need at least one element in the sequence for this to work.

Through this division (as in "divide and conquer"), we end up with a smaller, or shorter, sequence `tail` (that is, shorter than the original list `seq`) and a single item (or, a sequence of one item), or no item (or, an empty sequence), for which (hopefully) we have a trivial/easier solution.

In the case of the `length_by_recursion` function, defined earlier, the base case is when there are no items in the list. What is the length of a list with no items? The answer is trivially `0`.

If the length of a list with n items is N , then what is the length of a list with $n + 1$ items? That is $N + 1$ since there is one more item in the list.

That is precisely what the example program says. When there is no item in the list, `if not seq`, the length is 0 . For a list *with at least one item*, `else`, its length is 1 more (from the "head") than the length of the list with one fewer items (the slice `[1:]`, the "tail").



In this (trivial) example, N is n . The length of an n -item list is n . But, we are just demonstrating the recursion logic here. 😊

The `if` clause, `if not seq`, includes the Boolean expression `not seq`, which evaluates to `True` if `seq` is empty, e.g., `[]`, because the truth value of an empty list is `False`. Otherwise, `not seq` evaluates to `False` since any non-empty list evaluates to `True` in the Boolean context.

As stated, all Boolean operators treat their arguments as Boolean expressions. (As explained, `and` and `or` behave in a somewhat special way. In what way? 😊) Likewise, the `if` and `elif` clauses in the conditional statements (as well as in the conditional expressions) expect Boolean expressions, and the truth values of those arguments are used.

To summarize the rules of the truth values, from the previous lesson,

- For all numerical type objects, their truth values are all `True` unless they are `0`, `0.0`, or `False`.
- For string objects, the truth value of an empty string is `False`. For all other strings, their truth values are `True`.
- For `None`, its truth value is always `False`.
- For `...` (ellipsis), its truth value is always `True`.
- For lists and tuples, an empty list/tuple evaluates to `False`. All other lists and tuples with non-zero items are evaluated to `True`.
- As we will see shortly, the truth values of all objects of a custom type (e.g., `enum`

or `class`) are `True` by default. But, it can be customized.

Now, let's take a look again at the statements on the lines 64-66 in the `rps.game` module.

```
64 player_hand = reader.read_hand()
65 if not player_hand:
66     return
```

The type of `player_hand` is `Optional[Hand]` since the function `reader.read_hand()` returns the `Optional[Hand]` type, which is equivalent to `Hand | None`. The truth values of the members of the `Hand` enum type are all `True` by default, as just indicated.

The conditional expression `not player_hand`, therefore, can be `True` only if `player_hand` is `None`, that is, if the function call `reader.read_hand()` returns `None`. In such a case, we just `return` from the `_loop()` function, effectively terminating the `for` loop (lines 63-88).

14.6. Object Oriented Programming (OOP)

We saved the best for last. Well, almost the last. ☺ The OOP can help create more modular and more structured Python programs, whose components can be more easily reusable, etc. In many problem domains, thinking in terms of "classes" and "objects" (in the sense of the OOP) can be rather natural and intuitive.

First and foremost, the object oriented programming means "encapsulation". An *object* encapsulates, or hides, certain data and functionalities and it only exposes (a limited part of) its data through well-defined "surfaces", or "interfaces".

A `class` definition is used to create a "template" for objects. A `class` defines a custom type, how to create an object of that type, and how to access the object, among other things.

14.7. Class

A `class` always implicitly "inherits" from a builtin type `object` in Python. This is true for any builtin or custom types. For example,

```
>>> o = object()
>>> type(o)
<class 'object'>
>>> isinstance(bool, object)
True
>>> isinstance(101, object)
True
```

A `class` can inherit from another type (which is in turn a subtype of `object`). Python supports the multiple inheritance. That is, a new Python class can inherit from more than one base classes. We will not discuss the multiple inheritance in this book, however.

14.7. Class

A `class` defines a custom type. This is generally true in many OOP languages, and `class` (including `enum`) is the only way to create user-defined types in Python. A type defines what kind of data that its object/instance can hold and what kind of methods are allowed on that object/instance, etc.

Here's a general syntax:

```
class ClassName(BaseClass):
    pass    # Statements go here...
```

Just like function definitions, a class definition includes a series of statements. In fact, any kind of statements (e.g., including even (mostly useless) expression statements ☺). But, most class definitions primarily include the definitions of variables ("class variables" and "object variables") and functions ("static functions", "class methods", and "object methods").

When a class definition is read/executed by the Python interpreter, these statements are executed. For example,

```
>>> class X(object):
...     if True:
...         print(True)
...     else:
...         print(False)
...
True
```

①

① This is the output of the `if` statement. Unlike function definitions, Python executes all statements in a `class` definition while creating a `class` object.

This may seem rather unusual, or even bizarre, to the people with experience in other OOP languages, but, as indicated, Python has much more flexibility.

A `class` definition is somewhat similar to a "module". These statements are only executed for the first time when the `class` statement is executed (e.g., when the Python interpreter reads the statement). When we *use* the "class object" (e.g., to create an instance object of that class), these statements are not executed. For instance,

```
>>> x = X()
>>>
```

①

① We "call" the class to create an instance of that class. The "function" `X` for class `X` used this way is a *constructor function*. See below.

When the "BaseClass" is `object`, it can be omitted from the class definition. The pair of parentheses is optional when BaseClass is missing. That is, `class X(object): pass`, `class X(): pass`, and `class X: pass` are all equivalent. We use this simplified syntax when we define the `Game` class, line 8 of `rps/game.py`.

When the `class` definition is executed, the Python interpreter creates a *class object*

in memory.



Again, the terminology is a bit confusing, but we are using the term "object" here as in "everything in Python is an object". A function is an object. Likewise, a class is an object.

As with the function definition, a class definition must first be executed and its name be introduced to the program before the class object can be used. A class definition has its own namespace and scope. In particular, any function definitions in a class definition bind their names within this class scope.

14.8. Class Objects

As stated, Python's `class` statement creates a class object in memory (just like the `def` statement creates a function object). A class object supports the "attribute references", e.g., through the dot notation, just like any other Python objects.

```
>>> class SoSimple:
...     one_name = "simple"
...
>>> SoSimple.one_name           ①
'simple'
>>> SoSimple.one_number = 666   ②
>>> SoSimple.one_number
666
>>> dir(SoSimple)               ③
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 'one_name', 'one_number']
```

① The attribute included in the class definition.

- ② You can add any attributes to an object. The super power of the dynamically typed language. 😊
- ③ Note that a class object comes with a number of predefined attributes (similar to the builtin types or function objects), all of which start and end with double underscores (`__`), aka "dunder".

This type of attributes of a class correspond to the static variables (or, static fields) and the static methods in other OOP programming languages.

14.9. Constructors

In addition to the attribute references, a `class` object supports the "instantiation operation". A function in Python can be "called". Likewise, a `class` can also be "called" in Python. (That is, they are "callable", as we briefly mentioned earlier.)

In other OOP languages, a function with the same name as the class, or a function that creates an object of that class, is often called a "constructor". A `class` object in Python is a constructor for the objects of the given class/type.



Unlike in other OOP languages, the constructors in Python are just functions, or more precisely, `callable`'s. For instance, they do not require special operators like `new`.

Let's see an example:

```
>>> s = SoSimple()
>>> type(s)
<class '__main__.SoSimple'>
>>> s.one_name
'simple'
>>> s.one_number
666
```

14.9. Constructors

Note that the instance object, `s`, includes the ad-hoc attribute, `one_number`, as well as `one_name`, which is part of the original class definition. One can add any additional attributes to a given instance object as well:

```
>>> s.one_address = "Playa"
>>> s.one_address
'Playa'
>>> dir(s)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattr__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'one_address', 'one_name', 'one_number']
```

Note that the instance `s` includes more or less the same predefined attributes in the original `SoSimple` class as well as other attributes later added to this class object `SoSimple`, and those specific to the instance object `s` itself.

We can delete an attribute defined in a class or in an instance as well. This can be done using the same builtin function, `del`, which is used to remove a name/alias.

```
>>> del(s.one_number)
>>> dir(s)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
'__format__', '__ge__', '__getattr__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
'one_address', 'one_name']
```

The instance object `s` no longer has the attribute, `one_number`, after the `del()` call.

Note that a class object is not only used as a "template" when creating an instance

object of that class, but they essentially share the same attributes.

```
>>> s = SoSimple()
>>> s.one_name
'simple'
>>> SoSimple.one_name = "not simple any more"
>>> SoSimple.one_name
'not simple any more'
>>> s.one_name
'not simple any more'
```



What is happening here? 😊

Class instantiation (or, instantiating an instance object of a class) can be customized by overwriting the `__init__` method of the class. This method is automatically called, e.g., by the Python interpreter, after an instance has been created.

```
>>> class SoEasy:
...     def __init__(self):
...         self.one_title = "programmer"
...
>>> s = SoEasy()
>>> s.one_title
'programmer'
>>> dir(s)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 'one_title']
```

Note the function signature. The `__init__` method has at least one parameter. The first parameter always refers to the instance object just created. Although the

14.9. Constructors

parameter name is not important to the Python interpreter (just like other function parameter names), we *always* use the name `self` here. This is one of the areas where we cannot just say that it's a preference, or it's just stylistic.

All Python programmers use the name `self` in this context. No exceptions! ☺ The `self` parameter refers to an instance object of the given class/type. Python provides no other ways to indicate this information other than this `self` naming convention.

An instance object for this type `SoEasy` has an attribute, `one_title`, automatically attached to it. This is because we create this name/attribute and attach it to the `self` object in the `SoEasy.__init__` method.

Just like functions, class constructors can also have additional `init` parameters. For instance,

```
>>> class Temperature:
...     def __init__(self, degree: float = 32.0):
...         self.degree = degree
...
>>> t = Temperature(212.0)
>>> t.degree
212.0
```

Note the constructor syntax, `Temperature(212.0)`. Since, in this example, the `degree` parameter has a default value, it can be called without any arguments as well.

```
>>> t = Temperature()
>>> t.degree
32.0
```

14.10. Class Variables

A class object (e.g., defined by the `class` statement) plays two roles, among others. First, as we have seen before, it is the *constructor* for the instance objects of the given class/type. Second, it holds the common variables across all instances of the class. In fact, the class variables are *shared* by all instance objects, as we just mentioned.

```
>>> class Car:
...     brand = "GM"
...
>>> car1, car2 = Car(), Car()
>>> car1.brand, car2.brand
('GM', 'GM')
>>> Car.brand = "Ford"
>>> car1.brand, car2.brand
('Ford', 'Ford')
```

We can also access the class variables from an instance object, using a more explicit syntax.

```
>>> type(car1).brand, type(car2).brand    ①
('Ford', 'Ford')
```

① Both `type(car1)` and `type(car2)` refer to the `Car` class object.

In order to update the class variables from an instance object, we will need to use this explicit syntax:

```
>>> type(car1).brand = "BMW"
>>> type(car1).brand, type(car2).brand
('BMW', 'BMW')
>>> Car.brand
```

14.11. Instance Objects

```
'BMW'
```

As we can see, `Car.brand`, `car1.brand`, and `car2.brand` all refer to one and the same object. (This is because `car1`, for instance, does not have an attribute named `brand`, and when `car1.brand` is evaluated, it returns `Car.brand`.) If you do the following, however,

```
>>> car1.brand = "Toyota"
```

This creates a new instance variable for the instance `car1`. It simply hides the class variable with the same name if we use the instance attribute reference syntax.

```
>>> car1.brand, car2.brand  
('Toyota', 'BMW')
```

But, its class variable `brand` is still there:

```
>>> type(car1).brand, type(car2).brand  
('BMW', 'BMW')  
>>> Car.brand  
'BMW'
```

14.11. Instance Objects

An *instance object* of a class includes all the attributes defined in the class, and it can include other instance-specific attributes. Attribute references can be used to refer to those attributes of an instance object. There are two kinds of attributes, the data attributes, or fields or variables, and the methods.

For instance, in the previous examples, `t`, `car1`, and `car2` are all instance objects, created by "calling a class object", e.g., `Temperature()` or `Car()`. (Note, however,

that instance objects themselves are not *callable*.)

As mentioned, an instance object, just like everything else in Python, has attributes, namely, the data attributes and the method attributes. Initially, most of its attributes come from its type when it is created. But, as with other kinds of custom objects (including functions, classes, etc.), new attributes can be added to the instance objects.



Therefore, the OOP in Python is somewhat different from other more conventional OOP where the objects of a type all behave exactly the same. As stated, however, you can use Python just like the (statically typed) OOP languages. You will just need to stick to a certain set of rules (e.g., not adding additional attributes, or not deleting existing attributes, etc., after an instance object has been created).

Instance objects have methods that correspond to the functions in a class. All functions that take an instance object as its first argument (e.g., `self`) are, by definition, "methods", and Python allows the object method calling syntax for these functions. For example,

```
>>> class Ship:
...     def fly(self):
...         print("I cannot fly. Only spaceships can fly.")
...
>>> s = Ship()
>>> s.fly()
I cannot fly. Only spaceships can fly.
```

Here, we call the method `fly()` on the instance object, `s`. This is equivalent to the function call:

```
>>> Ship.fly(s) ①
```

I cannot fly. Only spaceships can fly.

① Note the function argument in this call.

In fact, `<instance>.f(...)` is just a "syntactic sugar" for the more normal function call syntax `<class_name>.f(self, ...)`. (Note the difference in the parameter list.) This works as long as the first argument of the function, `self`, is of the given type/class.



Note that the class name prefix with a dot `.`, e.g., `Ship.fly()` is syntactically similar to the way we use the "dotted names" to refer to the names in a module. `s.fly()`, however, is a method syntax. We call the method, e.g., `fly`, on the object `s`.

14.12. Instance Variables

Although we can add any attributes to an instance object in Python, it is conventionally done in the `__init__` method. Then, all instance objects of the class will have the same (but, separate) attributes.

```
>>> class Pet:
...     def __init__(self):
...         self.kind = "dog"
...
>>> pet1, pet2 = Pet(), Pet()
>>> pet1.kind, pet2.kind
('dog', 'dog')
```

In the initializer, the parameter `self` refers to the instance object which has been just created by *calling* the class object. In the case of `pet1`, for instance, `self` and `self.kind` refer to `pet1` and `pet1.kind`, respectively. Likewise, for the `pet2` instance, `self` and `self.kind` refer to `pet2` and `pet2.kind`, respectively.

The attribute `kind` is an instance variable, and it belongs to a specific instance

object, and they are not shared across different instances. In addition, the class object does not have that attribute:

```
>>> Pet.kind
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Pet' has no attribute 'kind'
```

Note the syntax to define an instance variable in a class definition. Instance variables are defined on the `self` object. Hence, as a corollary, in a class definition, you can only define instance variables within an instance method.

These instance variables "belong" to the respective instance objects. Therefore, we can change the values of instance variables without affecting other instances.

```
>>> pet1.kind = "cat"
>>> pet1.kind, pet2.kind
('cat', 'dog')
```



As mentioned earlier, all user-defined types are mutable. All objects of user-defined types are mutable.

14.13. Instance Methods

As we have indicated, the most common statements used in a `class` definition are the `def` statements to define functions. A function defined in a class is an attribute of a class, and it is also an attribute of any instance object of that class. And, it can be used with the method syntax on the object as long as the type of its first argument is of the same class/type.



It is worth repeating that all attributes defined in a class are both the attributes of a class object and those of an instance object. All

14.13. Instance Methods

these attributes are shared between the class and all of its instance objects. If you need an attribute specific to each instance, then, in the case of data attributes, they need to be defined on an instance, `self`. In the case of functions, they need to take an instance, `self`, as the first argument.

Unlike the class variables, the class functions have two kinds, besides the instance methods: One that is just a function (except for the dotted name syntax), and the other which is a part of a class object and which can access the class variables. They are called the "static methods" and "class methods" in Python, respectively. One can use "decorators" like `@staticmethod` or `@classmethod` to make the distinction clearer, among other things. These are, however, beyond the scope of this lesson. They are less frequently used than the instance methods.

```
>>> class RingBearer():
...     def __init__(self):
...         self.ring = "one ring to rule them all"
...     def protect_ring(self):
...         print(f"protect the {self.ring}")
...
>>> r = RingBearer()
>>> r.protect_ring()
protect the one ring to rule them all
>>> RingBearer.protect_ring(r)
protect the one ring to rule them all
```

As we can easily verify, the method call syntax `r.protect_ring()` is equivalent to the class function call syntax `RingBearer.protect_ring(r)`. As stated, for functions that we use as instance methods, we *always* use the name `self` for their first arguments, which are the instance objects. Likewise, we do not use the name `self` for anything else. (Note, however, that `self` is not a Python keyword unlike `this` in other OOP languages.)

As far as the type annotation is concerned, we do not annotate the `self` variable since it is really superfluous from the context as long as we follow the convention.



Again, Python is extremely flexible compared to other programming languages. It is important to stick to certain commonly used conventions, however.

14.14. Private Members

In Python, there is no real "data hiding". There is no "private" attributes, data or methods, for an instance object. One might argue that Python is not a true object-oriented programming language since it does not support one of the most fundamental requirements of OOP.

But, that is not entirely true. As with many other things in Python, we follow certain well-defined conventions or rules, and as long as *everybody* sticks to those conventions, there is really no problem.

In Python, a name prefixed with an *underscore* `_` is treated as sort of "private". That is, by convention, we do not directly access the members of other class objects or instance objects if their names start with one or more underscores. They are considered an implementation detail, and they are not part of the "public API".

Again, this is a gentleman's agreement (or, more like a gentleperson's agreement ☺), and if you insist on accessing the "private members" of other class or instance objects, Python will let you. Ultimately, however, if you do that, then your code may end up becoming "fragile" in the long run. (For example, your program might "break" if somebody changes their implementations of the library that you are relying on.)

An exception, and a legitimate use case, is when you are using your own classes. Although you may want to "hide" some variables and methods in a class from the "outside world", it might be still useful to be able to use them in other classes in the same program/package. Again, Python lets you do that. It is completely up to you.

14.15. Dunder Attributes

In the `rps.Game` class, the `_win_or_lose` function is a *private/internal* static function, and all other methods are *private/internal* except the `start` method. All the instance variables are also *private/internal*.

Python *does* have some minimal support for name hiding, however. When a name of a variable, a method, or a function in a class *starts with at least two underscores and ends with at most one underscore*, then Python modifies the name. It is called the "name mangling" although it does not truly "mangle" the names (e.g., as in C++). Regardless, using the mangled names should be avoided, even within your own programs. (Note that the "dunder names" are not mangled, or modified, since they end with two underscores.)

14.15. Dunder Attributes

The `__init__` method is a dunder method, as indicated. This particular method is used to provide any initialization code for the newly instantiated instance object.

In the case of `rps.Game`, we use this method to create and initialize four instance variables, `_wins`, `_losses`, `_ties`, and `_num_rounds`. Lines 17-21.

```
17 def __init__(self, num_rounds = MAX_ROUNDS):
18     self._wins = 0
19     self._losses = 0
20     self._ties = 0
21     self._num_rounds = num_rounds
```

Its base type `object` has the following attributes:

```
>>> dir(object)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__',
 '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__',
 '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
```

```
'__str__', '__subclasshook__']
```

Since every type inherits, either directly/implicitly or indirectly, from this base type `object`, these attributes are always available to all types, builtin or user-defined. Some of those data attributes might be empty, and some of those method attributes might have empty implementations.

The `__doc__` attribute stores the docstring of the type, if any. Otherwise, it is an empty string. The `__init__` method is automatically called on the newly created instance object, as we just indicated. By default, this method is a no-op operation unless it is explicitly implemented in a particular subclass.

The `__str__` method is used when an object is used in a string context.

```
def __str__(self) -> str:
    ...
```

This method is similar, for instance, to the `toString` methods in other programming languages. In the rock paper scissors program version 3, both `Hand` and `WinOrLose` enum classes, which we will discuss shortly, have the `__str__` method defined. (Lines 7-16 in the `rps.hand` module, and lines 7-16 in the `rps.result` module.)

For example, for the `result.WinOrLose` enum type,

```
7 def __str__(self):
8     match self:
9         case WinOrLose.WIN:
10             return "Win"
11         case WinOrLose.LOSE:
12             return "Lose"
13         case WinOrLose.TIE:
14             return "Tie"
15         case _:
```

```
16         raise ValueError
```

We can, and we should in many cases, "override" some of these dunder methods to customize the behavior of our custom class. For example, methods like `__eq__`, `__ne__`, `__ge__`, `__gt__`, `__le__`, and `__lt__` are used to customize the equality and comparison-related behaviors of the custom types.

For other dunder methods, and data attributes, the readers will learn and get used to many of them, over time.

14.16. Inheritance

Another salient feature of the OOP is the type inheritance. We have been using the inheritance all along. As stated, every type in Python, builtin or user-defined, inherits from, or is a subtype of, `object`.

To define a class that inherits from a subtype of `object`, we specify the base class (or, the "parent class") in the parentheses following the class name. For example,

```
class JustInt(int):
    pass
```

```
>>> h = JustInt(5)
>>> h
5
>>> h * 2
10
```

The new type `JustInt` inherits from the builtin type `int`. It is a subtype of `int`. It is just like `int` since we did not change the behaviors inherited from its parent class, `int`. We can use `JustInt` just about anywhere `int` can be used.

We can also "override" any of the methods defined in the `int` type. For instance,

```
class SpecialInt(int):
    def __str__(self):
        return f"I'm a special {super().real}"
```

```
>>> s = SpecialInt(10)
>>> s
10
>>> print(s)
I'm a special 10
```

①

① The value of `s` is `10`.

The `print` function takes a string argument, and hence `s.__str__` is (automatically) called when the variable `s` of type `SpecialInt` is used in the string context.

Let's try a *slightly* more complicated example.

```
class Pet:
    def __init__(self, name = ""):
        self.name = name
    def __str__(self):
        return f"My name is {self.name}."
    def bite(self):
        print("I do not bite.")
```

We define a new type `Pet` using the `class` keyword. This `Pet` type inherits from the base class, `object`, and it overrides the `__init__` and `__str__` methods.

"Overriding a method", in Python, simply means redefining the method of its parent class in the derived class. Same as "overwriting". There is really no concept of "

14.16. Inheritance

virtual" vs "non virtual" methods in Python. You can always "overwrite" virtually any attributes that have been previously defined. It matters little, as far as application programming is concerned, whether the attribute comes from one of its parent classes or it is an intrinsic part of the class.

In Python, as explained in the beginning of this book, the last object bound to a given name is the object that name refers to. If you define a function multiple times with the same name, then the last name/definition "wins".

For example,

```
class X:
    def a():
        print('a')
    def a():
        print('A')
```

This is a valid Python program. If you call `X.a()`, then it will print out `A`. A lot of people, especially those with experience in other programming languages, might be in the panic mode at this point. ☹️ This will cause a compile error, no doubt, in most statically typed languages. And yet, it *works* in Python. 😊

This program is, fundamentally, no different from the following:

```
a = 10
a = 20
```

After executing this program (e.g., in Python REPL), the value of `a` is now `20`. This is a perfectly valid program in Python as is the previous one. The last binding overwrites the previous one, if any.

Now, going back to the `Pet` example, although the `__init__` and `__str__` methods are already defined (e.g., inherited from the base class `object`), we define them again with the same names, and the names now refer to the newly defined methods.

As stated, the OOP in Python is rather different from that of the more typical OOP in other programming languages like C++, Java, and C#.



If you are not familiar with the terms like "virtual" and "overriding", etc. in OOP, do not worry. Those concepts do not exist in Python. (Or, in other words, all methods in Python are *effectively virtual*, and the distinction between virtual and non-virtual has no meaning.)

We may use the same or similar terms, but they may have different meanings. As stated, if we say "overriding" in Python, for example, it simply means that we are redefining a method which is inherited from its parent class.

The `Pet` class includes another method called `bite`. Note that all three methods are "instance methods" as we can easily tell from the `self` arguments. (And, there are no other clues.)

Let's try using this `Pet` class in the REPL:

```
>>> p = Pet("python")
>>> p.name
'python'
>>> p.bite()
I do not bite.
>>> print(p)
My name is python.
```

It works as expected. It has `name` (an instance variable) and it has `bite` (an instance method). They work as expected. In the string context (e.g., as an argument to the `print` function), the "correct" method `__str__` is called. (This is because, as far as the `Pet` class is concerned (either class object or instance object), there is only one method with that name.)

14.16. Inheritance

```
>>> dir(Pet)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__',
 '__format__', '__ge__', '__getattr__', '__gt__', '__hash__',
 '__init__', '__init_subclass__', '__le__', '__lt__', '__module__',
 '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
 '__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__',
 'bite']
>>> dir(p)
['__class__', ... '__weakref__', 'bite', 'name'] ①
```

① Removed the middle part for brevity.

Note the small difference in their attribute lists. The attribute `self.name` exists only in an instance object, and not in the class object. But, the function or method, `bite`, is found in both kinds of objects. As explained, this function can be called from both class object and instance object, but they are different in the way we call them. (That is, the function syntax vs method syntax.)

Now, let's define a new type `Cat` as a subclass of `Pet`:

```
class Cat(Pet):
    def __init__(self, name, lives):
        super().__init__(name)
        self.lives = lives
    def __str__(self):
        return f"Hi, I'm {self.name} the Cat. I have {self.lives} lives."
```

The `Cat` class includes everything that `Pet` has, e.g., by "inheritance".

```
>>> dir(Cat)
['__class__', ... '__weakref__', 'bite']
```

The attribute list is exactly the same as that of `Pet` since we did not add anything

new. Instead, we did overwrite, or replaced, the `__init__` and `__str__` methods with new implementations.

Note the implementation of `Cat`'s initializer. In the `__init__` implementation in the derived class, we (almost always) call the base class's initializer method,

```
super().__init__(name)
```

In this particular example, the `name` instance variable is initialized in `Pet`'s initializer, and hence we need to call it from `Cat`'s `__init__`. Otherwise, this variable will not be properly (as intended) initialized.

The call `super()` refers to the base class. Alternatively, we could use more explicit syntax: `super().__init__(name)` is equivalent to `Pet.__init__(self, name)`.



In this case, the distinction between the constructor and the initializer is not important.

The `Cat` class defines an additional instance variable, `self.lives`, which needs to be provided when we call the `Cat`'s constructor (e.g., the callable class object). Here's an example:

```
>>> c = Cat("Garfield", 9)
>>> c.name
'Garfield'
>>> c.lives
9
>>> c.bite()
I do not bite.
>>> print(c)
Hi, I'm Garfield the Cat. I have 9 lives.
```

They all seem to work "as expected". Or, do they? ☹

There are a number of different ways to look at how the "inheritance" works in Python.



In one interpretation, when we call `bite()` on the instance object `c`, the method is not defined in the object itself. Hence, Python next searches for the method in its base class `Pet`, and then its base class (if any), and so on, until it finds the named method. In this example, `c.bites()` effectively ends up calling the `bite` method defined in `Pet`.

But, conceptually, it is a lot simpler to think that every method available for an instance of a derived class (including those from its ancestor classes) is defined in the derived class itself rather than going through the class hierarchy. In this interpretation, the fact that all methods in Python are sort of "virtual" is almost trivial since at the point of execution, every method is available on the object itself.

```
>>> dir(c)
['__class__', ... '__weakref__', 'bite', 'lives', 'name']
```

The instance object, `c` of the type `Cat`, has one more instance variable than the instance object, `p`, of the base type `Pet`. Namely, the new data attribute `lives`, which has been added in the `Cat`'s `__init__` method.

Let's try one more example. `Dog` this time, which is another `Pet` subtype.

```
class Dog(Pet):
    def __str__(self):
        return f"I'm a dog, and people call me {self.name}."
    def bark(self):
        print("Woof woof")
    def bite(self):
```

```
print("Barking dog never bites? Try me. ;)")
```

```
>>> d = Dog("Scooby Doo")
>>> d.name
'Scooby Doo'
>>> d.bark()
Woof woof
>>> d.bite()
Barking dog never bites? Try me. ;)
>>> print(d)
I'm a dog, and people call me Scooby Doo.
```

Again, all work "as expected". The `self.name` attribute is defined in the base class, `Pet`, as an instance variable. Hence we can use `d.name`. The new `bark` method is defined in this new `Dog` type, and when we call `d.bark()`, this method is called. `Dog`'s `bite` overrides, or overwrites, `Pet`'s `bite`, and hence when we call `d.bite()`, the one defined in `Dog` "wins". (Or, as stated, in our interpretation, `Dog` has only one method named `bite`. The base class's `bite` method object may exist in memory, but it can no longer be referred to as the name `bite` from a `Dog` instance.) The `print` function likewise uses the `Dog`'s `__str__` method, and not the one defined in `Pet`.

The `Dog` class now has one more attribute than `Pet`, an instance function, `bark`:

```
>>> dir(Dog)
['__class__', ... '__weakref__', 'bark', 'bite']
```

The same with the instance `d` of type `Dog`. It has one more method `bark` than its counterpart, e.g., `p` of the parent type `Pet`:

```
>>> dir(d)
['__class__', ... '__weakref__', 'bark', 'bite', 'name']
```

14.16. Inheritance

We have been using the builtin `issubclass` function in this book. This function takes two arguments of the type, `type`, and it returns `True` if the first argument is a subclass of the second argument. (Yes. Even `type` is an object in Python. 😊)

```
>>> type(Pet), type(Cat), type(Dog)
(<class 'type'>, <class 'type'>, <class 'type'>)
```

```
>>> issubclass(Pet, object), issubclass(Pet, Pet)
(True, True)
>>> issubclass(Cat, Pet), issubclass(Cat, object)
(True, True)
>>> issubclass(Dog, Pet), issubclass(Dog, object)
(True, True)
>>> issubclass(Cat, Dog), issubclass(Dog, Cat)
(False, False)
```

Likewise, we have used the builtin `isinstance` function before. This function takes two arguments, and it returns `True` if the first argument is an instance object of the second argument.

```
>>> isinstance(p, Pet), isinstance(p, object)
(True, True)
>>> isinstance(p, Dog), isinstance(p, Cat)
(False, False)
>>> isinstance(c, Cat), isinstance(c, Pet), isinstance(c, object)
(True, True, True)
>>> isinstance(d, Dog), isinstance(d, Pet), isinstance(d, object)
(True, True, True)
>>> isinstance(c, Dog), isinstance(d, Cat)
(False, False)
```

Note that the return value of the `isinstance` function call is based on the object's `__class__` attribute as well as the type relationship between the classes involved.

```
>>> p.__class__, c.__class__, d.__class__
(<class '__main__.Pet'>, <class '__main__.Cat'>, <class '__main__.Dog'>)
```

Hence, we can change the type of any (mutable) user-defined object. This is probably *unthinkable* in the statically typed programming languages, but this is *Python*. 😊 For example,

```
>>> p.__class__ = Cat
>>> isinstance(p, Cat)
True
```

Now, the object `p` is an instance of `Cat`. To be complete, we will need to add the missing attribute, `lives`, to `p`. Otherwise, it would not behave exactly like `Cat` (even though it says that its type is `Cat`).

```
>>> p.lives = 1
>>> p.bite()
I do not bite.
>>> print(p)
Hi, I'm python the Cat. I have 1 lives.
```

The concept of "type" in the dynamically typed languages is rather different from that in the statically typed languages. The "type" has any real meaning only at run time in the languages like Python.

14.17. Polymorphism

We saved the best for last. Did we already say that? 😊 If you are still here, congratulations. You just earned our respect. 🙌 We are covering somewhat advanced (but, still very important) subjects. It usually takes a beginner Python programmer several years to understand, and use effectively, Python's OOP.

14.17. Polymorphism

As we have seen throughout the few earlier lessons, the OOP in Python is somewhat "unorthodox". Despite all that, you can do *real OOP* in Python, as long as you stick to the conventions. As far as the "class-based polymorphism" is concerned, however, *there is no such thing as polymorphism in Python*. ☹️ Now, you can skip this lesson. Seriously. 😊

There are a few different kinds of polymorphisms.

In the statically typed OOP languages, an object declared as a certain type at compile time may turn out to be a different (but, nonetheless related) type at run time. This is possible in the languages like C++, Java, and C#, only through inheritance (and, using pointers/references). This is generally called the inheritance-based polymorphism.

In Python, and in other dynamically typed languages, types are "malleable", so to speak. An object of one type can be made to another (possibly completely unrelated) type at run time, as we have seen with a simple example above.

Let's take a look at an example of "non-polymorphism" in Python:

```
>>> for pet in (c, d):
...     pet.name
...     pet.bite()
...     print(pet)
...
'Garfield'
I do not bite.
Hi, I'm Garfield the Cat. I have 9 lives.
'Scooby Doo'
Barking dog never bites? Try me. ;)
I'm a dog, and people call me Scooby Doo.
```

In this example, the types of `c` and `d` are `Cat` and `Dog`, respectively. What is the type of the loop variable `pet` then? In the statically typed languages, if we had a similar program like this using the comparable syntax, it has to be a common type of `Cat` and `Dog`, that is, `Pet` in the example that we have been using. Although the variable

is declared as an object of type `Pet` (e.g., as the loop variable in the comparable `for` loop), at run time they behave like `Cat` or `Dog` based on their actual types.

This is an example of the "inheritance-base polymorphism". (This is true for the language like C++, Java, and C#. Other statically typed languages, e.g., most notably Haskell, support even more flexible polymorphisms.)

In Python, on the other hand, the type is really determined at run time. At run time, when the name `pet` refers to `c` in the tuple, its type is `Cat`, and when `pet` refers to `d` in the next iteration, its type is `Dog`. As emphasized before, in Python, it is not the variable but the object that a type is associated with.

It is, essentially, no different from the following example:

```
pet = Cat("Cheshire Cat")
pet = Dog("Beethoven")
```

The name `pet` can refer to anything. There is no "polymorphism". Or, in other words, everything is "polymorphic". Python does *not* need this kind of polymorphism because it is much more flexible to begin with compared to other statically typed languages.

Generics is also a form of polymorphism. An implementation can be provided for a set of different/related types rather than for a specific type. This is a must in the statically and strongly typed languages. Otherwise, you will end up having to provide (essentially) the same implementations for different types, e.g., one for `int`, and one for `float`, etc.

Python, on the other hand, does not need generics since, as stated, the types in Python is "malleable" and "dynamic".

In the above `for` loop example over `c` and `d`, as long as the object has two attributes, `name` and `bite()`, it can be included in the sequence. Their particular class names are not important.

14.17. Polymorphism

For example, we can create an object of an arbitrary type,

```
>>> class X: pass
...
>>> x = X()
>>> x.name = "no name"
>>> def bite(): print("I am not an animal")
...
>>> x.bite = bite
>>> type(x)
<class '__main__.X'>
```

As long as this object, `x`, supports the attributes used in the loop, for instance, it can be included there:

```
>>> for pet in (x,):
...     pet.name
...     pet.bite()
...     print(pet)
...
'no name'
I am not an animal
<__main__.X object at 0x7f5d462e7790>
```

The type `X` has no relation to the types `Pet`, or `Cat` or `Dog`. And, it still *just works*.



The type systems like this are often called the "duck typing" (as in "if it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck"). Python's support for types, e.g., the `__class__` attribute, the `issubclass` and `isinstance` functions, and the class inheritance, provide some convenience, but they are not essential.

On the other hand, however, if you use the static type checking, e.g.,

using the tools like *mypy*, then you will need to think about this kinds of (advanced) type support features, including generics and inheritance-based polymorphism, etc.

The OOP support in Python may be called the "duck OOP". ☹ It looks like OOP, and more or less it behaves like OOP, and hence it is likely an OOP.

14.18. Truth Values

All objects (and expressions) have the "truth values", as explained before. They yield a Boolean value, **True** or **False**, in the *boolean context*.

The truth value of an object of a custom type is always **True** by default, regardless of the values of their attributes. In order to give a different behavior, e.g., having either **True** or **False** based on their internal states, we will need to implement the `__bool__` method. This method, by default, does not exist in a custom type (as can be seen from the attribute list of **Pet** above).

If an object, or a type, implements this method, then it is called in the boolean context, e.g., by the Python interpreter. For example,

```
class BiggerThan8:
    def __init__(self, zzz):
        self.zzz = zzz
    def __bool__(self):
        return True if self.zzz >= 8 else False
```

```
>>> z = BiggerThan8(5)
>>> bool(z)
False
>>> z = BiggerThan8(10)
>>> bool(z)
```

```
True
```

As we have seen before, the builtin `bool` function returns the Boolean value of the argument (object or expression, or name). This code provides another example of the "duck typing". As long as the object `z` has the method `__bool__`, it works as expected. The precise type of `z`, and its name, is irrelevant.



This is but one example of customizing the behavior of a type by implementing/overriding one of the special/dunder methods.

14.19. Enum

An `enum` is a collection of names bound to constant value objects. These objects are called the members of the enum, and they all have to be unique. The values of the enum members can be of any type, but immutable types such as `int` or `str` are typically used. Although they can be of mixed types (e.g, a Union type), it is generally recommended to use a single type for uniformity.

An `enum` defines a new custom type just like a `class`. An `enum` is generally used to "group" related constants under a single type. Each member has a name and a value. The members of an `enum` type can be iterated over just like an object of a sequence type.

An enum can be created either with the class syntax or the functional syntax. We use the class syntax in our rock paper scissors program v3. Although we use the class syntax, the enum types are a bit special. They are not normal classes. For instance, we cannot create a new instance of an enum type. The enum members are "singletons" (or more precisely, "multitons") of the given enum type, and they all have to be created in the enum definition.

Here's an example:

```
from enum import Enum
```

```
class Color(Enum):
    RED = "red"
    GREEN = "green"
    BLUE = "blue"
```

You create a new `enum` class by inheriting from the `enum.Enum` type. (Refer to other references for the functional syntax, if interested.)

The class variables defined in an enum class are the (singleton) instances of the enum type. Since they are constants, we follow the all-caps naming convention.

```
>>> Color
<enum 'Color'>
>>> type(Color.RED)
<enum 'Color'>
>>> Color.RED
<Color.RED: 'red'>
```

Note that the type of an enum member (e.g., `Color.RED`) is not the type of its value (e.g., `"red"`). Its type is the same `enum` type, e.g., `Color` in this example.

```
>>> isinstance(Color.RED, Color)
True
>>> issubclass(Color, Enum)
True
```

Furthermore,

```
>>> dir(Color)
['BLUE', 'GREEN', 'RED', '__class__', '__doc__', '__members__',
 '__module__']
>>> dir(Color.RED)
['__class__', '__doc__', '__module__', 'name', 'value']
```

14.19. Enum

```
>>> Color.__members__  
mappingproxy({'RED': <Color.RED: 'red'>, 'GREEN': <Color.GREEN: 'green'>, 'BLUE': <Color.BLUE: 'blue'>})  
>>> Color.RED.name, Color.RED.value  
('RED', 'red')
```

Note that each instance, e.g., `Color.RED`, has attributes, `name` and `value`, which its type, `Color`, does not.

We can iterate over the enum `__members__`, e.g., using the `for - in` statement. For instance,

```
for c in Color:  
    print(f"{c}:\t{c.value}")
```

```
Color.RED:      red  
Color.GREEN:    green  
Color.BLUE:     blue
```

One other "limitation" of an `enum` type is that you cannot use an enum type as a base class for other type, including another enum type. If the type inheritance is important, then you will need to use the normal `class` to create new types.

In the rock paper scissors program version 3, we define two `enum` types:

```
4 class WinOrLose(enum.Enum):  
5     WIN, LOSE, TIE = 1, -1, 0  
6     ...
```

```
4 class Hand(enum.Enum):  
5     ROCK, PAPER, SCISSORS = "r", "p", "s"
```

6 ...

Both enums have the `__str__` methods defined, which we discussed earlier. In these functions, we return different string values based on the enum members (e.g., the `self` argument). We could have done the same using the enum's `names` or `values`.

We will discuss the Python's `match` statement next, which is used in the implementation of both `__str__` methods.

14.20. Match Statement

We had to wait for over 30 years for this. 😊 Python's new `match - case` statements can now be used where complex `if - elif - else` statements may be required. The `match` statements are generally easier to read, and they tend to convey the program logic, and the programmer's intention, more clearly. In certain situations, `match` statements can be more efficient than the corresponding (long) `if - elif - else` statements.

The `match` statement can be used like the traditional `switch` statement in C. But, more generally, it can be used for the "structural pattern matching", or just "pattern matching" for short, which most, if not all, "modern programming languages" support. (Finally, Python joined the ranks of the *modern languages*. 😊)

In the simplest usage, it is more or less equivalent to C's `switch` statement. For instance, using the color `enum` example above,

```
>>> color = Color.BLUE
>>> match color:
...     case Color.RED:
...         print("Color red detected")
...     case Color.BLUE:
...         print("Blue wins!")
...     case _:
...         print("We don't care about other colors :)")
```

14.20. Match Statement

```
...  
Blue wins!
```

There are a few things to note. First, Python's `match - case` is a statement and not an expression. For example, you cannot use the `match` statement on the right hand side of an assignment.

Second, the `case` clauses do not require `break` statements, unlike in C's `switch - case` statement. They do not "fall through". (This is similar to Go's `switch` statement.)

Third, the wildcard expression `_` is used for "catch all". That is, the (optional) `case _` clause is more or less equivalent to the *default* case in the C-style switch statement. If there is no matching `case`, then the `match` statement becomes no-op. If the catch-all case `case _` is specified, then the statements within this default case are executed.

And, at least one `case` clause is required, which can be the catch-all default case. That is, for instance, we cannot just use a placeholder like the `pass` statement in the body of the `match` statement.

Since there is no default "fall through" behavior, we use the vertical bars `|` to separate multiple matching expressions. They represent the alternative cases (as in `or`). For example,

```
>>> day_of_week = "Wednesday"  
>>> match day_of_week.lower():  
...     case "saturday" | "sunday":  
...         print("It's a weekend!")  
...     case _:  
...         print("Today is just another day :(")  
...  
Today is just another day :(
```

If `day_of_week` is either "Saturday" or "Sunday", in this example, the first `case`

would have matched, and it would have printed *It's a weekend!* instead. 😊

In the final version of the rock paper scissors program, we use the `match - case` statements in multiple places (like just about everywhere 😊). These statements could have been written, for instance, using `if` statements (just like all Python programmers have been doing *for the last 30 years* 😊).

For example, the following `match` statement, lines 21-29, in the `def` compound statement of the `rps.reader` module

```

21 match s:
22     case "r":
23         return Hand.ROCK
24     case "p":
25         return Hand.PAPER
26     case "s":
27         return Hand.SCISSORS
28     case _:
29         raise ValueError("Invalid hand")

```

This could have been alternatively written as follows:

```

if s == "r":
    return Hand.ROCK
elif s == "p":
    return Hand.PAPER
elif s == "s":
    return Hand.SCISSORS
else:
    raise ValueError("Invalid hand")

```

They both do the same thing. In these small examples, there is really no difference. Choosing one or the other would be just a matter of style.

14.20. Match Statement

In general, the `match` statements can be used in more limited use cases, but in those cases, they tend to "work" better. As stated, they are more explicit and easier to read. In this small example code, for instance, the `if` and `elif` clause sequence could have included any boolean expressions, and we need to go through each case from top to bottom to understand the code. On the other hand, the case clauses, following the `match`, are more "uniform", or consistent. It requires less "effort" to understand this code fragment.

The `match` statement can also be used for more complex *pattern matching*. For example,

```
for p in [(0, 0), (0, 2), (3, 3), (2, 4)]:
    match p:
        case (0, 0):
            print("I'm the origin!")
        case (x, 0):
            print(f"I'm special! I'm on the x-axis, x = {x}")
        case (0, y):
            print(f"I'm special! I'm on the y-axis, y = {y}")
        case (x, y) if x == y or x == -y:
            print(f"I'm also special! I'm on a diagonal, ({x}, {y})")
        case (x, y):
            print(f"I'm just a random point. ;(")
```

This `for in` loop prints out the following:

```
I'm the origin!
I'm special! I'm on the y-axis, y = 2
I'm also special! I'm on a diagonal, (3, 3)
I'm just a random point. ;(
```

In this case, the match pattern is a tuple of 2 integers. We use the `for` loop iterating over a list of 4 example tuples. The patterns are tested from top to bottom in sequence, and as soon as there is a match, that matched case, and only that case, is

executed.

For the first item in the list, `(0, 0)`, in this example, the first pattern, the tuple literal `(0, 0)`, matches it exactly. Hence the print statement in that case is executed, printing out *I'm the origin!*

The second item, `(0, 2)`, does not match the first two patterns since the second element of this tuple is not `0`. However, it matches the pattern in the third case, `(0, y)` because the first element `0` matches the first element `0` in the pattern and the second element `2` matches an (arbitrary) int `y`. In this case, the value of `y` becomes that of the matched object, e.g., `2` in this case. Hence, this case clause prints out *I'm special! I'm on the y-axis, y = 2.*

The third item, `(3, 3)`, happens to match the fourth pattern `(x, y) if x == y or x == -y`, but no other patterns before that. This pattern includes a conditional statement, known as the "guard". This adds a constraint to the matching variables. Although the pattern `(x, y)` can match any tuples, in this case, this particular pattern with a guard `(x, y) if x == y or x == -y` only matches when the two elements in the tuple satisfy the specified condition, e.g., if the absolute values of the two are the same.

The last item in the `for` loop, `(2, 4)`, does not match any of these patterns, and hence it goes to the last pattern. Note that the pattern `(x, y)` matches any 2-item tuples, and hence in this particular example, `case (x, y)` acts as the catch-all case, e.g., just like `case _`. But, unlike the default `case _`, the `case (x, y)` "captures" the `x` and `y` values. We do not use these values in this example, however, and we merely print out a sad message: (I'm not special.) *I'm just a random point.* ;(☹



Well, everybody is special. Readers! You are special. Especially, if you are reading this book. ☺

In our program, we use the pattern matching against the tuples of type `typing.Tuple[Hand, Hand]` where `Hand` is an `enum` type, lines 20-26, in the `compare_hands` function definition in the `rps.hand` module.

14.20. Match Statement

```
20 match(h1, h2):
21     case(Hand.ROCK, Hand.SCISSORS) | (Hand.PAPER, Hand.ROCK) |
      (Hand.SCISSORS, Hand.PAPER):
22         return 1
23     case(Hand.SCISSORS, Hand.ROCK) | (Hand.ROCK, Hand.PAPER) |
      (Hand.PAPER, Hand.SCISSORS):
24         return -1
25     case _:
26         return 0
```

Each of the first two case clauses uses three alternative tuple patterns. If the tuple of `(h1, h2)` does not match either of these patterns, then it matches the default case, and the function `compare_hands` returns `0`.

For example, when `h1` is `Rock` and `h2` is `Paper`, it matches the second case (and, its second alternative), and hence the function returns `-1`. And so forth. The readers are encouraged to go through other `match` statements in the program of this lesson, as an exercise.

One thing to note is that in the `__str__` methods of `Hand` and `Result` enums, the last default catch-all cases are not needed, in theory. This is because the first three cases, in both implementations, are "exhaustive". There cannot be any other values for these two enum types.

In some statically typed languages, this kind of situation would have caused a compile time error, e.g., since this code segment is not "reachable". On the other hand, in Python, the type system is rather weak, and "anything can happen". ☺ Hence, this kind of error checking could be useful.

(For example, explicitly raising an exception rather than silently failing, e.g., by returning an invalid value `None` or an empty string `" "`, is often a better decision. If we decide to do so, for instance, then we can even handle the exception in the high-level part of the program.)

Now that we have learned the `match` statement, we can implement our [earlier](#)

`length_by_recursion` function slightly differently. Here's a new implementation, `list_length`:

```
def list_length(seq):
    match seq:
        case []:                                ①
            return 0
        case [_, *y]:                            ②
            return 1 + list_length(y)
```

① The base case.

② The recursive case.

Although we did not go through all the supported pattern types in Python's `match` statement, the pattern can be a list as well. The pattern of the first case, the base case in this recursion, is an empty list `[]`. If the argument `seq` is an empty list, then this case will match, and the function will return `0`, as expected.

Otherwise, it matches the second case. In fact, it will match all lists except for the empty list (which is being handled by the first `case`). The pattern `[_, *y]` will assign the first item of the list to the wildcard variable `_` and "the rest" to the list variable `y`. (Note the `*` before `y`.) In functional programming, as indicated, the first element is called the "head", and the rest is often called the "tail".

In this particular function, we ignore the head, e.g., since its count is always `1` regardless of its value, and we recursively call the `list_length` function to the tail list, e.g., `y`, which has one less element than the original argument, `arg`. In this next call, the tail `y` becomes `arg`. This recursion continues until `y`, and hence `arg` in the next call, becomes an empty list, when the base case is invoked. (That is, no more recursive calls.)

14.21. Dictionary

We haven't used the dictionary type much in this book, but it is one of the most

14.21. Dictionary

important builtin types in Python. A (dictionary), or `dict`, is a key-value pair collection data structure, comparable to "map", "hash", "hashmap", or, "associative array" etc., in other programming languages. Let's try updating the program just so that we can use a dictionary. 😊

In the program, we use four separate instance variables to keep track of the total rounds, wins, losses, and ties (lines 18-21, the `game` module). Here's the `__init__` method (slightly modified):

```
def __init__(self, num_rounds):
    self._wins, self._losses, self._ties = 0, 0, 0
    self._num_rounds = num_rounds
```

Instead of this, let's use one variable of the `dict` type.

```
def __init__(self, num_rounds):
    self.stats = {
        "wins": 0,
        "losses": 0,
        "ties": 0,
        "num rounds": num_rounds,
    }
```

In this case, one instance variable, `self.stats`, is initialized with a "dict literal" (`{...}`). The type of this variable is `typing.Dict[str, int]` in the typing framework. The items in a dictionary is a key value pair (e.g., `"wins": 0`), with the items separated by commas.

We will need to make the corresponding changes, for example, in the `_loop` function,

```
wol = Game._win_or_lose(cmp)
match wol:
```

```

case WinOrLose.WIN:
    self.stats["wins"] += 1
    print("You win!")
case WinOrLose.LOSE:
    self.stats["losses"] += 1
    print("You lose!")
case _:
    self.stats["ties"] += 1
    print("Tie.")

```

Note the augmented assignment statements on the items of the dictionary. As stated, `dict` is a mutable type. `_end_game` needs to be modified appropriately as well (which we will leave as an exercise to the readers).

A dictionary literal, and a dictionary expression, uses a pair of curly braces. For example, `{}` is an empty dictionary literal:

```

>>> {}
{}
>>> type({})
<class 'dict'>

```

You can also create an empty dictionary using the `dict()` constructor function.

```

>>> dict()
{}

```

As we have seen before, all builtin type constructor functions work more or less the same way:

```

>>> int(), float(), bool(), str(), tuple(), list(), set(), dict()
(0, 0.0, False, '', (), [], set(), {})

```

14.21. Dictionary

Note that the `set` type, which is another builtin type in Python, is somewhat different. It does not have a simple literal for an empty `set`. It uses the same curly braces as `dict` for its literal syntax other than the empty `set`. A `set` is a collection type which includes zero, one, or more *unordered* items. All items in a `set` need to be unique.

Using the same constructor functions,

```
>>> list([4, 5, 5])
[4, 5, 5]
>>> set([4, 5, 5])
{4, 5}
>>> dict([('a', 4), ('a', 5), ('b', 5)])
{'a': 5, 'b': 5}
```

A new `list` can be created from other sequence objects, like another list. Likewise, a new `set` can be created from other collection type objects. One thing to note is that, in this example, we use a three-item `list`, `[4, 5, 5]`, as an argument to the `set()` function, but we end up getting a `set` of two items. (Note the curly brace literal syntax.) This is because we cannot have duplicates (e.g., two `5`s) in a `set`.

The same holds true with the `dict` type. One cannot have more than one items *with the same key* in a `dict` object. In this example, the argument to the `dict` constructor function is a list of three items, or three key-value pairs. And, because two items have the same key (e.g., `'a'`), the function returns a `dict` of two items. The value of the first item (`'a', 4`) has been replaced by that of the same key.

`dict` is a mutable complex type. Items can be added to a dictionary using the "index notation". Likewise, the values of the items in a dictionary can be modified. The items in a dictionary can also be accessed/read via a similar syntax. We use the same `del` builtin function to delete an item for a given key.

Here are some examples:

```

>>> d = {"k1": 1}           ①
>>> d
{'k1': 1}
>>> d["k1"]                 ②
1
>>> d.get("k1")             ③
1
>>> d.get("k0", 0)          ④
0
>>> d["k2"] = 2             ⑤
>>> d
{'k1': 1, 'k2': 2}
>>> d["k1"] = 100           ⑥
>>> d
{'k1': 100, 'k2': 2}
>>> del(d["k1"])            ⑦
>>> d
{'k2': 2}
>>> len(d)                  ⑧
1

```

- ① The variable `d` is bound to a `dict` object `{"k1": 1}`.
- ② We can get the value of an item using the subscript operator. If no item for the given key is found in the `dict` object, it will raise a `KeyError` exception.
- ③ Alternatively, we can access an item using the `get` method of the `dict` type.
- ④ The `dict.get` method can take a second argument as a default value. If the given key is not found in the dictionary, then this default value will be returned (e.g., instead of raising an exception).
- ⑤ A new item `"k2": 2` (e.g., whose key is `"k2"` and whose value is `2`) is added to this `dict` object through an "assignment". At this point, the value of the expression `d["k2"]` is `2`.
- ⑥ The same syntax can be used to update the value of an existing item. That is, it overwrites the value of the item with the given key if it already exists. Otherwise,

14.21. Dictionary

it inserts the new item into the dictionary.

- ⑦ We can delete an item using the `del` function. If the specified item does not exist in the object, then it will raise a `KeyError`.
- ⑧ The builtin function `len` returns the number of items, key value pairs, in a dictionary.

A dictionary can be iterated with the `for - in` statement just like other iterable type objects. For instance,

```
data = {"New York": 8.4, "L.A.": 4.0, "Chicago": 2.7}
for city in data:
    print(f"The population of {city} is {data[city]} millions.")
```

This `for` loop iterates over the keys of the dictionary, `city` in this example.

```
The population of New York is 8.4 millions.
The population of L.A. is 4.0 millions.
The population of Chicago is 2.7 millions.
```

Alternatively, we can iterate over the items.

```
for (city, pop) in data.items():
    print(f"The population of {city} is {pop} millions.")
```

This will print out the same text. Note that, in this example, we use the tuple unpacking to assign the loop variable, an (unnamed) two-item tuple, to two separate variables, `city` and `pop`.

The items in a dictionary are generally not "ordered". This is true for all, or most, "map" types across different languages. A dictionary/map is typically used with the key-based lookup, and hence in many cases, orders are not important.

However, in Modern Python, the "insertion order" is preserved (although you cannot still sort them in a particular order based on their keys or values, etc.). That is, the items in a dictionary are ordered based on when the items are added. This can be useful, in some limited circumstances, for instance, when we loop over a (small) dictionary.

In the above `for` loop examples, as can be seen from the sample output, the iteration goes over the items in the same order as they are included in the `dict` literal.

14.22. Putting It All Together

Let's recap. We implemented the rock paper scissors program using the *modern python language*. In particular, we used classes and enums to represent the "real world" objects and concepts. We also tried one of the most powerful constructs in modern programming, namely pattern matching, using Python's new `match` statement.

This new implementation is packaged into a package module, `rps`. The package includes 5 separate (sub-)modules.

Let's examine their inter-module dependencies. The `rps.game` module imports all four modules, `rps.result`, `rps.rand`, `rps.hand`, and `rps.reader`. Both `rps.rand` and `rps.reader` modules import the `rps.hand` module. The `rps.hand` and `rps.result` modules have no dependencies within the package. (You can easily verify this by looking at the top of each module.)

So, the "top" is the `rps.game` module, and the `rps.hand` and `rps.result` modules are at the bottom, with the `rps.rand` and `rps.reader` modules in the middle.

```
[game]-----*
|           |           |           |
|           |           [rand]    [reader]
|           |           |           |
[result]    [hand]---*-----*
```

14.22. Putting It All Together

Let's start from the top, the `rps.game` module. This module includes one constant, `MAX_ROUNDS`, and one `class` definition, `Game`. `Game`'s initializer method takes one optional argument, `num_rounds`. Its default value is `MAX_ROUNDS`, which is set to 5.

```
17 def __init__(self, num_rounds = MAX_ROUNDS):
18     ...
```

Therefore, we can provide a different `num_rounds` value while creating an instance object of `rps.Game`. For example,

```
>>> from rps.game import Game
>>> g = Game(1000)
>>> g._num_rounds
1000
```

Although `_num_rounds` is a "private" or "internal" instance variable, nothing really prevents it from being used outside the class definition. We verify that the variable is indeed correctly set by "sneakily" examining its value in the REPL. 😊

The `Game` class includes the total of 6 methods and 1 static function. Only one of these is a "public" method, so to speak, and the rest are "hidden" by convention. Hence, one can easily infer, in this case, that the public API comprises this one method, `rps.Game.start`.

The `start` method is a sequence of three statements, each of which is a method call.

```
23 def start(self):
24     "..."
25     self._banner()
26     self._loop()
27     self._end_game()
```

As we have seen earlier, the core program logic is included in the `game._loop`

method, which iterates the rock paper scissors rounds for `num_rounds` times, lines 63-88.

In each round, we read the player's hand by calling the `reader.read_hand` function (line 64), we generate a random computer hand by calling the `rand.random_hand` function (line 68), and we determine the winner, print out the result, and update the "game stats" (lines 70-86).

```

63 for _ in range(self._num_rounds):
64     player_hand = reader.read_hand() ①
65     if not player_hand:
66         return
67
68     computer_hand = rand.random_hand() ②
69
70     print(f"Your hand: {player_hand},", ③
71           f"computer hand: {computer_hand} ->",
72           sep=" ",
73           end=" ")
74
75     cmp = compare_hands(player_hand, computer_hand) ④
76     wol = Game._win_or_lose(cmp) ⑤
77     match wol: ⑥
78         case WinOrLose.WIN:
79             self._wins += 1
80             print("You win!")
81         case WinOrLose.LOSE:
82             self._losses += 1
83             print("You lose!")
84         case _:
85             self._ties += 1
86             print("Tie.")
87
88     self._end_round() ⑦

```

① Read the player's hand.

14.22. Putting It All Together

- ② Generate a random hand for the computer.
- ③ Print out both hands.
- ④ Compare hands.
- ⑤ Decide who wins. These two steps could have been combined into one.
- ⑥ Print out the result and update the "stats".
- ⑦ Print the end-of-round message.

The `read_hand` function of the `rps.reader` module reads the user input, and it returns the user hand as `Hand`. Note that we treat Ctrl+D, Ctrl+C, `q`, and `x` as a request to end the game, and the function simply returns `None`, lines 9-10 or `rps/reader.py`.

```
5 def read_hand() -> Optional[Hand]:
6     while True:
7         try:
8             i = input("Rock (r), Paper (p), or Scissors (s)? ").lower()
9             if i.startswith("q") or i.startswith("x"):
10                 return
11
12             return _parse_hand(i[0])
13         except (EOFError, KeyboardInterrupt):
14             return
15         except:
16             print("Invalid input. Try again.")
17             continue
```

The `None` return value effectively terminates the entire program, by returning from the `_loop` method, lines 65-66 of `rps/game.py`.

The `random_hand` function of the `rps.rand` module uses the `random.choice` function to randomly select one of the three hands, line 6. Again, this function returns an object of the `rps.Hand` type.

```

5 def random_hand():
6     return random.choice((Hand.ROCK, Hand.PAPER, Hand.SCISSORS))

```

Next, we compare these hands with the `compare_hands` function defined in the `rps.hand` module. This function returns `1`, `-1`, or `0` depending on whether the first hand wins, loses, or otherwise, respectively.

The result is then displayed, and the game stats are updated, in lines 76-86 in the `rps.game` module. This routine is repeated for `self._num_rounds` time unless the user decides to terminate the program before then. At the end of the game, the overall stats is shown via the `_end_game` method.

That's all there is to it. ☺ If you run the main script, *main.py*, then it first creates an instance of the type `Game`, and it calls its method `start`, which takes care of the rest. The `rps` module, e.g., its 5 submodules as a single package, is "reusable", and it can be shared.

14.23. Code Review

We will leave the code review to the readers. You can start by going through [the code in the beginning of this part](#) again.

As stated, there is more than one way *to skin a python*. There can be many different ways in which this program could have been written. It is a good training to read other people's code with some healthy skepticism. (And, "healthy" is the keyword here.) It goes without saying that this advice applies to all the sample code included in this book as well.

One obvious thing to note in the rock paper scissors game implementation in this part is that there is some "weird code". ☺

The `compare_hands` function defined in the `rps.hand` module returns an integer, `1`, `-1`, or `0`, and we convert this returned value into an `enum` value in the `_win_or_lose` method of the `Game` type. Why?

14.23. Code Review

We could have done this with one function call. Instead of returning an `int`, we could have just returned a member of `WinOrLose`. Why do it in two steps?

We did it this way because we just wanted to see if the readers were paying attention. 😏 Jk, but, besides that, there is a good reason.

As written, the `rps.hand` module has no dependencies on other modules. (The `enum` module is a part of the standard library.) This can be a good thing, e.g., in terms of sharing your modules. The `hand` module can be used separately outside the `rps` package, for instance.

If we had written this function, `compare_hands`, differently, e.g., to return the members of the `WinOrLose` enum type, etc., then we would have ended up with (slightly) more complex dependencies, which may or may not be a good thing.

In this small program, there is really no difference. If anything, the simpler option, e.g., using one function call than two, is generally preferred. But, as we start building larger and larger systems, this kind of consideration becomes a more and more important part in designing software systems. In software architecture, we often prefer "loose coupling" between the software components (e.g., classes, modules, services, etc.).

By the way, returning `-1`, `0`, or `1` in a "comparison function" is more or less a convention, in many different contexts.

One more lab, and we are done. 😊

Chapter 15. Lab 3 - OOP and Other Modern Features

But in the end it's only a passing thing, this shadow; even darkness must pass.

— Sam (The Lord of the Rings)

15.1. Days of the Week

Create an enum that has seven members, Monday through Sunday (or, Sunday through Saturday, depending on what day is considered the first day of the week in your locale ☺).

You can use integer values, e.g., `1` through `7`, or use string values.

Note that when we use integers, we do not tend to use `0` as a valid member value, for example, because its truth value is `False` although the truth values of all enum members (or instance objects) are `True` by default. This can potentially cause a confusion.

When you include a special member like "Nothing", "Absent", "Unknown", etc., that can be semantically viewed as false, then you can use `0` (or, an empty string, etc.), and you can override the `__bool__` method.

Write a script that prints out all names of the 7 members in the enum.

15.2. I'll Be Going ...

Create an enum for the four cardinal directions on the compass. For example, the enum type may include the four members, `NORTH`, `EAST`, `SOUTH`, and `WEST`, or `N`, `E`, `S`, and `W`, etc.

15.3. Playing Cards

Override the `__str__` method so that it prints out an appropriate text, e.g., "north", "east", "south", and "west", for each of the four members, regardless of its enum value.

Write a function that picks a random direction, e.g., using a function in the `random` module, and prints out a text like this, depending on the random direction,

```
I have nothing to do today.  
I'll be going to the east, as the wind blows. :)
```

Write a simple script that calls this function 666 times. 😊

15.3. Playing Cards

The standard deck of cards (aka "French deck") comprises 52 cards. Each card has a suit and a rank.

Create an `enum` for the 4 suits, "Spade", "Diamond", "Heart", and "Clubs".

Then, create a `class` representing a card, whose data attributes are `suit` of the Suit enum type, and `rank` of the `int` type. The ranks of the cards are Ace, 2, 3, ..., 9, 10, Jack, Queen, and King. You can map this set to any `int` values as you see fit.

Write a program that creates all 52 cards of the standard deck. Then, print out all cards. You can override the `__str__` methods, or any other dunder methods, of the suit enum and/or the card class, as needed.

15.4. Length Function

Python has the builtin `len` function, which we have been using throughout this book. The `len` function returns the length of a given sequence (that is, the number of elements in the sequence).

Write a length function, without using the builtin `len`, that takes a `list` of `int` as an

argument and returns the length of the given list.

- First, implement this function using a `for - in` loop, that is, "iteratively".
- Then, implement this function "recursively", using a `match - case` statement.

Write a script that tests these two implementations, e.g., for a number of sample `int` lists.

15.5. Sum Function

Python has another builtin `sum` function, which, given a sequence of numbers, returns the sum of all their values.

```
sum(iterable, /, start=0)
    Return the sum of a 'start' value (default: 0) plus an iterable of
    numbers

    When the iterable is empty, return the start value.
    This function is intended specifically for use with numeric values and
    may
    reject non-numeric types.
```

Now, without using this builtin function ☺, write a function that takes a `list` of `int` and returns the sum of all its items.

- First, implement this function iteratively using a `for - in` loop.
- Next, implement this function recursively using a `match - case` statement. Or, you can use an `if - elif - else` statement if that seems more natural to you.

Write a script that tests these two implementations, e.g., for a number of sample `int` lists. You can reuse the same lists from the previous exercise.

15.6. Product Function

Python does *not* have a builtin "product" function ☹, that is, a function that is like `sum()` but returns the product of the list elements rather than their sum.

Write a function that takes a `list` of `ints` and returns the product of all its items. That is, given a list `[1, 2, 3, 4]`, the function will return `24`, from `1 * 2 * 3 * 4`.

- First, implement this function iteratively using a `for - in` loop.
- Then, implement this function recursively, e.g., using a `match - case` statement.

Write a script that tests these two implementations, e.g., for a number of sample `int` lists. You can reuse the same lists from the previous two exercises.

15.7. String Length Comparison

Write a comparison function that takes two string arguments and returns `-1` if the first argument is shorter than the second, `1` if the opposite is true, and `0` if the lengths of both strings are the same.

Write a script to test this function with various strings, such as the names of some Python species. ☺

15.8. String Concat Function

Strings in Python can be concatenated using the plus, or concatenation operator, `+`.

Write a function that takes a `list` of strings, and returns a string which is a concatenation of all items in the list, separated by commas.

- Implement this function iteratively, e.g., using a `for - in` loop.
- Implement this function again, now recursively, e.g., using a `match - case`

statement.

Write a script that tests these two implementations, e.g., for a number of sample `str` lists.

In fact, Python has a builtin string method `join`, which "joins" all items in a given list. For example,

```
>>> food = ["mice", "lizards", "birds", "pigs", "monkeys"]
>>> print("Pythons eat", ", ".join(food))
Pythons eat mice, lizards, birds, pigs, monkeys
```

15.9. Multiplication Table

Write a C program, oops, a Python script ☺, which prints out the multiplication table between integers `2` and `12`. That is, the output might look like this:

	2	3	4	5	6	7	8	9	10	11	12
2	4	6	8	10	12	14	16	18	20	22	24
3	6	9	12	15	18	21	24	27	30	33	36
4	8	12	16	20	24	28	32	36	40	44	48
5	10	15	20	25	30	35	40	45	50	55	60
6	12	18	24	30	36	42	48	54	60	66	72
7	14	21	28	35	42	49	56	63	70	77	84
8	16	24	32	40	48	56	64	72	80	88	96
9	18	27	36	45	54	63	72	81	90	99	108
10	20	30	40	50	60	70	80	90	100	110	120
11	22	33	44	55	66	77	88	99	110	121	132
12	24	36	48	60	72	84	96	108	120	132	144

You can use nested `for` loops, one for going through the first argument and the other for going through the second argument in multiplications.

15.10. Fibonacci Sequence

Note that, to align the output values, if you want to, you will need to use the `sep` and `end` arguments in the `print()` function calls, as well as the formatting strings for the `int` values. For example, to right-align `int` values, while printing the numbers from 8 through 11,

```
>>> for i in range(8, 11 + 1):  
...     print(f"i = {i:>2}")  
...  
i =  8  
i =  9  
i = 10  
i = 11
```

The format string `>2` in `{i:>2}` indicates that the number `i` should be printed with at least 2 spaces.

15.10. Fibonacci Sequence

The values of the Fibonacci sequence are defined by the previous two values in a sequence. That is, given 2 and 3 in the sequence, the next number is $2 + 3 = 5$. The sequence starts with 0 and 0, 0 and 1, or 1 and 1, etc. They all lead to sequences that are pretty much equivalent in the long run.

Refer, for instance, to a Wikipedia article, [Fibonacci numbers](https://en.wikipedia.org/wiki/Fibonacci_number) [https://en.wikipedia.org/wiki/Fibonacci_number], for more information.

Write a function that generates the Fibonacci sequence for the first `n` numbers, e.g., using the `while` loop statement.

You can start from any of the valid pairs of small numbers, like 0, 0, or 0, 1, and so forth.

You can do this iteratively, e.g., by storing the previous two number in the iteration, or you can do this recursively, e.g., by calling the same function to get the previous

numbers. Or, you can do two different implementations both ways. 😊

15.11. Rock Paper Scissors

Close the book. We are done. 😊

No, one last problem. Forget everything that you learned in this book, if any. 😊 Now, the problem is,

- Write a rock paper scissors program that lets you play the game with the computer.

Can you do it? Will you do it?

Well, all the exercises in this book are optional, as we made it abundantly clear. 😊

Good luck!

Wrapping Up

This day does not belong to one man but to all. Let us together rebuild this world that we may share in the days of peace.

— Aragon (The Lord of the Rings)

A time for the "final exam"! How exciting! 😊

Obviously, there is no final exam. 😊 This is an optional part, which includes some simple Python "programming projects". You can go back and review the previous lessons, or you can try to work on some new programs in this part.



This is a long book. Not only that, we have covered a lot of topics in this book, including many difficult topics. If you have understood, and "internalized", even a fraction of what this book teaches, you are now a much better programmer than the vast majority of the Python programmers out there. (No kidding. 😊)

If you'd like, you can stop here. Take a break, and next time, when you feel like it, read this book again from the beginning, or from [the beginning of the first project](#), and at that time you can finish these extra lessons.

Chapter 16. Final Projects



16.1. Computer vs Computer

In the last three rock paper scissors game lessons, we ended up creating the computer players. The random hand generator is essentially a computer player.

Now, create a program in which two computer players play against each other.

- Take the number of rounds as a command line argument. (You will need to figure out how to read the command line arguments in Python, and convert it into `int`, etc.)
- Print out the two hands and the outcome, in one line, for each round.
- Record the win, loss, and tie for each computer player, in each round.

You can iterate this for a million times. Then, at the end of the program run, print out the total wins, losses, and ties for both players. In theory, they have to be $1/3$, $1/3$, and $1/3$. Are they close enough?



Even if your goal is to iterate for one million times, you never do that from the beginning. While you are developing a program, and debugging/troubleshooting, you always start from a small number of iterations, like one, and two, and ten, and so forth...

16.2. How to Prevent Cheating

As a programmer, you can do just about anything. You can "cheat".

Now the question is, how to make your users trust your program, and hence indirectly trust you. This is an open ended, and rather general, question.

In the games like the one that we worked on in this book, where the user plays against the computer, "transparency" is rather important.

One way to do this is to open-source, or publish, your program source code so that other people can examine it. In many cases, however, that is not a very practical solution.

Another way is to provide the users with some way to "audit" a particular run of the game, that is, after the game has been played. This is, in fact, commonly done, for example, in the online poker games. Cards are shuffled, and they are recorded (e.g., saved into a file) and they are kept in a safe/secrete place (so that they cannot be tempered with) *before* a game is played.

After the game is finished, we can verify that the game has been played out exactly the way the cards were pre-shuffled.

Let's try a similar technique for the rock paper scissors game.

When a new game starts,

- First, generate a series of random hands, say, 100 of them.
- Write this sequence into a file, e.g., with a particular name.

- Make the file un-writable in some way (at least, from the program).
- Play the game using this series of hands as the computer hands, that is, instead of generating them on the fly.
- After the game ends, print out all the hands, both the computer's and the player's.
- The user can verify the computer hand series by reading the hand sequence file that was saved.

Modify your rock paper scissors program including these changes. Although we did not discuss how to open, read to, and write from a file, it is part of the project to figure out how to do this on your own. 😊

16.3. Student Records

Create a class `Student`, which has the following three data attributes (instance variables):

- `name`: A string.
- `attendance`: A list of `bools`.
- `score`: A `float` between `50.0` and `100.0` (both inclusive).

This class also includes an instance method, `grade`, which returns an enum object, `A`, `B`, `C`, `D`, and `F`. You can leave its implementation empty for now.

Override the `__init__` method so that it takes one argument, `name`, of the string type. Assign this value to the instance variable `name`. Initialize `attendance` and `score` with an empty list `[]` and `0.0`, respectively. Override any other dunder methods as necessary, or as desired.

Create 200 instance objects of this class, and give each a random, but unique, name. Any strings will do as long as there are no duplicates. (How can you ensure that?)

Create a dictionary of all these 200 `Students`, with their `name` as a key and the

16.3. Student Records

instance object itself as a value.

Generate 200 random `floats` between 50.0 and 100.0, and assign each of these values to each of the 200 `Student` instances. You can use the dictionary object that you just created for iteration. (You will need to study the `random` module functions to find the appropriate function. And, you'll need to be a little bit creative. See the note below.)

Generate 800 ($= 200 * 40$) random `bools` with the 80% probability of having `True` and 20% of having `False`. Add 40 of these values to each of the `attendance` lists for the 200 `Students`.



This can be a bit tricky, but as we stated in the very beginning, programming is really about problem solving. Solving this problem does not require any more Python knowledge than we have already covered in this book.

Now implement an `enum` class, with five members: `A`, `B`, `C`, `D`, and `F`.

Implement the `grade` instance method for the `Student` class. Here's the rule:

- If the student's attendance is less than 70% out of the 40 days, then he/she automatically fails regardless of their scores. That is, its grade is `F`.
- Otherwise, that is, as long as their attendance is 70% or higher, the attendance does not affect their grade.
 1. If the student's score is 90 or higher, his/her grade is `A`.
 2. Otherwise, and if the student's score is 80 or higher, their grade is `B`.
 3. Otherwise, and if the student's score is 70 or higher, their grade is `C`.
 4. Otherwise, and if the student's score is 60 or higher, their grade is `D`.
 5. Otherwise, the student's grade is `F`. That is, he/she fails the class.



Although we define `grade` as a method, we do not want to compute

this value every time the `grade()` method is called. Once a student's attendance and their score is known, the grade is determined (according to the above rules). How would you implement the `grade` method so that we do not have to do the same computation over and over again?

Hint: Why do we use instance variables in a `class`?

Write a function which takes a student's name and returns their grade.

Print out the names of the students who failed the course, that is, all students with the grade `F`.

Print out each student's name and grade in each line for the students who did not fail the course. That is, for all students whose grade is `A`, `B`, `C`, or `D`. Print the list in a sorted way in terms of their names.



We did not cover how to sort a dictionary in this book. In fact, dictionary objects are not generally sortable. This can be possibly a rather difficult problem. If you cannot figure this out, then you can just print their names as they are stored in the dictionary.

Hint: What kinds of data types are sortable?

Finally, compute the average score of all passed students.

16.4. War (Card Game)

"War", or "Battle", is one of the simplest card games for children, typically played by two players.

One card is dealt to each player, and whoever has the card with a higher rank wins the round, and he/she gets the cards.

If both cards are the same, then they "go to war", and keep playing, while increasing

16.4. War (Card Game)

the stake, until one side wins. Here's a link to the Wikipedia article: [War \(Card Game\)](https://en.wikipedia.org/wiki/War_(card_game))
[https://en.wikipedia.org/wiki/War_(card_game)]

At the end of the game, whoever collects the most cards wins.

The war card game has many similarities with the rock paper scissors game. Implement the war game that lets the user play against the computer.



Try to follow the same initial steps that we did when we started working on the rock papers scissors game. For example, you will need to break down this problem into smaller tasks, among other things. One thing to note is that you do not have to *exactly* replicate the real world war game. You can make certain simplifications to make the implementation easier. For instance, you may find dealing with a single deck (e.g., 52 cards) a bit difficult. In that case, you can use an "infinite" number of cards. (How would that simplify our implementation?) There are many other ways in which you can simplify the game more suitable for the computer play.

Chapter 17. Epilog - Let's Play!

You must trust yourself. Trust your own strength.

— Gandalf (The Lord of the Rings)

We have covered a lot of important concepts of programming in Python in this book. We have covered a lot of ground. As the Gandalf the Wise says 😊, however, *"The world is not in your books and map. It is out there."* Ultimately, you will need to "go out" and do, and practice.

What you have learned in this book will become your compass, a guiding light, on your journey. Your lifelong journey of learning, programming, and *happiness*. 😊 It will make your journey more enjoyable. You will not be lost as much regardless of where you are heading.

Congratulations!!!

Let's celebrate! Let's play some games! That is, the rock paper scissors game. 😊

Use your own program, to play. We call this practice the "dogfooding". You may find some (more) bugs while using your own program, or you may find some pain points in your program that you haven't realized before.

Regardless, it is a "reward". It is a *real joy* to use the program that *you* have created. This is the joy that *only the programmers can experience*. 😊

Here you go.

```
$ python main.py
```

```
-----  
Welcome to Rock Paper Scissors!  
Type X or Q to end the game.  
-----
```

```
Rock (r), Paper (p), or Scissors (s)? r
Your hand: Rock, computer hand: Scissors -> You win!
Your wins: 1, losses: 0 out of 1 rounds
-----
Rock (r), Paper (p), or Scissors (s)? r
Your hand: Rock, computer hand: Paper -> You lose!
Your wins: 1, losses: 1 out of 2 rounds
-----
Rock (r), Paper (p), or Scissors (s)? r
Your hand: Rock, computer hand: Scissors -> You win!
Your wins: 2, losses: 1 out of 3 rounds
-----
Rock (r), Paper (p), or Scissors (s)? r
Your hand: Rock, computer hand: Paper -> You lose!
Your wins: 2, losses: 2 out of 4 rounds
-----
Rock (r), Paper (p), or Scissors (s)? r
Your hand: Rock, computer hand: Paper -> You lose!
Your wins: 2, losses: 3 out of 5 rounds
-----
Thanks for playing Rock, Paper, Scissors!!
Your final score:
Wins: 2, Losses: 3, Total rounds: 5.
-----
```

Urghh. I lost. *But, that's all right.* ☹

Index

@

\$, 27, 114

\$ symbol, 112

&&, 27

'float' type, 66

+ operation, 64

+ operator, 65

-c or -i flags, 186

-m flag, 112, 187

-m flag, 187

.gitignore, 117

.gitignore file for Python, 117

.gitignore files, 117

3.10 release, 260

3.10 syntax, 266

4 spaces, 138

4-space indentation rule, 162

= symbol, 77

@classmethod, 289

@staticmethod, 289

[], 74

_, 52, 54, 78

_name, 81-82

_variable, 72, 81

__bool__, 307

__bool__ method, 306, 328

__class__ attribute, 301, 305

__doc__ attribute, 292

__eq__, 293

__ge__, 293

__gt__, 293

__init__, 294-295, 298

__init__ implementation, 298

__init__ method, 282, 287, 291-292,
299, 317, 338

__init__.py, 194

__init__.py file, 194

__le__, 293

__lt__, 293

__members__, 309

__ne__, 293

__str__, 294-296, 298

__str__ method, 292, 300, 329

__str__ methods, 310, 315, 329

A

absolute basics, 39

absolute file path, 124

absolute import, 195

actions, 253

active **Exception**, 231

active virtual environment, 114

addition, 40

addition + operator, 64

additions, 41, 45

algebraic types, 266

algorithms, 272

alias, 80, 83, 143

aliases, 262

All caps, 234

all caps names, 198

all-caps naming convention, 308

- alternative cases, 311
- Anaconda, 111
- and**, 137, 157-158
- and** and **or**, 159
- and** and **or** operators, 157
- and** expression, 137, 160
- and** operator, 158
- and** or **or** binary operations, 159
- annotations, 33
- append**, 78, 91
- append** method, 78, 81
- append()** call, 81
- argument, 17, 32, 52-53, 85, 127, 186
- argument objects, 85
- argument of type **str**, 131
- argument passing, 167
- argument values, 67
- arguments, 45, 199, 201
- arithmetic and comparison operators, 212
- arithmetic expression, 161
- arithmetic operation, 41-42
- Arithmetic Operations, 40
- arithmetic operations, 45, 137
- arithmetic operator, 218
- array, 74
- array type, 74
- array types, 74
- ascending order, 90
- Assignment, 83
- assignment, 77-78, 82-85, 144-145, 167, 235, 320
- assignment expression, 82
- assignment operator, 77
- assignment statement, 77-78, 82-83, 85, 89, 135, 145, 196
- assignments, 85, 147
- associated *attributes*, 91
- at compile time, 303
- at run time, 32, 57, 145, 302-303
- at run time*, 214
- attribute, 83
- attribute lists, 297
- attribute of an object, 78
- Attribute references, 285
- attribute references, 279-280
- Attributes, 133
- attributes, 78, 133-135, 190
- augmented arithmetic operator, 218
- augmented arithmetic statements, 219
- auto styling and formatting, 114
- autopep8* extension, 126
- autopep8** module, 114
- B**
- backslash, 30
- backslash + newline, 264, 266
- backslash + newline character, 265
- backslash + other white space, 264
- backslash ****, 264
- Backslashes, 264
- base 10, 57
- base case, 273-274, 316
- base class, 293-294, 298-300, 309
- base class **object**, 295
- base classes, 277
- base type, 299
- base type **object**, 76, 291-292
- BaseException**, 228, 232
- BASH, 25, 27, 109, 112

- BASH shell, 47
- basic builtin types, 69
- Basic Concepts of Programming, 38
- basic mathematics, 45
- basic OOP programming techniques, 243
- basics of programming, 18, 96
- basics of Python, 68
- beginning programmers, 18, 23, 48
- better programmer, 34
- binary, 57
- binary arithmetic operations, 40
- binary Boolean operators, 157
- binary distribution, 21
- binary operation, 137
- bind a new name, 83
- binding, 144
- binds, 82
- BitBucket, 116
- block scoped language, 151
- block-scoped* language, 150
- block-scoped languages, 125
- blocks, 125
- `bool`, 40, 48, 55-56, 58, 269
- `bool` expression, 270
- `bool` function, 59
- `bool` literal, 55
- `bool` literals, 57
- `bool` type, 40
- `bool` type, 45, 135
- `bool` value, 56, 66
- `bool` values, 42, 135, 158
- `bool()` constructor function, 270
- Boolean, 40
- Boolean `and`, 137
- Boolean AND operator, 137
- Boolean Context, 269
- Boolean context, 141, 261, 270-271, 275
- boolean context, 269-270
- boolean context*, 306
- Boolean expression, 56, 58, 62, 139, 159-161, 165, 185, 190, 216, 223, 225-226, 269, 275
- boolean expression, 159
- Boolean Expressions, 55
- Boolean expressions, 135, 139, 269, 275
- boolean expressions, 313
- Boolean Literals, 40
- Boolean Operators, 157
- Boolean operators, 269, 275
- boolean operators, 158
- Boolean operators in Python, 158
- Boolean value, 59, 216, 269-270, 306-307
- Boolean value of `None`, 60
- Boolean values, 60, 269
- booleans, 45
- bound, 73, 78, 81, 83, 85, 143, 165
- bound object, 146
- bound to, 197, 218, 320
- bound to an object, 263
- bound to the object, 77
- Bourne Shell, 109
- Bourne shell, 112
- brackets, 161, 264-265
- brackets*, 162
- branch, 118
- branches, 118
- `break`, 225, 229
- `break` simple statement, 225
- `break` statement, 225

- `break` statements, 240, 311
- bugs, 32
- build time, 267
- build tools, 267
- built-in function, 48
- built-in simple type, 145
- built-in type, 54
- built-in types, 48
- builtin, 59
- builtin `append` method, 78
- Builtin `bool` Function, 58
- builtin `bool` function, 66, 307
- builtin complex types, 69
- builtin compound data type, 74
- builtin `del` function, 147
- builtin `dir` function, 92
- builtin function, 17, 52, 58, 281, 330
- builtin function `del`, 146
- builtin function `input`, 142
- builtin function `len`, 321
- builtin function `pow`, 246
- builtin function `reversed`, 248
- builtin function `sum`, 222
- builtin functions, 56, 65, 167, 221
- builtin `id` function, 43
- builtin immutable types, 260
- Builtin `input` Function, 142
- builtin `input` function, 126, 232
- builtin `isinstance` function, 301
- builtin `issubclass` function, 301
- builtin `len`, 329
- builtin `len` function, 271, 329
- builtin method, 91
- builtin method `reverse`, 247
- builtin methods, 79, 153
- builtin methods*, 154
- builtin number type, 39
- builtin numeric types, 45
- builtin operators, 212
- builtin or custom types, 277
- Builtin `print` Function, 52
- builtin `print` function, 17
- builtin `range` function, 221
- builtin `sorted` function, 90
- builtin `str` constructor function, 214
- builtin `str` type, 153
- builtin `str.lower` method, 242
- builtin `str.strip` method, 241
- builtin string `lower` method, 143
- builtin string method `join`, 332
- builtin `sum` function, 330
- builtin terminal in VSCode, 28
- builtin type, 90, 319
- builtin type constructor functions, 318
- Builtin `type` Function, 48
- builtin type in Python, 63
- builtin type `int`, 293
- builtin type `object`, 277
- builtin types, 43, 45, 55, 57, 59, 280, 317
- builtin types*, 154
- builtin types and functions*, 130
- builtin types and objects, 133

C

- C shell, 112
- C-descendent languages, 264
- C-style *blocks*, 152

- C-style language, 261
- C-style language background, 151
- C-style languages, 58, 66, 79-80, 125, 139, 144, 145, 157-158, 161, 167, 216, 223
- C-style* languages, 144, 207
- C-style programming language, 80
- C-style programming languages, 79, 144, 151, 160, 167, 263
- C-style switch statement, 311
- C-style) programming languages, 62, 71
- call, 67, 133
- call a function, 85, 167
- call chain, 226, 230
- call return chain, 227
- call `super()`, 298
- call the function, 32
- callable, 135, 280
- callable*, 286
- callable class object, 298
- callable's*, 280
- called function, 85, 167
- callee function, 167
- caller, 167
- caller of the function, 139
- calling, 212
- calling a function, 32
- calling context, 85
- camel case, 149
- card games, 340
- cascading indentations, 138
- `case _`, 314
- `case _` clause, 311
- `case` clause, 311
- case clause, 314
- `case` clauses, 311
- case clauses, 313
- case sensitive, 149
- case-sensitive, 55
- casting, 59
- catch-all case, 314
- catch-all case `case _`, 311
- catch-all default case, 311
- catch-all `except` clause, 241
- character, 64
- characters, 63-64, 220
- choice* function, 211
- Class, 277
- `class`, 260, 276-277, 280, 307, 309
- class, 282, 300
- `class` definition, 276, 278, 288, 323
- class definition, 277-279, 281, 288, 323
- class definitions, 277
- class function call syntax, 289
- class functions, 289
- class hierarchy, 299
- class inheritance, 305
- class instances, 189
- Class instantiation, 282
- `class` keyword, 294
- class methods, 277, 289
- class name, 260
- class object, 278-281, 284, 287-289, 297
- class object*, 278
- `class` object, 280
- Class Objects, 279
- class objects, 133, 290
- class or instance objects, 290
- class scope, 279

- `class` statement, 253, 278-279, 284
- class syntax, 307
- class variable, 285
- Class Variables, 284
- class variables, 277, 284, 289, 308
- class-based polymorphism, 303
- classes, 49, 189, 253, 276, 307
- classes and enums, 322
- Classes in Python, 253
- classic C-style `for`, 222
- classic `for`, 219
- clause, 62
- CLI basics, 25
- CLI tools, 20
- CLI tools for development, 20
- CLI version of *pip*, 113
- client-server model, 115
- CMD, 25, 27, 113
- code box, 28
- code duplications, 169
- code editor, 126
- code file, 184
- code intelligence, 129
- code readability, 170
- code review, 169-170
- code sample, 27
- collection type, 74, 245
- collection type objects, 85, 319
- colon, 31, 132
- colon `:`, 132, 164
- colons, 139
- combo module, 190
- combo modules, 190-191
- command, 27
- command `code`, 26
- command line, 26, 99, 186
- command line argument, 27, 29, 37, 124, 336
- command line argument syntax, 187
- command line arguments, 336
- command line flag, 30
- Command Line Interface, 20
- command name, 35
- Command Prompt, 20
- commands, 29
- commas, 196
- comment, 126
- comment syntax, 131
- Commenting, 127
- comments, 132
- community-driven project, 126
- comparison, 137
- comparison function, 327, 331
- comparison operations, 135
- Comparison Operators, 135
- comparison operators, 135-136
- comparisons in Python, 137
- compile error, 295
- compile time error, 58
- compiler, 58
- compilers, 32
- complex, 60
- complex (mutable) type, 95
- complex literals, 69
- complex numbers, 39
- Complex Types, 68
- complex types, 69
- compound, 62
- compound data types, 94

- compound **for** statement, 234
- compound Python statements, 209
- compound statement, 31, 61-62, 96, 126, 137, 138, 184-185, 223
- compound statements, 161, 184
- compound type object, 72
- compound types, 69
- comprehension, 95
- computer, 38
- computer memory, 39
- computer play, 341
- computer player, 186, 336
- computer players, 336
- computer program, 18
- computer programming, 51, 100, 273
- computer programming languages, 49
- Computer programs, 184
- computer's hand, 186
- concatenated string, 64
- concatenation, 31, 331
- concatenation operator, 331
- conditional clauses, 62
- conditional compound statement, 164
- Conditional Expressions, 216
- conditional expressions, 216, 275
- conditional **if** statement, 127
- conditional logic, 239
- Conditional Statement, 62
- conditional statement, 62, 132, 155, 191, 216, 314
- conditional* statements, 137
- conditional statements, 275
- console programs, 20
- constant value objects, 307
- constant variables, 185
- constants, 198
- constructor, 280, 298
- constructor*, 284
- constructor function, 90
- constructor functions, 59, 215
- constructor syntax, 283
- Constructors, 280
- constructors, 280
- contexts, 271
- continuation prompt, 64
- continue**, 241
- continue** statement, 226, 241
- continue** statements, 229
- control, 233
- convention, 222
- conventional OOP, 286
- conversion, 59
- corner cases, 163
- CPython, 130
- create a new file, 27
- create and delete a folder, 27
- curly brace, 319
- curly braces, 95, 151, 318-319
- current directory, 109
- current shell, 112
- current stable release, 21
- current user, 110
- current working directory, 26, 110, 112
- currently active virtual environment, 114
- currently-bound object, 146
- custom **enum** type, 268
- custom exception type, 228
- custom functions, 167

- custom objects, 286
- custom type, 247, 275-277, 307
- custom type definitions, 262
- custom type `Game`, 259
- custom types, 253

D

- data and actions, 253
- data attribute, 85, 299
- data attributes, 91, 133, 285-286, 289, 292-293, 338
- data hiding, 290
- data science, 111, 263
- data structure, 95, 317
- debugging, 128, 155
- decimal numbers, 57
- decimal part, 41
- decimal point, 39
- deconstructing, 196
- decorators, 289
- decrement, 221
- `def` compound statement, 312
- `def` function statement, 138
- `def` line, 132
- `def` statement, 135, 138, 148, 279
- `def` statements, 208, 288
- default argument values, 201
- default* case, 311
- default case, 311
- default `case _`, 314
- default value, 202
- default values, 201
- definition of *type*, 55
- definition statements, 189, 233

- definitions of functions, 126
- `del`, 281
- `del` builtin function, 319
- `del` function, 146, 321
- `del()` call, 281
- derived class, 294, 298-299
- descending order, 90
- descriptive name, 150
- descriptive names, 150, 190
- design decision, 232, 268
- designing software systems, 327
- desktop calculator, 38
- destructuring, 196
- dev environment setup, 34
- develop a software, 33
- develop Python programs, 19
- Developing in WSL, 24
- development, 36
- development computer, 115
- development environment, 19, 173
- development process, 121
- development using the builtin terminals, 28
- development workspace, 120
- `dict`, 95, 317, 319
- `dict` constructor function, 319
- `dict` literal, 317
- `dict` literal, 322
- `dict` object, 319-320
- `dict` type, 95, 317, 319-320
- `dict()` constructor function, 318
- `dict.get` method, 320
- dictionaries, 95
- Dictionary, 316
- dictionary, 95, 317, 319, 321, 338, 340

- dictionary comprehension, 95
- dictionary expression, 318
- dictionary literal, 318
- dictionary object, 339
- dictionary objects, 340
- dictionary type, 95, 316
- different `id`, 84
- different identity, 44
- different implementations, 103
- different list object, 84
- different name, 84
- different object, 84
- different objects, 84
- different ordering, 90
- different variables, 84
- digital calculator, 38
- `dir` function, 129, 133
- `dir` function call, 92
- `dir(str)` function, 153
- direct import name syntax, 166
- directory names, 27
- divide and conquer, 104, 107
- divide the problem, 105
- division, 41
- division by zero, 51
- division in Python, 71
- division operator, 41
- divisions, 45
- doc comment, 207
- doc string, 207-208
- doc string literals, 208
- Doc Strings, 207
- doc strings, 207
- docstring, 234
- docstring of the type, 292
- documentation, 211
- documentation purposes, 132
- documentations, 127, 130, 208, 216
- dollar prefix, 27
- domain knowledge, 18, 100
- dot files, 117
- dot notation, 78, 279
- dot prefix, 156
- dot syntax, 130
- dotted module name, 193
- dotted module names, 195
- dotted names, 287
- double, 43
- double precision, 43
- double quote pairs, 64
- double quote string literals, 65
- double quoted string, 215
- double quoted string literals, 64
- double quotes, 17, 32, 63-64, 208, 214-215
- double underscores, 92
- download page, 24
- Download Visual Studio Code, 24
- duck OOP, 306
- duck typing, 305, 307
- Dunder Attributes, 291
- dunder attributes, 92
- dunder method, 291
- dunder methods, 92, 293, 329, 338
- dunder names, 291
- duplicates, 319, 338
- dynamic size array, 74
- dynamic type system, 267
- Dynamic Typing, 57

- dynamically typed, 57
- dynamically typed language, 280
- dynamically typed languages, 32, 266, 302-303, 305
- dynamically typed programming languages, 32
- dynamically types languages, 58

E

- edge cases, 163
- editors, 23
- `elif`, 137-138, 160
- `elif` clause, 139, 159
- `elif` keyword, 161, 269
- `elif` lines, 139
- Ellipsis, 209-210
- `ellipsis` literal, 210
- `ellipsis` type, 210
- Ellipsis value, 209
- `else`, 137-138, 140, 160, 224
- `else` clause, 62, 139-140, 155, 224-225
- emacs, 23
- empty dictionary, 318
- empty dictionary literal, 318
- empty line, 208
- empty line gaps, 127
- empty line input, 96
- empty lines, 31, 125-126, 184, 235
- empty list, 270, 316, 338
- empty sequence, 94
- empty `set`, 319
- empty set of elements, 88
- empty string, 140, 155, 165, 292
- encapsulation, 276
- end index, 89
- end slice index, 89
- English alphabets, 63
- Enter, 28
- entire sequence, 87
- entry point, 233
- Enum, 307
- `enum`, 277, 307, 310
- `enum`, 307
- `enum` class, 308
- `enum` class, 339
- `enum` definition, 307
- `enum` member, 308
- `enum` members, 307, 310, 328
- `enum` module, 327
- `enum` object, 338
- `enum` type, 276, 307, 309, 328
- `enum` type, 307-309, 314
- `Enum` types, 198
- `enum` types, 307
- `enum` types, 309
- `enum` value, 326
- `enum.Enum` type, 308
- `enumerate`, 222
- `enumerate` function, 222
- enums, 261
- EOF signal, 37, 66
- EOFError, 226
- `EOFError`, 232, 241, 268
- `EOFError` exception, 232
- equality and comparison-related behaviors, 293
- equality comparison operator, 164
- equality operator, 39, 77
- error, 226
- error checking, 315

- error handler, 179
- Error Handling, 162, 226
- error handling, 143, 155, 163, 179, 186, 240, 242
- error message, 241
- error messages, 29, 226
- error type, 228
- error-like situations, 232
- Errors, 49, 226
- errors, 241-242
- escape character, 64, 264
- escape sequence, 64
- escaping, 64
- examples of assignment, 83
- `except`, 228
- `except` chain, 228
- `except` clause, 226, 228-230, 241
- `except` clauses, 227, 229-230
- `Exception`, 229
- exception, 241
- exception handling, 226-227
- exception handling framework, 226, 232
- exception handling syntax, 226
- exception object, 228, 230
- `Exception` type, 228
- exception type, 229
- exceptions, 226-227, 232
- executable program, 121
- executable script statements, 191
- executable scripts, 189
- executable statements, 189
- execution of the script, 238
- exercises, 34
- existing sequence, 90
- `exit`, 37
- exiting object, 144
- exiting objects, 145
- Exiting the program, 226
- experienced Python programmers, 33
- expression, 39, 44, 50-53, 62, 66, 70, 96, 167, 202, 205-206, 216
- Expression List, 70
- expression list, 70-71, 94, 205
- expression list*, 196
- expression list syntax, 96
- expression statement, 206, 210
- Expression Statements, 205
- expression statements, 207
- Expressions, 50
- expressions, 55
- extension pack, 24
- Extensions, 26
- extensions, 34
- external libraries, 115
- external library dependencies, 111
- extra attributes, 135
- F**
- f-string, 63, 213-215
- f-string expression, 186, 214, 246
- f-String Expressions, 213
- f-string expressions, 215
- f-strings, 214
- `False`, 51, 56, 60, 227
- Fibonacci sequence, 333
- field or method, 83
- file extension, 30, 38
- file name, 30, 124
- file name conventions, 190

- file name extension, 30
- file names, 190
- file system, 27
- `finally`, 226
- `finally` clause, 229
- `finally` suite, 230
- first statement, 233
- flag, 30
- `float`, 42, 54
- `float` literals, 57
- `float` number, 41, 71
- `float` number objects, 43
- `float` type, 41, 49, 56
- `float` value, 41, 45
- `float` values, 45
- floating point numbers, 39, 43, 45, 59
- floating point numbers*, 66
- floor division, 42, 45
- folder, 27
- folder names, 110
- `for`, 223
- `for - in`, 219, 221
- `for - in` loop, 330-331
- `for - in` statement, 184, 309, 321
- `for` and `while`, 223
- `for` and `while` loops, 219, 225
- `for` and `while` statements, 188
- `for in`, 224
- `for in` loop, 220, 248, 313
- `for in range`, 224
- `for` loop, 223, 226, 236-237, 239, 276, 304, 313, 314, 321-322
- `for` loop compound statement, 219
- `for` or `while`, 223-225
- For Range Loop, 219
- `for range` loop, 245
- `for` statement, 184, 219, 224, 234
- `for` statement body, 236
- formal type systems, 266
- formatted string literal, 213
- fragile, 290
- `from import` statement, 236
- `from import` syntax, 130, 166, 262
- from left to right*, 70, 96
- from left to right, 137, 184
- from right to left*, 86
- from top to bottom, 184
- full power of the IDEs, 23
- full-blown IDEs, 23
- Function, 66
- function, 17, 31, 66-67
- function*, 66
- function argument, 31
- function arguments, 167
- function body, 132-133, 164
- function call, 37, 52-54, 56, 72, 139, 146, 152, 161, 167, 193, 200, 234, 239, 276, 286, 327
- function call and return, 226
- function call* expression, 140
- function call `print()`, 127
- function calls, 206
- function `choice`, 186
- function `def`, 185
- function `def` statement, 133, 138
- function `def` statements, 147
- Function Definition, 131
- function definition, 33, 125, 132, 138, 148, 152,

156, 163-164, 225, 236-237, 279, 314
 function definition statement, 203
 Function Definitions, 199
 function definitions, 140, 144, 160, 183-185, 199, 208, 262, 277, 279
 function execution, 139
 function implementation, 132
 function in Python, 52
 function name, 17, 49, 132, 135, 163, 237
 function names, 131, 147
 function object, 134-135, 147, 237, 279
 function object in memory, 133, 237
 function objects, 133, 147, 280
 function or class definition, 151
 function overloading, 201, 221
 function parameter, 32, 148, 151-152
 function parameter name, 132
 function parameter names, 283
 Function parameters, 168
 function parameters, 163, 201, 238
 function `randint`, 127
 function `random.choice`, 186
 function signature, 204, 282
 function syntax, 297
 function `sys.exit`, 226
 function `type`, 72
 functional programming, 55, 217, 272, 316
 functional programming languages, 261, 266
 functional programming principles, 55
 functional programming styles, 17, 55, 261
 functional programming techniques, 55
 functional syntax, 307-308
 Functions, 52, 169, 253
 functions, 17, 66, 78, 133
 function's return value, 139
G
`Game` class, 260
`game` module, 192, 202, 217, 247, 266, 269
 game of rock paper scissors, 18
`Game` type, 259
 gap of one empty line, 126
 gap of two empty lines, 126
 gaps of two empty lines, 126
 garbage collected, 146
 garbage collection, 218
 garbage collector, 218
 garbage-collected, 217
 general structure, 164
 Generics, 267, 304
 generics, 267, 304, 306
 gentleman's agreement, 290
 gentleperson's agreement, 290
`get` method, 320
`git`, 116, 118
`git`, 116
`git`, 118, 244
`git` commands, 118
`git` program, 116
`git` repository, 117, 120, 173
 GitHub, 116
 GitLab, 116
`global`, 145, 152
`global`, 151
 global function, 91
 grammatically required, 222
 graphical user interface, 27
 grouped statements, 236

- grouping, 70
- guard, 314
- GUI desktop, 26
- GUI programming, 24
- GUI Python tool, 23

H

- `hand` module, 327
- handle the exception, 315
- hard-coded, 202, 234
- hash sign, 126
- hash sign `#`, 131
- hash symbol, 126
- hash symbol `#`, 207
- Haskell, 261
- help doc, 262
- help docs, 215
- `help` function, 92
- help prompt, 137
- help session, 137
- `help()` function, 137
- `hex` function, 134
- hexadecimal, 57
- hexadecimal representation, 134
- high level program structure, 107
- high level structures, 169
- high-level designs, 107
- high-level programming languages, 55
- higher precedence, 159
- hobby programmer, 24
- home, 108
- home directory, 109
- home folder, 110
- homogeneous, 76

- homogenous lists, 199

I

- `id` values, 84
- identifiers, 148, 192
- identity, 17, 43
- IDEs, 20, 23-24, 264
- idiomatic, 222
- idiomatic guard statement, 192
- idiomatic `if` statement, 192
- IDLE, 23
- `if`, 137-139
- `if` "clause", 62
- `if - elif - else`, 216
- `if - elif - else` statement, 141, 239, 330
- `if - elif - else` statements, 310
- `if - else` expression, 216, 218
- `if - else` statements, 141
- `if` clause, 139, 185, 275
- `if` compound statement, 184
- `if` condition, 184
- `if` conditional expression, 220, 273
- `if` expression, 216, 237
- `if` keyword, 62, 269
- `if` line, 62
- `if` or `elif`, 269
- `if` Statement, 137
- `if` statement, 62, 127, 137-140, 142, 144, 150-151, 157, 159, 161, 164-166, 168, 185, 192, 216, 227, 233, 237, 264, 269
- `if` statements, 137, 141-142, 161, 186, 269, 312
- `if` suite, 62
- Immutability, 43

- immutable, 43, 45, 68, 75, 94, 145
- immutable*, 66
- immutable and mutable objects, 147
- immutable custom types, 260
- immutable object, 43-45, 84
- immutable objects, 79, 82, 145
- immutable or mutable, 144
- immutable types, 44, 212, 307
- imperative, 17
- imperative languages, 205
- imperative programming, 17, 79, 217, 272
- imperative programming language, 55
- imperative programming languages, 52, 79, 225
- imperative) programming languages, 223
- implicit variable `_`, 78
- `import`, 262
- import a module, 129, 166
- import path, 194
- Import Statement, 127
- `import` statement, 38, 191
- import statement, 124, 183
- `import` statements, 38, 125, 156, 184, 195, 230, 233
- import the definitions, 128
- import this, 93
- importable module, 190
- importable* modules, 193
- imported module, 128
- imported names, 262-263
- importing modules, 128
- importing names, 166
- `importlib` module, 128
- `importlib.reload` function, 128
- `in`, 245
- `in` keyword, 219
- in production, 32
- increment, 221
- indentation differences, 141
- indentation level, 62, 138
- indentation rules, 138
- indentations, 62, 125, 127, 138, 162, 184-186, 240
- index, 72, 220
- index loop variable, 222
- index notation, 72, 74, 77, 94, 319
- `IndexError`, 86, 88, 165, 241
- `IndexError` exception, 86, 94, 165, 232
- Indexing, 94
- indexing, 85-86, 88-89, 94
- indexing and slicing, 88
- indexing expression, 241
- indexing expressions, 86
- infinite loop, 224-225
- infinite precisions, 43
- infinity, 51
- inheritance, 293
- inheritance, 293, 297, 299, 303
- inheritance-base polymorphism, 304
- inheritance-based polymorphism, 303, 306
- inherits, 277, 292
- `init` parameters, 283
- initial `None` value, 145
- initial value, 143, 165, 198
- initial values, 198
- initialization, 189
- initialization code, 291
- initialization statements, 190

- initialization steps, 189
- initializations, 189
- initializer, 298
- initializer method, 298, 323
- inner function definition, 126
- innermost loop, 225
- input and output, 179
- input expression, 96
- `input` function, 142-143, 232, 241
- `input` function call, 155, 231
- input processing, 242
- input prompt, 232
- `input` statement, 241
- input validation, 171, 242
- `input()`, 155
- `input()` call, 155
- `input()` function, 232
- `input()` function call, 143-144, 155
- input/output handling, 232
- insertion order, 322
- install python, 22
- install Python 3.10, 22, 34
- Install WSL, 25
- instance, 57, 281-282, 300, 309
- instance attribute reference syntax, 285
- instance method, 288, 296, 338-339
- Instance Methods, 288
- instance methods, 289, 296
- instance object, 253, 278, 281-288, 290, 297, 299, 301, 339
- instance object*, 285
- Instance Objects, 285
- Instance objects, 286
- instance objects, 133, 253, 284-290, 328, 338
- instance of a class, 253
- instance of VSCode, 26
- instance variable, 285, 287-288, 296, 298-300, 323, 338
- Instance Variables, 287
- Instance variables, 288
- instance variables, 288, 291, 317
- instances, 339
- instances of the enum type, 308
- instantiated instance object, 291
- instantiation operation, 280
- `int`, 42, 54-55
- `int` and `float`, 48, 55, 66
- `int` argument, 234
- `int` arguments, 221
- `int` literal, 70
- `int` object, 84, 234
- `int` sequence, 236
- `int` type, 49, 56, 66, 77, 294
- `int` value, 41
- integer division, 41
- integer literals, 39, 57, 63
- integer number, 49
- integer numbers, 39-40, 45, 222
- integer sequence, 220-221
- integer value index, 88
- integer values, 45
- Integers, 43
- Integers, 45, 59, 66
- Integrated Development Environment, 20
- intellisense, 129
- inter-module dependencies, 322
- interactive and non-interactive modes, 206
- interactive help session, 137

- interactive mode, 38, 44, 53, 62, 96, 121, 128, 222
- interactive Python interpreter, 51
- interactive shell, 186
- interactive shell prompt, 153
- Interactive vs Non-Interactive Modes, 51
- interfaces, 276
- internal states, 217, 306
- internal) module name, 190
- international characters, 149
- interpolated strings, 214
- interpreter, 47, 51, 54, 96, 233
- invalid index, 86, 94
- invalid negative index, 86
- invoking the Python interpreter, 101
- `isinstance`, 56, 305
- `isinstance` function, 56
- `isinstance` function call, 301
- `issubclass`, 56, 305
- `issubclass` function, 56
- item assignment, 94
- iterable type objects, 321
- iteration, 223, 226, 236, 339
- iteratively, 330-331, 333
- iTerm2, 25

J

- Javascript, 58, 144, 203

K

- kebab case, 190
- key value pair, 317
- key value pairs, 321
- key-based lookup, 321
- key-value pair, 317
- key-value pair collection, 95
- key-value pairs, 319
- keyboard shortcut, 27-28
- KeyboardInterrupt, 226
- `KeyboardInterrupt`, 231, 241
- `KeyboardInterrupt` exception, 231, 268
- `KeyError`, 321
- `KeyError` exception, 320
- keys or values, 322
- keyword, 31, 36
- keyword argument, 32, 90, 200, 220, 238
- keyword argument syntax, 200
- keyword arguments, 199-201, 238
- keyword `as`, 230
- keyword `def`, 131, 163
- keyword `except`, 232
- keyword `for`, 184
- keyword only, 245
- keyword only parameter, 200
- keyword only parameters, 199-200
- keyword phrases, 22
- keyword `while`, 185
- keyword-only, 200
- keywords, 137, 145
- keywords `elif` and `else`, 62

L

- lab session, 124
- language features, 18, 65
- language interpreter, 17
- language reference, 79
- language server, 24
- language servers, 24

- language specifications, 43
- language syntax, 99
- large programs, 150
- larger systems, 327
- leading spaces, 61
- leading underscores, 260
- leading white spaces, 61, 162
- learn Python, 21
- learning programming, 24, 99
- learning Python programming, 18
- learning to program, 28
- legitimate use case, 290
- `len` builtin function, 76
- `len` function, 329
- length, 271
- length function, 272
- length of a list, 74-75
- length* of a tuple, 73
- levels of indentations, 184
- library dependencies, 110
- library* module, 128
- lightweight IDEs, 24
- line*, 264
- line breaks, 161
- line continuation, 264
- line numbers, 28-29
- line-based, 160
- line-based programming language, 264
- Lines, 264
- Lines in Python, 160
- lines of code, 31
- Linux, 22
- Linux distribution, 20, 22
- Linux on WSL, 22
- Linux on your Windows subsystem, 21
- Linux Shell, 25
- Linux terminal, 113
- `list`, 75, 270
- list comprehension, 95
- `list` constructor function, 90
- `list` in Python, 69
- list in Python, 75
- list literal, 74, 77, 92, 94
- list literal syntax, 90
- List Literals, 74
- list method, 78, 81
- list object, 74-75, 77-78, 81-84, 94, 271
- list objects, 92
- List slicing, 86
- `list` type, 74, 78, 94
- list type*, 92
- list value, 74
- `list.append`, 81
- `list.pop`, 81
- lists and tuples, 275
- lists or tuples, 161
- literal syntax, 57, 319
- literals, 39, 55, 57, 95, 214
- literals in Python, 39, 63
- local development computer, 130
- local namespace, 262
- local scope, 147
- logical AND and OR operators, 157
- logical line, 160, 264
- logical true, 40
- logical values*, 55
- logical values, 66
- long string, 64

- long string expressions, 184
- long string literal, 266
- long string literals, 265
- long strings, 63, 265
- loop, 226
- loop body, 222
- loop variable, 219, 222, 321
- looping, 223
- loops, 220
- loose coupling, 327
- loosely typed, 58, 203
- loosely typed*, 271
- loosely typed language, 271
- low level details, 169
- lower**, 153
- lower camel case, 149
- lower case, 242
- lower** method, 155

M

- Mac, 21, 66
- machine learning, 263
- machine learning projects, 111
- main loop, 107
- main rock paper scissors game, 260
- main script, 164, 183, 185-186, 188, 233, 253, 259, 326
- main script module, 196
- mangled names, 291
- master*, 118
- master* branch, 118
- match**, 313
- match - case**, 311
- match - case** statement, 330-331

- match - case** statement syntax, 167
- match - case** statements, 310, 312
- match pattern, 313
- Match** Statement, 310
- match** statement, 260, 310-313, 315-316
- match** statements, 310, 313, 315
- matched case, 313
- matching **case**, 311
- matching variables, 314
- mathematical set, 55
- mathematics, 77
- maybe, 267
- members of the enum, 307
- membership test operator, 245
- memory address, 134
- memory location, 49, 134
- mercurial**, 116
- method attribute, 135
- method attributes, 133, 286, 292
- method call, 82
- method call syntax, 289
- method defined on **list**, 91
- method syntax, 287-288, 297
- methods, 78, 91, 133, 285
- mixed type lists, 76
- modern languages*, 310
- modern programmer's editors, 24
- modern programming, 49, 252
- modern programming languages, 40, 58, 214, 261, 269, 310
- modern programming languages*, 86
- Modern Python, 322
- modern python language*, 322
- modern Python programming*, 55

- modern standard convention, 194
- modular, 170, 276
- modularity, 170
- module, 30, 38, 122, 127, 187-188, 243, 261, 278
- module*, 30
- module import syntax, 166
- module initialization, 191
- module module, 189
- module modules, 190
- module name, 124-125, 190, 195, 233, 262
- module name collisions, 193
- module name prefix, 166
- module name qualifications, 130
- module prefix, 166
- module vs script comparison, 194
- module-level access control, 189
- Modules, 190, 262
- modules, 115, 124, 189, 193
- modules and classes*, 189
- Modules and Packages, 261
- modules and scripts, 188-189
- modules or packages, 188, 190
- modulo, 42
- modulo operation, 86, 88
- modulo operator, 42, 45
- Monty Python's Flying Circus, 16
- more modular, 243
- most current version, 21
- most recent version, 21
- multi variable assignment, 198
- multi-character sequence, 64
- multi-user system, 110
- multiline comments, 207
- multiline string literal, 64, 265
- multiline string literals, 64
- multiple (physical) lines, 161
- multiple assignment, 197
- multiple expressions, 70, 96
- multiple folders, 26
- multiple inheritance, 277
- multiple installations of Python, 22
- multiple instances of VSCode, 26
- multiple lines, 30, 138, 161, 264-265
- multiple matching expressions, 311
- multiple names, 85
- multiple physical lines, 138
- multiple programming languages, 24
- multiple Python source files, 122
- multiple rounds, 233
- multiple terminals, 28
- Multiplication, 41
- multiplication table, 332
- multiplications, 41, 45
- mutability and immutability*, 75
- mutable, 43, 75, 77, 94, 260, 288
- mutable and immutable sequences, 87
- mutable and immutable types, 87
- mutable complex type, 319
- mutable list objects, 94
- mutable object, 83
- mutable object*, 85
- mutable objects, 79, 83
- mutable objects*, 82
- mutable or immutable, 44
- mutable or immutable types, 87
- mutable sequence, 88
- mutable type, 80, 203, 318
- mutable types, 84

mutating operations, 94

N

name, 80, 143

name `_`, 73, 78, 222

name aliases, 263

`name` and `value`, 309

name binding, 144, 147, 167, 184

name bindings, 147, 185, 233

name collision, 166

name collisions, 263

name conflicts, 166

name for an object, 79

name hiding, 291

name mangling, 291

name `randint`, 130

name `self`, 283, 289

`NameError`, 146, 148

names, 52, 79-80, 82, 148

names and assignment, 147

names in Python, 149

names/references, 146

namespaces, 193, 263

naming convention, 149, 234

Naming Conventions, 149

naming conventions, 190

native language, 149

negative index, 86

Negative indexes, 86

negative number indexes, 88

nested `for` loops, 332

nested `if-else` statements, 142

nested loops, 225

nested namespaces, 194

`new`, 280

new `enum` class, 308

new file, 27

new function, 131, 163

new list, 87, 90, 95

new `list`, 319

new list object, 212

new name, 234

new project, 108

new sequence, 90

new `set`, 319

new type, 297

new types, 309

new version of Python, 21

new way of programming, 33

newline, 30, 37, 64, 160-161, 220

newline character, 64, 264

newline symbols, 161

newlines, 64, 160, 266

newly created instance object, 292

no side effect, 55

no-op expression statements, 209

no-op operation, 292

non virtual, 295

non-Boolean expression, 58

non-definition statements, 189

non-descriptive name, 150

non-empty list, 275

non-empty lists, 270

non-interactive mode, 38, 44, 52, 61, 96, 128,
184, 206

non-numerical expression, 271

non-optional keyword parameters, 201

non-optional positional arguments, 201

- `None`, 51, 54-56, 60, 66-67, 81, 96, 140, 145, 230, 268, 276, 325
- `None` Object, 54
- `None` value, 66
- `NoneType`, 54-55
- `NoneType` type, 55
- `nonlocal`, 145, 152
- normal function call syntax, 287
- normal shell prompt, 112
- `not`, 157-158
- `not` expression, 269
- `not in`, 245
- `not` operator, 139
- `not`, `and`, and `or`, 269
- Notepad, 23
- `null`, 66, 80, 145
- null pointer exception, 145
- null pointer exceptions, 80
- `null` value, 80
- number literal, 273
- number objects, 43
- number of items, 321
- number types, 55, 59, 66
- numbers, 39
- numbers in programming, 43
- Numbers in Python, 47
- numeric arguments, 212
- numeric literal, 273
- Numeric literals, 63
- numeric literals, 149, 273
- numeric objects, 136
- numeric type, 45, 48
- numerical context, 40, 51
- numerical types, 45
- object, 17, 48, 83, 190, 253, 279
- object*, 77, 276
- `object`, 278, 293-294
- object in Python, 43, 58, 133, 301
- object* in Python, 263
- object method calling syntax, 286
- object methods, 277
- object orient programming style, 105
- object oriented, 17
- Object Oriented Programming, 276
- object oriented programming, 252, 276
- object oriented styles, 252
- object variables, 277
- object-centered, 79
- object-centric, 83
- object-oriented programming language, 290
- Objects, 144
- objects, 43, 58, 66, 80, 133, 276
- objects*, 91, 144
- objects in Python, 57
- Objects of simples types, 43
- objects of the `list` type, 91
- objects of the simple types, 68
- Objects' identities, 78
- object's identity, 134
- octal, 57
- one and the same object, 285
- one empty line, 125
- one file, 122
- one file - one program, 122
- one instance variable, 317
- one letter strings, 220
- one or more underscores, 290

- one program, 122
- one statement, 31
- one underscore, 291
- one workspace, 26
- one-element tuple, 71
- one-empty-line gap, 208
- one-line gap, 235
- online poker games, 337
- OOP, 252, 276, 293, 306
- OOP features, 260
- OOP in Python, 286, 296, 303
- OOP languages, 277-278, 280, 286, 289
- OOP paradigms, 252
- OOP programming languages, 253, 280
- open VSCode, 27
- open-source project, 215
- operand types, 212
- operands, 41, 71
- operator, 31, 212
- operator **not**, 157
- operator overloading, 212
- operator precedence, 159, 212
- operator precedence rule, 212
- operator precedence rules, 137
- operators, 137
- optimizations, 242
- option, 267
- Optional and Union Types, 266
- optional arguments, 247
- optional **else** clause, 226
- optional keyword argument, 91
- optional keyword-only parameters, 201
- optional parameter, 202
- optional parameters, 202

- optional positional parameters, 201
- Optional** type, 204, 261
- optional wildcard **else** clause, 229
- or**, 157-158
- or** expression, 160
- or** operation, 158, 160
- or** operator, 158
- or** operators, 137
- original list, 87
- original sequence, 87, 91
- other virtual environments, 111
- output** module, 185, 235, 238, 246
- overall structure, 185
- overloaded, 212
- override, 294
- overriding, 296
- Overriding a method, 294
- overwriting, 294

P

- package, 125, 194-195, 262, 322
- package**, 125
- package folder, 194
- package in Python, 193
- Package Install, 113
- package managers, 116
- package module**, 125
- package module, 125, 194, 253, 322
- package modules, 115, 189, 262
- Package names, 194
- package structure, 194
- package-based module names, 195
- packages, 110, 124-125, 189, 193
- pair of curly braces, 125, 214

- pair of double quotes, 63
- pair of matching triple double quotes, 64
- pair of *matching* triple single quotes, 64
- pair of parentheses, 17, 31, 53, 69-70, 133, 167, 168, 169, 170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 270, 271, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 346, 347, 348, 349, 350, 351, 352, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 372, 373, 374, 375, 376, 377, 378, 379, 380, 381, 382, 383, 384, 385, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 417, 418, 419, 420, 421, 422, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476, 477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580, 581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787, 788, 789, 790, 791, 792, 793, 794, 795, 796, 797, 798, 799, 800, 801, 802, 803, 804, 805, 806, 807, 808, 809, 810, 811, 812, 813, 814, 815, 816, 817, 818, 819, 820, 821, 822, 823, 824, 825, 826, 827, 828, 829, 830, 831, 832, 833, 834, 835, 836, 837, 838, 839, 840, 841, 842, 843, 844, 845, 846, 847, 848, 849, 850, 851, 852, 853, 854, 855, 856, 857, 858, 859, 860, 861, 862, 863, 864, 865, 866, 867, 868, 869, 870, 871, 872, 873, 874, 875, 876, 877, 878, 879, 880, 881, 882, 883, 884, 885, 886, 887, 888, 889, 890, 891, 892, 893, 894, 895, 896, 897, 898, 899, 900, 901, 902, 903, 904, 905, 906, 907, 908, 909, 910, 911, 912, 913, 914, 915, 916, 917, 918, 919, 920, 921, 922, 923, 924, 925, 926, 927, 928, 929, 930, 931, 932, 933, 934, 935, 936, 937, 938, 939, 940, 941, 942, 943, 944, 945, 946, 947, 948, 949, 950, 951, 952, 953, 954, 955, 956, 957, 958, 959, 960, 961, 962, 963, 964, 965, 966, 967, 968, 969, 970, 971, 972, 973, 974, 975, 976, 977, 978, 979, 980, 981, 982, 983, 984, 985, 986, 987, 988, 989, 990, 991, 992, 993, 994, 995, 996, 997, 998, 999, 1000
- 163, 167, 260, 278
- pair of single quotes, 63
- pair of square brackets, 72
- pairs of parentheses, 72
- parameter, 167
- parameter list, 199-200, 287
- parameter names, 164
- parameter=value* syntax, 199
- parameterized types, 267
- parameters, 85, 199-201
- parent class, 293-294, 296
- parent classes, 295
- parent directory, 26
- parent folder, 124, 194
- parent type, 300
- parentheses, 17, 70, 94, 139, 159, 167, 293
- particular extensions, 26
- parts of a program, 127
- Pascal case, 149
- Pascal case names, 149
- Pascal-case, 190
- PascalCase names, 260
- pass*, 36
- pass by reference, 144, 167
- pass by value, 167
- pass* statement, 36, 62, 209-210, 311
- pass* statements, 132
- pattern matching, 260, 310, 314
- pattern matching*, 313
- pattern types, 316
- peer to peer, 116
- peer to peer source control system, 116
- peer-to-peer systems, 116
- PEP 8, 126, 138
- Perl, 131
- physical line, 265
- physical lines, 127, 160
- physical structure, 264
- pip, 113
- pip* CLI command, 114
- pip* command line tool, 113
- pip* module, 114
- platform, 24, 26-27
- play* function, 184, 188
- Play Rock Paper Scissors, 101
- play()* function, 184
- play_a_round* function, 184, 193
- pointers, 79-80
- polymorphic, 212
- Polymorphism, 302
- polymorphism, 221, 304
- polymorphisms, 303
- pop*, 91
- pop* method, 81-82
- popular programming languages, 58
- portability, 149
- positional and keyword, 200
- positional argument syntax, 200
- positional arguments, 200-201
- positional only, 200, 245
- positional only arguments, 154
- positional only parameters, 199-200
- positional or keyword arguments, 201
- positional-only, 200, 238

- positive index, 88
- power operator, 246
- PowerShell, 25, 113
- PowerShell on Windows, 20, 27
- precisions, 43
- predefined attributes, 280-281
- predefined variable `_`, 96
- prefixes, 63
- previous value, 52
- primitive types, 66
- `print`, 52
- `print` function, 30, 36, 52, 127-128, 142, 186, 206, 266, 296, 300
- print statement, 314
- `print()` call, 53
- `print()` function, 37, 54, 220, 235
- `print()` function call, 54
- `print()` function calls, 333
- `print()` statement, 161, 219, 246
- `print()` statements, 62
- Private Members, 290
- private members, 290
- private/internal* static function, 291
- problem solving, 339
- problem solving skills, 100
- procedural, 17
- procedures, 52
- processes, 24
- Professional software developers, 115
- professional software developers, 108
- program, 121-122
- program comment, 126
- program control, 139, 229
- program errors, 50
- program execution, 155, 234
- program file, 38, 138, 264
- program files, 62, 96, 121
- program logic, 259, 310
- program run, 84, 185, 336
- program source code, 337
- program source file, 28
- program source files, 115
- program state, 83
- program text structure, 125
- Programmers, 17
- programmers, 34
- programmers using Windows, 25
- programmer's editors, 23, 264
- programmer's intention, 310
- programming, 17, 38-39, 47, 66, 73
- programming errors, 185
- programming in Python, 18, 33
- programming language, 17-18, 24, 80
- programming languages, 25, 43, 63, 74, 77, 110, 133, 135, 144, 148, 184, 197, 226, 233, 261, 270, 290, 292, 295-296, 317
- programming on Windows, 25
- programming project, 100, 111
- programming projects, 335
- programming techniques, 18
- programming terms, 40, 48
- programming with Python, 19
- programs, 122, 184
- project folder, 107, 110-111
- projects, 110
- prompt, 36
- pseudo-random number generator, 155
- public API, 232, 290, 323

- Public APIs, 127
- public git server, 117
- pull request, 216
- pure functional programming, 217
- pure functional programming languages, 55
- pure functions, 55
- PyPi package, 125
- PyPi packages, 113, 125
- Python, 58, 296
- `python`, 22
- Python "program", 193
- Python builtin function, 73
- Python CLI tool, 34
- Python code, 66
- Python code file, 243, 261
- Python command, 30
- python command, 22, 28-29, 35, 47, 124
- python commands, 22
- Python definitions, 188
- Python development, 108
- Python distributions, 21-23, 111, 113
- Python Enhancement Proposals, 126
- Python expression, 40
- Python expressions, 68
- Python expressions and statements, 161
- Python Extension, 25
- Python extension, 26
- Python file, 30, 128, 194
- Python grammar, 188, 222
- Python in Visual Studio Code, 26
- Python Installation, 20
- Python integers, 43
- Python Interactive Session, 46, 65
- Python interactive session, 35
- Python interactive shell, 23, 37, 52, 81, 96
- Python interactive shell prompt, 38
- Python interpreter, 28, 33, 36-37, 44, 49-50, 52, 62-63, 70-71, 82, 96, 111, 121, 125, 130-133, 146, 150, 184, 188, 190, 204, 206-207, 234, 238, 260, 278, 282-283
- Python interpreter implementation, 130
- Python interpreter program, 79
- Python interpreter tools, 20
- Python keyword, 62, 289
- Python keyword `as`, 262
- Python keyword `class`, 260
- Python keyword `import`, 128
- Python keywords, 137
- Python knowledge, 339
- Python language, 126, 199, 207, 232
- Python language construct, 125
- Python language syntax, 265
- Python module, 127, 187, 261
- python module, 191
- Python Modules, 188
- Python modules, 189, 193
- Python object, 253
- Python objects, 279
- python on ubuntu, 22
- Python Packages, 193
- Python program, 16-17, 29, 31, 37, 52, 121, 124-125, 188, 231-232, 264
- Python program file, 123, 188
- Python program files, 117, 189, 233
- Python programmers, 33, 91, 138, 283
- Python programming, 18, 83, 99
- Python programming style, 199
- Python programs, 36, 52, 80, 95, 110, 121-122,

- 126, 149-150, 160, 184, 204, 233-234, 240, 262, 271
- Python projects, 33, 110-111
- Python prompt, 36, 47
- Python Releases for macOS, 21
- Python Releases for Windows, 21
- Python REPL, 35, 37, 44, 53-54, 56, 60-61, 65, 68, 72, 91, 93, 129-130, 133, 137, 153
- Python script, 29-30, 38, 124, 127, 186-187, 207
- Python scripts, 121-122
- Python shell, 38, 46, 69, 161
- Python shell prompt, 38
- Python source code files, 180
- Python standard formatting rules, 126
- Python standard library module, 114, 155
- Python statements, 44, 68
- Python style guideline, 62
- Python syntax, 33, 96, 108
- Python the Programming Language, 224
- Python tools, 46
- Python typing, 76, 204, 266, 271
- Python typing framework, 266
- Python typing system, 261, 267
- python version, 21
- Python version 2.x, 22
- Python virtual environment, 113, 120
- Python with VSCode, 26
- `python3`, 22
- Pythonic, 222
- Python's comment, 264
- Python's import system, 188
- Python's `None`, 80
- `quit`, 37
- R**
 - raise an error, 241
 - `raise` an exception, 227
 - raise an exception, 227, 230, 268
 - raise errors, 51
 - `raise` statement, 231
 - raising an error, 232
 - raising an exception, 227
 - `randint` function, 165
 - `randint()` function, 166
 - `random`, 155
 - random computer hand, 240
 - random hand, 102, 186
 - random hand generator, 336
 - random integer, 210
 - random integer number, 156
 - Random Module, 155
 - `random` module, 127, 129-130, 156, 186, 211, 262-263, 329
 - `random` module functions, 339
 - random numbers, 210
 - `random.choice`, 211
 - `random.choice` function, 325
 - `random.randint`, 130
 - `random.randint` function, 130, 156, 210
 - range, 86-87, 94-95, 223
 - `range` function, 221
 - range function call, 236
 - `range(start, stop)`, 221
 - `range(start, stop, 1)`, 221

- `range(stop)`, 221
- rational number, 49
- readability, 51, 126, 150
- `reader` module, 268
- reading material, 34
- real function*, 52
- real number, 49
- real number values, 56
- real numbers, 39, 66
- real programming, 99
- real programming experience, 99
- real programs, 120
- real software development, 99
- real world problems, 29
- real-world programming problems, 18
- real-world projects, 100
- rebind an existing name, 83
- rebinding, 146
- Recursion, 271-273
- recursion, 272-273, 316
- recursion logic, 275
- Recursions, 272
- recursions, 272
- recursive algorithm, 273
- recursive calls, 316
- recursive case, 273-274
- recursive function, 272
- recursive process, 105
- recursively, 330-331
- reference, 80, 83, 143
- reference*, 95
- reference to the returned object, 85
- reference types, 79
- referenced object, 263
- references, 80
- references*, 144
- related statements, 126
- relative import, 195
- relative import syntax, 196
- relearning process, 21
- relearning processes, 21
- remainder, 42
- remainder after floor division, 45
- Remote - WSL extension, 24
- Remote Development extension pack, 24
- remote server, 116
- REPL, 40, 46, 68, 96, 323
- REPL prompt, 153
- requirements of OOP, 290
- result of an expression, 52
- `return`, 229, 276
- `return` statement, 138-140, 164, 225
- `return` statements, 240
- return value, 67, 82, 139-140
- returned list, 87
- returned object, 85
- Rock Paper Scissors, 18, 101, 105, 122
- rock paper scissors, 18
- rock paper scissors 2, 194
- Rock Paper Scissors app, 106
- rock paper scissors app, 162, 192
- rock paper scissors apps, 107
- rock paper scissors game, 19, 32, 99-100, 105, 110, 114, 124, 162, 170, 173, 180, 242, 252, 260, 336-337, 341
- rock paper scissors game implementation, 326
- rock paper scissors game logic, 239
- rock paper scissors game rules, 127

- rock paper scissors game version 2, 244
- rock paper scissors game, version 2, 231
- rock paper scissors implementation, 262
- rock paper scissors program, 111, 122, 131, 148, 153-154, 159, 163, 167, 169-170, 172, 177, 179, 185-186, 202, 215, 225, 232, 246, 249, 250, 259, 268, 273, 312, 322, 338
- rock paper scissors program v3, 307
- rock paper scissors program version 2, 217
- rock paper scissors program version 3, 292, 309
- Rock Paper Scissors project, 100
- rock paper scissors project, 114-115, 120
- rock paper scissors rounds, 221, 324
- rock papers scissors game, 341
- rock, paper, and scissors, 211
- `rock_paper_scissor` module, 225, 239
- `rock_paper_scissors` module, 185, 196, 203, 211, 236
- round brackets, 69
- round of rock paper scissors, 236, 239
- round-off errors, 43
- `rps` module, 326
- `rps` package, 262, 327
- `rps.Game` class, 291
- `rps.game` module, 276, 322-323
- `rps.hand` module, 292, 322, 326-327
- `rps.rand` module, 325
- `rps.reader` module, 312, 325
- `rps.result` module, 292
- `rps2/rock_paper_scissors` module, 192
- run time, 165, 267
- runnable* script, 128
- runnable script, 183, 190
- running a Python program, 30
- runtime errors, 271
- runtimes, 233
- same attributes, 282
- same identity, 44
- same indentations, 62
- same items*, 87
- same list object, 84
- same object, 44, 84-85, 215
- same object*, 84
- same suite, 62
- same value, 84
- sample code, 21, 28, 34
- sample output, 54
- sample outputs, 47
- sample programs, 47
- scope, 151, 165
- Scopes, 151
- scoping rules, 152
- Screenshots, 28
- screenshots, 28
- script, 29, 37, 96, 122, 131, 233, 243
- script*, 30
- script file, 190
- script file name, 180, 186
- script languages, 131
- script module, 186
- script modules, 189-191
- script or module, 128
- script statements, 190
- scripting, 264
- scripts, 96, 121, 180, 193

- scripts and modules, 150
- search box, 26
- secret of Python, 59
- `self`, 154, 289, 296
- `self` naming convention, 283
- `self` object, 288
- `self` parameter, 283
- `self` variable, 290
- semantically constants, 198
- semantics, 78, 85
- semicolon, 161
- semicolons, 96
- separate lines, 30
- sequence, 90, 219, 226
- sequence object, 90
- sequence objects, 89, 319
- sequence of statements, 52, 66
- Sequence Replication, 212
- sequence type, 74, 86, 186, 211, 245, 307
- sequence type object, 72
- sequence type objects, 213
- sequence types, 197
- sequence unpacking, 197
- sequences, 73
- series of statements, 55, 277
- server, 115
- `set` type, 319
- `set()` function, 319
- setup steps, 108
- shell, 25
- shell prompt, 21, 37, 47, 112
- shell prompt `$`, 113
- shell script `activate`, 112
- Shell scripting, 131
- shell scripting, 25, 264
- shell scripts, 27, 264
- short circuit, 158, 160
- short circuiting, 158
- short scripts, 30
- short string literal, 265
- short string literals, 64
- short strings, 65, 265
- short) strings, 64
- short-circuiting, 158
- short-circuiting rule, 165
- short-circuits, 160
- side effect*, 52, 55
- side effect, 52-54, 206
- side effects, 52, 55, 62, 206-207, 209
- simple, 62
- Simple and Compound Statements, 60
- simple builtin types, 66
- simple commands, 30
- simple literal, 319
- simple literals, 68
- simple program, 99
- simple Python program, 34
- simple Python script, 23
- simple statement, 31, 127
- Simple statements, 96
- simple statements, 60, 83
- simple type literals, 68
- simple types, 43, 66, 68
- single compound statement, 126-127
- single expression, 71
- single file scripts, 188
- single installation of VS Code, 24
- single line command, 30

- single *logical line*, 264
- single* object, 72
- single object, 85
- single physical line, 264
- single *physical line*, 264
- single Python file, 188
- single Python source file, 128
- single quote, 65
- single quote pairs, 64
- single quoted string, 215
- single quoted string literals, 64
- single quotes, 63-64, 208, 214-215
- single round, 233
- single value, 71
- slice operation, 86
- slices, 87
- Slicing, 86, 94
- slicing, 86-88, 91, 94
- slicing range, 88
- slicing syntax, 89
- small project, 108
- small Python program, 26
- snake case, 149, 163
- snake case names, 149-150
- snake-case, 190
- snake-case module names, 189
- software architecture, 327
- software components, 327
- software development, 20, 108
- software engineering, 55
- software project, 18, 108-109, 115
- software projects, 108
- Software testing, 32
- `sort`, 91

- `sort` method, 94
- `sorted`, 91
- `sorted` function, 90, 94
- `sorted` function call, 90
- sorted sequence, 91
- `sorted()` function, 91
- Sorting, 89
- sorting behavior, 91
- source code, 115-116
- source code changes, 117
- source code file, 122
- source code files, 115, 122, 253
- Source Control System, 115
- source control system, 115-116
- source control systems, 115
- source files, 187, 233, 253
- source* Unix command, 112
- Spaces, 160
- special name `_`, 81
- special syntax, 57, 63
- special syntaxes, 63
- specialized IDEs, 24
- spoken languages, 183
- square brackets, 74, 77, 92, 94
- stable version, 21
- stack overflow error, 272, 274
- standard deck, 329
- standard deck of cards, 329
- standard desktop distribution, 20
- standard formatting rules, 126
- standard libraries, 126
- standard library, 32, 130, 262, 327
- standard library `random` module, 124, 210
- standard library `venv`, 111

- standard module `random`, 129
- standard modules, 130
- standard naming conventions, 190
- standard output, 142
- standard Python interpreter, 114
- star operator `*`, 212, 235
- start IDLE, 23
- start index, 89
- `start` method, 259
- start slice index, 89
- starting script, 233
- `startswith`, 153
- statement, 17, 31, 36, 51-52, 82, 216
- statement in Python, 37
- statements, 17, 31, 55, 61-62, 188
- statements or expressions, 51
- states, 217
- static fields, 280
- static functions, 277
- static methods, 280, 289
- static type, 267
- static type checkers, 267
- static type checking, 305
- static variables, 280
- statically and strongly typed languages, 304
- statically typed, 58
- statically typed languages, 32, 267, 271, 295, 302, 304, 315
- statically typed OOP languages, 303
- statically typed programming languages, 32, 135, 267, 302
- `stdout`, 52
- `step`, 221
- `step` argument, 221
- storyline, 104
- `str`, 63
- `str` function, 215
- `str` type, 153, 198-199, 203
- `str.format` function, 215
- `str.lower` method, 155
- `str.startswith` method, 165
- string, 17, 31, 63
- string concatenation, 137
- string concatenation expression, 161
- String Concatenations, 64
- string concatenations, 65, 214-215, 265
- string constants, 261
- string context, 292, 296
- string expression, 235
- string expressions, 184
- string interpolations, 214
- string literal, 63, 206-208, 214-215
- string literal syntax, 213
- string literals, 63, 68, 207, 265-266
- string object, 142, 155, 215
- string objects, 275
- string representation, 164
- String slicing, 86
- `string` type, 17
- string type, 92, 338
- string value, 199
- Strings, 331
- strings, 30
- Strings in Python, 63
- strongly typed, 58, 203
- strongly typed imperative programming languages, 266
- structural pattern matching, 310

- structured Python programs, 276
- subclass, 57, 297, 301
- subclass of `Exception`, 228
- subfolders*, 180
- subfolders, 187
- Sublime Text, 23
- submodule, 195
- submodules, 193
- submodules of a single package, 194
- subroutines, 52
- subscript operator, 72-73, 86, 94, 320
- Subtraction, 41
- subtractions, 41, 45
- subtype, 45, 48, 55, 269, 293, 299
- subtype of `int`, 56, 293
- subtype of `object`, 277, 293
- subtyping, 228
- suite, 61-62, 160, 219
- suite*, 223
- `sum` function, 272
- surfaces, 276
- `switch - case` statement, 311
- `switch` statement, 260, 310-311
- `switch` statement in C, 310
- symbol `_`, 52
- syntactic difference, 78
- syntactic sugar, 65, 287
- Syntactically, 214
- syntactically, 161, 209
- syntactically valid, 132, 162-163
- syntactically valid statements*, 49
- syntax, 32, 77, 95, 130
- syntax highlighting, 23
- `SyntaxError`, 228

- `sys` module, 226
- `sys.exit()`, 232
- system exceptions, 228, 231
- system path, 26
- T**
- tabs, 64, 160
- Tasks, 107
- tasks, 34
- team environment, 115
- temporary folder, 27
- Terminal, 25
- terminal, 16-17, 28, 52, 54
- terminal app, 20
- terminal program, 25
- terminal programs, 20
- terminating the program, 226
- ternary operator `? :`, 216
- text, 17
- text editor, 117
- text editor for programming, 23
- text representation, 131
- the parameter `self`, 287
- the `rps.game` module, 326
- the string literal, 208
- third party libraries, 110, 125
- third party library dependencies, 114
- third party) libraries, 113
- `this`, 289
- tilde symbol, 110
- top level `.gitignore` file, 117
- top level* scope, 151
- top-level folder, 108
- `toString` methods, 292

- `touch` Unix command, 117
- trailing colon, 60
- trailing comma, 70-71, 75
- trailing `else` clause, 224
- trailing newline, 142
- triple quote long strings, 208
- triple quote strings, 184, 208
- triple quoted string, 208
- troubleshoot, 29
- troubleshooting, 34
- `True`, 56, 60, 227
- `True` and `False`, 40, 45, 55, 57, 66
- `True` or `False`, 56-58, 141, 216, 269
- truth value, 270, 275, 306, 328
- truth value*, 270
- Truth Values, 306
- truth values, 261, 275-276, 306, 328
- `try - except`, 227
- `try - except - finally`, 226
- `try - except` statement, 186
- `try - except` statement, 241
- `try` clause, 227-228, 241
- `try` compound statement, 226
- `try` statement, 229-231, 240
- `try` suite, 229, 241
- `tuple`, 69, 186
- tuple, 70, 72, 94, 196, 314
- tuple argument, 247
- tuple comprehension, 95
- `tuple` in Python, 69
- tuple literal, 69, 71-72, 74, 94
- tuple literal syntax, 70
- Tuple Literals, 69
- tuple literals, 69
- tuple object, 73, 75, 94
- `tuple` of `int` and `int`, 73
- tuple parameter, 239
- tuple patterns, 315
- Tuple slicing, 86
- Tuple Type, 71
- `tuple` type, 94
- tuple type, 247, 266
- tuple types, 267
- Tuple Unpacking, 196
- tuple unpacking, 321
- tuple value, 196
- tuple-like syntax, 232
- tuples, 70, 75, 314
- tuples and lists, 76
- two empty lines, 125-126, 132
- two line gaps, 126
- two players, 340
- two underscores, 291
- type, 17, 39, 48-49, 58, 90, 277, 302
- `type`, 301
- type annotation, 76, 143-144, 151, 165, 197, 234, 267, 290
- Type Annotations, 32
- type annotations, 32-34, 73-74, 76, 131, 164-165, 198, 203-204, 271
- type annotations for functions, 198, 204
- type annotations for variables, 198
- type `bool`, 55, 66
- `type` function, 48
- type `function`, 134
- type `Game`, 259
- type hint, 32, 198

- type inference, 198
- type inheritance, 293, 309
- type `NoneType`, 66
- type of an expression, 58
- type `rps.Game`, 260
- type support features, 306
- type system, 315
- type validation, 58
- type-related errors, 271
- `TypeError` exception, 136
- types, 34, 39, 48-49, 57-58, 68, 73, 271
- types in Python, 49
- typical OOP, 296
- typing, 32-33, 203
- `typing`, 32
- typing constraints, 204
- typing framework, 317
- typing *module*, 73
- `typing` module, 267
- typing support, 199
- typing system, 199
- `typing.Union` class, 266

U

- Ubuntu, 20, 22-23
- Ubuntu 20.04LTS, 20
- Ubuntu 22.04, 47
- Ubuntu subsystem, 24
- unary Boolean operator, 157
- unary `not` operator, 269
- `undefined`, 66
- `undefined` in Javascript, 66
- underscore, 27, 273
- underscore discard variable, 272

- underscore symbol, 149
- underscore-prefix name rule, 190
- underscores, 149, 189, 194
- underscores `_`, 273
- unexpected error, 163
- Unicode, 149
- union type, 266-267
- Union types, 267
- union types, 76, 204, 261, 266
- unique identities, 77, 79
- Unix, 29
- Unix shell, 20, 30
- Unix shell commands, 25
- Unix shelling scripting, 264
- Unix-like platform, 117
- Unix-like platforms, 231-232
- Unix-like systems, 25-26, 37, 109, 190
- Unix/Linux platforms, 66
- Unpacking, 197
- unpacking, 196-197
- unpredictable situations, 232
- unused variables, 222
- upper camel case, 149
- UpperCamelCase, 260
- user defined objects, 133
- user experience, 185
- user friendly, 179
- user hand, 240
- user input, 127, 142, 165, 169, 171, 179, 325
- user input value, 143
- user-defined object, 302
- user-defined types, 59, 69, 150, 212, 260, 277, 288

V

- valid assignment, 145
- valid expression*, 49
- valid expression, 51, 135, 137
- valid identifier, 222
- valid index, 94
- valid indexes, 86
- valid input, 186, 242
- valid* Python script, 205
- valid statement, 51
- valid string, 242
- value, 17, 39, 50-51, 58
- value **None**, 140
- value of `_`, 54, 73
- value of a function or method call, 82
- value of a pair of parentheses, 69
- value of a tuple, 75
- value of an expression, 53
- value of an expression list, 196
- value of an object, 94
- values, 39, 78, 80
- variable, 52, 58, 79-80, 95, 143
- variable `_`, 53, 77, 96
- variable declaration, 184
- variable definitions, 185
- variable in Python, 83
- variable name `_`, 222
- variable names, 190
- variable-centered, 79
- variable-centric, 83
- variable-centric view, 217
- variables, 27, 52, 58, 79-80, 96, 198, 262
- variables*, 144

- variables in Python, 79, 95

- Variables/Names, 144

- venv**, 111-112, 114, 173

- venv folder, 112

- venv** module, 112

- version control system, 115, 117

- Version control systems, 118

- version control systems, 115-116

- vertical bars, 266

- vertical bars `|`, 311

- vim, 23

- virtual, 295-296

- virtual environment, 110-113, 130

- Virtual Environments, 110

- virtual environments, 110-111

- Visual Studio Code, 23

- VS Code, 23-24, 34, 117, 129, 264

- VSCode, 24-26, 117

- VSCode keyboard shortcuts, 28

W

- war card game, 341

- war game, 341

- Web search, 22-23

- while**, 224

- while** Boolean expression, 223-224

- while Boolean expression, 225

- while** compound statement, 226

- While Loop, 223

- while** loop, 186, 223, 225-226, 231, 248

- while** loop statement, 333

- while** loops, 223

- while** statement, 185, 223-224, 240

- while** **True** statement, 240

- white space, 160
- white spaces, 29, 51, 61, 264
- whole sequence, 87
- wildcard expression `_`, 311
- wildcard `from import` syntax, 185
- wildcard import, 238
- wildcard import syntax, 131, 189
- wildcard module import, 260
- wildcard name import, 166
- wildcard syntax, 131
- Windows, 20-21, 66, 113
- Windows File Explorer, 27
- Windows host machine, 24
- Windows Store, 25
- Windows Subsystem for Linux, 20, 25
- Windows Terminal, 25
- Windows Terminal app, 20, 25
- Windows Terminal by Microsoft, 25
- Windows users, 24
- working environment, 34
- working space, 111
- Workspace, 108
- workspace, 26
- workspaces, 26, 117
- Writing a program, 104
- WSL, 20
- WSL on Windows, 21, 24

Z

- `ZeroDivisionError` exception, 51
- ZSH, 112

Credits

Images

All drawings used in this book are taken from undraw.co, an amazing service with an amazing open source license. Many thanks to the creator of the site: twitter.com/ninaLimpi!

Icons

All emoji icons used in this book are from fontawesome.com. Fontawesome is a very popular tool, probably used by almost everyone who does Web or mobile programming.

Typesetting

Here's another absolutely fantastic software, asciidoctor.org, which is used to create an ebook as well as paperback versions of this book. [AsciiDoc](https://asciidoctor.org) [https://asciidoctor.org] is like Markdown on steroid. You can follow them on Twitter: twitter.com/asciidoctor.

Other Resources

The author has relied on many resources on the Web in writing this book, in particular, [Python 3 Documentation](https://docs.python.org/3/) [https://docs.python.org/3/]. If the book includes any material from these resources, then the copyright of those content belong to the respective owners.

About the Author

Harry Yoon has been programming for over three decades. He has used over 20 different programming languages in his professional career. His experience spans from scientific programming and machine learning to enterprise software and Web and mobile app development.

You can reach him via email: harry@codingbookspress.com.

He occasionally hangs out on social media as well:

- TikTok: [@codeandtips](https://tiktok.com/@codeandtips) [https://tiktok.com/@codeandtips]
- Instagram: [@codeandtips](https://www.instagram.com/codeandtips/) [https://www.instagram.com/codeandtips/]
- Facebook Group: [Code and Tips](https://www.facebook.com/groups/codeandtips) [https://www.facebook.com/groups/codeandtips]
- Twitter: [@codeandtips](https://twitter.com/codeandtips) [https://twitter.com/codeandtips]