

BEGINNER
HANDS-ON PYTHON SERIES

HANDS-ON PYTHON

with 162 Exercises, 3 Projects,
3 Assignments & Final Exam



M U S A A R D A

HANDS-ON

PYTHON

with 162 Exercises, 3 Projects, 3 Assignments &
Final Exam

BEGINNER

Musa Arda

OceanofPDF.com

Hands-On Python with 162 Exercises, 3 Projects, 3 Assignments & Final Exam: Beginner

by Musa Arda

Copyright © 2021 Musa Arda.

All rights reserved. No portion of this book or its supplementary materials may be reproduced in any form without permission from the publisher, except as permitted by U.S. copyright law.

For permissions contact: python.hands.on.book@gmail.com

OceanofPDF.com

Contents

Preface

1. Introduction
2. Getting Started
3. The First Program
4. Variables
5. Functions I
6. Project 1 – Functions
7. Assignment 1 – Functions
8. Conditional Statements
9. Functions II
10. Loops
11. Strings
12. Project 2 - Words
13. Assignment 2 – Words
14. List
15. Dictionary
16. Tuple
17. Set
18. Comprehension
19. Project 3 - Snake Game

20. Assignment 3 - Snake Game

21. Final Exam

22. Conclusion

OceanofPDF.com

Preface

About This Book

This book is an in-depth and activity-based introduction to Python programming. It follows a unique approach by combining the theory of the language with the hands-on coding exercises. It will start from the absolute zero point, from ‘Hello World!’, and will cover all of the fundamental concepts. You will learn and practice all the basics of Python programming in this book.

Here is what you will find in this book:

Theory: In each topic, we will cover all the Theoretical Details with example coding.

True/False (42 questions): We will have True/False questions at the end of the sub-topics.

Coding Exercises (120 questions): At the end of each chapter, we will have Coding Exercise, Quizzes, of 10 questions.

Projects (3): We will have 3 Projects in this book. You will learn how to apply Python concepts on real world problems.

Assignments (3): After each project, you will have an Assignment. These assignments will let you build the project from scratch on your own.

Final Exam: At the end of this book you will have the Final Exam. It is a multiple choice exam with 20 questions and a limited

duration. The exam will let you to test your Python level.

About the Author

Musa Arda has Bachelor's degree from Industrial Engineering in 2007, and he has been working as a Software Developer for more than 14 years. He uses many programming languages and platforms like; Python, C#, Java, JavaScript, SAP ABAP, SQL, React, Flutter and more.

He creates online learning content and writes books to share his experience and knowledge with his students. His main purpose is to combine theory and hands-on practice in his teaching resources. That's why there are hundreds of programming exercises, quizzes, tests, projects, exams and assignments in all of his courses and books. He is dedicated to help his students to learn programming concepts in depth via a practical and exiting way.

How to Contact Us

Please feel free to get in contact with the author. To comment or ask technical questions about this book, you can send email to python.hands.on.book@gmail.com.

1. Introduction

Who Is This Book For?

The goal of this book is to help students to learn Python programming language in a hands-on and project-based manner.

With its unique style of combining theory and practice, this book is for:

- people who are new to computer science and programming
- people who are new to Python
- people who want to gain solid Python Skills
- people who want to learn Python by doing Projects, Quizzes, Coding Exercises, Tests and Exams
- people who want to learn Python for Machine Learning, Deep Learning, Data Science and Software Development in general
- anyone who wants to learn real Python

What Can You Expect to Learn?

The purpose of this book is to provide you a good introduction to Python programming. In general, you will gain solid programming skills and grasp the main idea of software

development. In particular, here are some highlights on what you can you expect to learn with this book.

You will:

- learn & master Programming Fundamentals, Coding Algorithms and Computer Science Concepts
- go from Beginner to Intermediate level in Python with hands-on approach
- do exercises on all fundamental topics of Python with 12 Quizzes and 120 Coding Exercises
- build 3 Real-World Project with Python and do 3 Assignments related to these projects
- take the Final Exam on Python with 20 questions to assess your learning
- build Python applications with Anaconda and Jupyter Lab and master them
- learn Python fundamentals you need for Machine Learning, Deep Learning, Data Science and Application Development
- gain solid and profound Python Programming skills needed for a Python career

Outline of This Book

In [Chapter 1](#), you will get the basics of the book. You will learn about this books approach to Python programming and how to get

most out of it.

In [Chapter 2](#), after a brief history of Python, you will set up your development environment. You will install Python and Anaconda. You will learn Jupyter Notebook basics and virtual environments in Python.

In [Chapter 3](#), you will write your first program and say “Hello World” with Python. You will learn arithmetic operations in Python as well as values and types. You will have your first Programming Quiz in this chapter.

In [Chapter 4](#), you will learn variables and how to define them in Python. You will learn Python Data Types and how to write comments in Python. You will also learn numeric operations, string operations and Python keywords.

In [Chapter 5](#), will meet functions in Python. This will be the first part of functions in this book. You will learn how to define your own functions. You will also learn math functions, parameters & arguments, scope, return statement and docstrings.

In [Chapter 6](#), you will have your first project. You will use Python’s built-in Turtle Module to draw shapes on the screen. It will be fun and give you the chance to use the concepts you learned so far.

In [Chapter 7](#), you will have your first assignment. It is based on the Turtle Project in Chapter 6 but with missing code segments in it. Don’t worry, you will have the guidelines to complete the project on your own.

In [Chapter 8](#), you will learn conditional statements in Python. You will learn if, else and elif structures and how to use them. You will also learn nested conditionals and recursion.

[Chapter 9](#) is the second and more detailed part of functions in Python. You will learn how to compose multiple functions, how to nest them, how to pass unknown parameters, how to use the lambda function and how to define functions that return other functions.

In [Chapter 10](#), you will learn loops in Python. Loops are one the main building blocks of programming and you will learn them in great detail. You will learn while and for loops, how to use nested loops, indexing, enumerate, break and continue structures in loops.

In [Chapter 11](#), you will learn strings in Python. You will learn string slicing, indexing and string methods. You will also learn how to compare strings and loop over them.

In [Chapter 12](#), you will have the second project. It is mainly on using functions, string operations and loops.

[Chapter 13](#) is your second assignment. This time you will try develop the project in Chapter 12 on your own by completing the missing code segments.

In [Chapter 14](#), you will learn another building block of Python programming, the List. List is one the most important data structures in Python. You will learn how to define lists, how to loop

over them, how to slice and index lists, how to use list methods and how to add elements to the list or delete elements from it.

[Chapter 15](#) is about another important data structure in Python, the Dictionary. You will learn how to create dictionaries, how to add elements to a dictionary, how to delete elements, and how to loop over dictionaries.

In [Chapter 16](#), you will learn Tuples. Tuples are one of the important data structures. You will learn how to create tuples, how to use tuple assignment and the zip() function.

[Chapter 17](#) is about Sets in Python. A set is a special data structure with some advantages in certain cases. You will learn how to define and use them as well.

In [Chapter 18](#), you will learn Comprehensions. Comprehensions are very powerful and versatile structures in Python. Your life will be much easier if you learn how to use them. Don't worry you will learn them and practice a lot on them in this chapter.

[Chapter 19](#) is your third project. You will be building the famous snake game in Python. You will create your own game and have fun playing it. With this project, you will have the chance to practice almost every concept you learn in this book.

In [Chapter 20](#), you will have your third assignment. It will be your turn to develop the snake game on your own. You will have hints and guidelines to complete the project.

[Chapter 21](#) is your Final Exam. Now it's time to assess your Python level. You will have an exam of 20 questions on the topics you learned in this book. The Final Exam will give you the chance to see what you learned and which topics you should repeat.

[Chapter 22](#) is the Conclusion. You will finalize this book and will see how you should proceed for the next step.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new URLs, email addresses, filenames, and file extensions.

Bold

Indicates new terms and important concepts to pay attention.

Constant width

Used for string literals and programming concepts within paragraphs.

Constant width bold

Used for program elements within paragraphs, such as variable or function names, data types, statements, and keywords.

We will write our code in the code cells. They refer to the actual cells in JupyterLab. Here is an example code cell with the cell number as 7:

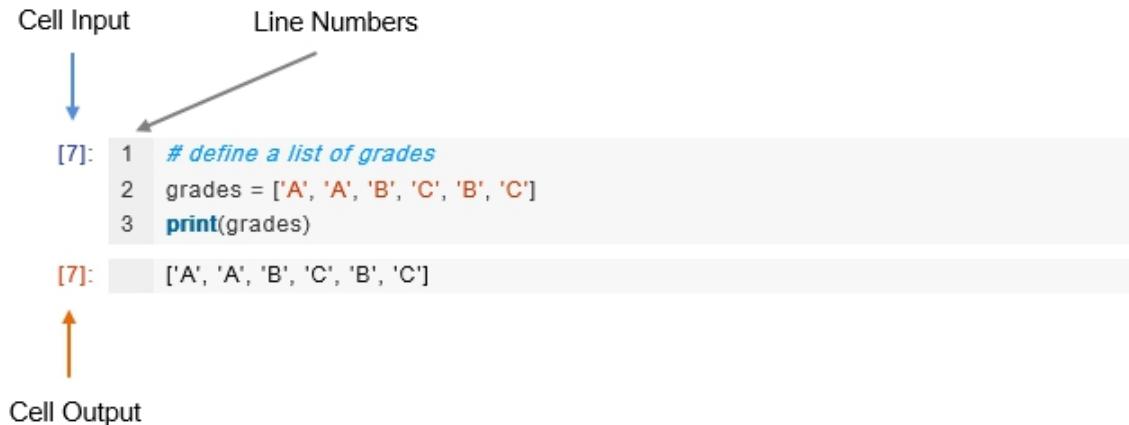


Figure 1-1: A code cell example used in this book

Using Code Examples

You can find all the supplementary resources for the book (starter files, quizzes, assignments, final exam etc.) available for download at <https://github.com/musaarda/python-hands-on-book-beginner>.

You & This Book

This book is designed in a way that, you can learn and practice Python. At each chapter you will learn the basic concepts and how to use them with examples. Then you will have a quiz of 10 questions at the end of the chapter. First you will try to solve the quiz questions on your own, then I will provide the solutions in detail. You will have projects after each block of core concepts. And after every project you will have an assignment to test your understanding. This will be your path to learn real Python.

Before we deep dive into Python, I want to give you some tips for how you can get most out of this book:

- Read the topics carefully before you try to solve the quizzes
- Try to code yourself while you are reading the concepts in the chapters
- Try to solve quizzes by yourself, before checking the solutions
- Read the quiz solutions and try to replicate them
- Code the projects line by line with the book
- Do the assignments (seriously)
- Do not start a new chapter before finishing and solving quiz of the previous one
- Repeat the topics you fail in the Final Exam
- Learning takes time, so give yourself enough time digest the concepts

2. Getting Started

A Brief History of Python

[Python](#) is a widely used and general-purpose, high-level programming language. Its design philosophy emphasizes code readability and its syntax allows programmers to express concepts in fewer lines of code. Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including procedural, object-oriented and functional programming.

It was initially designed by [Guido van Rossum](#) in the late 1980s and developed by [Python Software Foundation](#). Python was named after the BBC TV show Monty Python's Flying Circus.

In 1991, Van Rossum published the code labeled version 0.9.0. Then in 1994, Python reached version 1.0.

Python 2.0 was released in 2000, which started the Python 2.x family that has been used widely for almost a decade. The final release, Python 2.7.18, occurred on 2020 and marked the [end-of-life of Python 2](#). As of January 1st, 2020 no new bug reports, fixes, or changes will be made to Python 2, and Python 2 is no longer supported.

Python 3.0 (also called "Python 3000" or "Py3K") was released on December 3, 2008 and it was a major, backwards-incompatible release. Since Python 3.0 broke backward compatibility, much

Python 2 code does not run unmodified on Python 3. Although some old projects still use Python 2, the preferred approach is using Python 3. As of this writing, the latest version is 3.9, but every code we write in this book should run on Python 3.6 or any later version.

Why Anaconda and JupyterLab?

In this book we will be using JupyterLab (Jupyter Notebook) for our development environment. It is easy to learn and due to its cell-based structure you will be able to run and check every line of code you learn in this book. Don't worry, we have a section on Jupyter Notebook basics. You are strongly recommended to use JupyterLab to follow this book. All of your quizzes, assignments and even the final exam will be in Jupyter Notebooks.

Anaconda is the platform that includes Python and JupyterLab. So with installing Anaconda, you will get Python and JupyterLab on your machine. That's why we need Anaconda for this book.

It's up to you to choose any platform or IDE for Python development. But for this book, things will be much easier if you use Anaconda and JupyterLab.

Installing Anaconda

To begin Python programming we have to install Python and a development environment (IDE - Integrated Development Environment) on our machine. We can install Python directly from

www.python.org but to make things easier for you, we will not do this. Instead, we will install [Anaconda](#) which is one the most popular platforms for Python distribution. Anaconda comes with built-in Python in it, so you do not need a standalone Python installation. Anaconda also includes JupyterLab which will be our IDE for this book.

To install Anaconda, navigate to www.anaconda.com. This is the official web site of Anaconda, and here under Products, select the **Individual Edition**. Individual Edition is free and is more than enough for every development we do in this book.

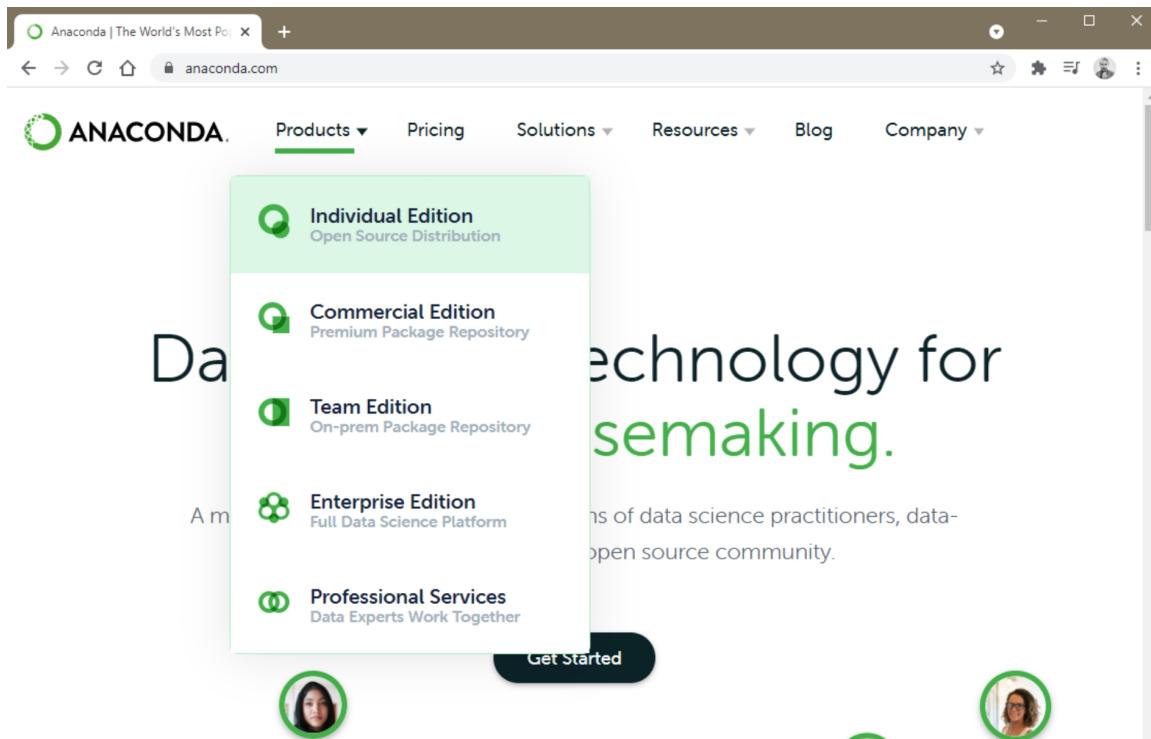
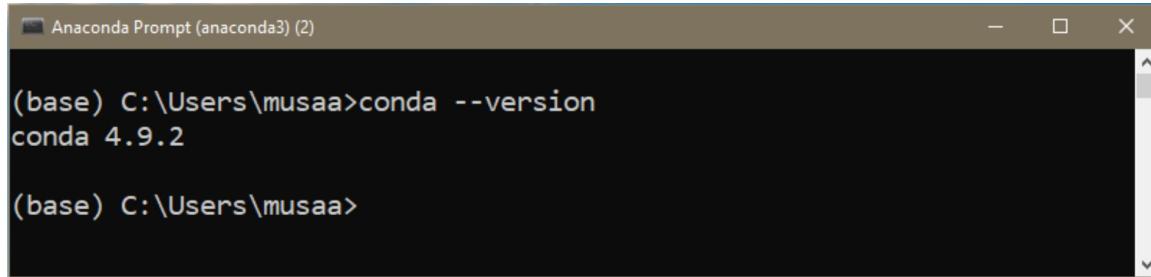


Figure 2-1: Anaconda official web site and Individual Edition

Anaconda Individual Edition contains [conda](#) and [Anaconda Navigator](#), as well as Python and hundreds of scientific [packages](#).

When you install Anaconda, you have all of them installed on your machine.

conda works on your command line interface (CLI) such as Anaconda Prompt on Windows and terminal on macOS and Linux.



```
(base) C:\Users\musaa>conda --version
conda 4.9.2

(base) C:\Users\musaa>
```

Figure 2-2: Anaconda Prompt command line interface on Windows

Navigator is a desktop graphical user interface (GUI) that allows you to launch applications and easily manage conda packages, environments, and channels without using command-line commands.

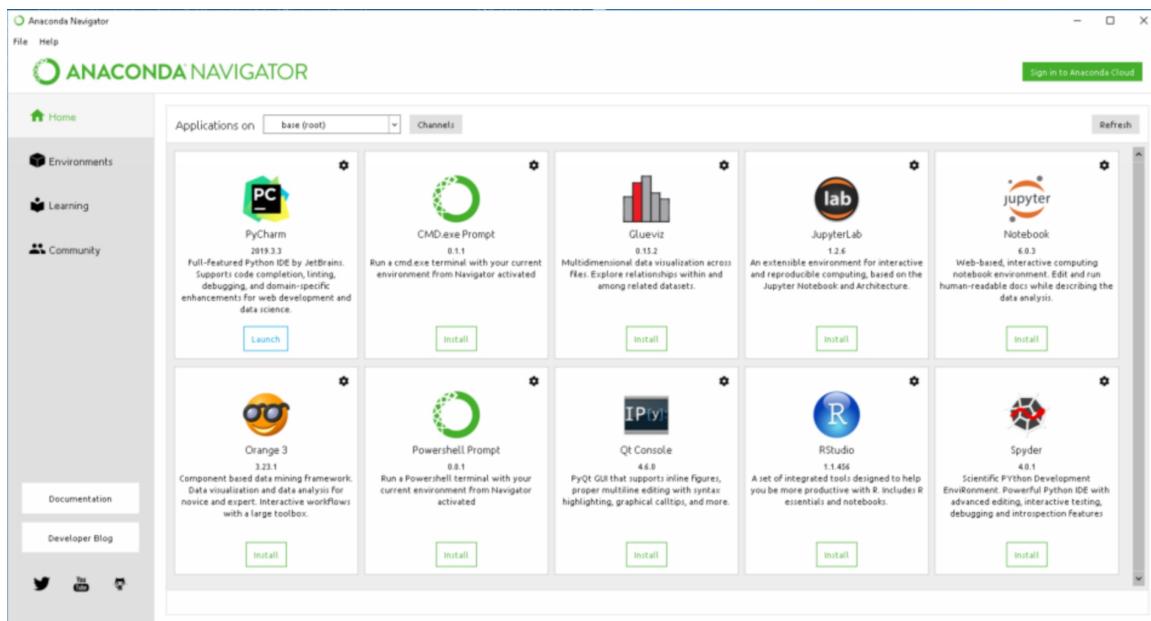


Figure 2-3: Anaconda Navigator

You can use both conda and Navigator to write code and manage your packages and environments. You can even switch between them, and the work you do with one can be viewed in the other. In this book, we will mainly use Conda Prompt to launch the JupyterLab.

Let's install Anaconda now. Once you download the Individual Edition, it's very easy to install it. I will be demonstrating the steps on Windows, but if you are using macOS or Linux, [here](#) are the steps for those operating systems.

When you start the installation process the first screen is the Welcome screen. Just click Next to proceed to setup.

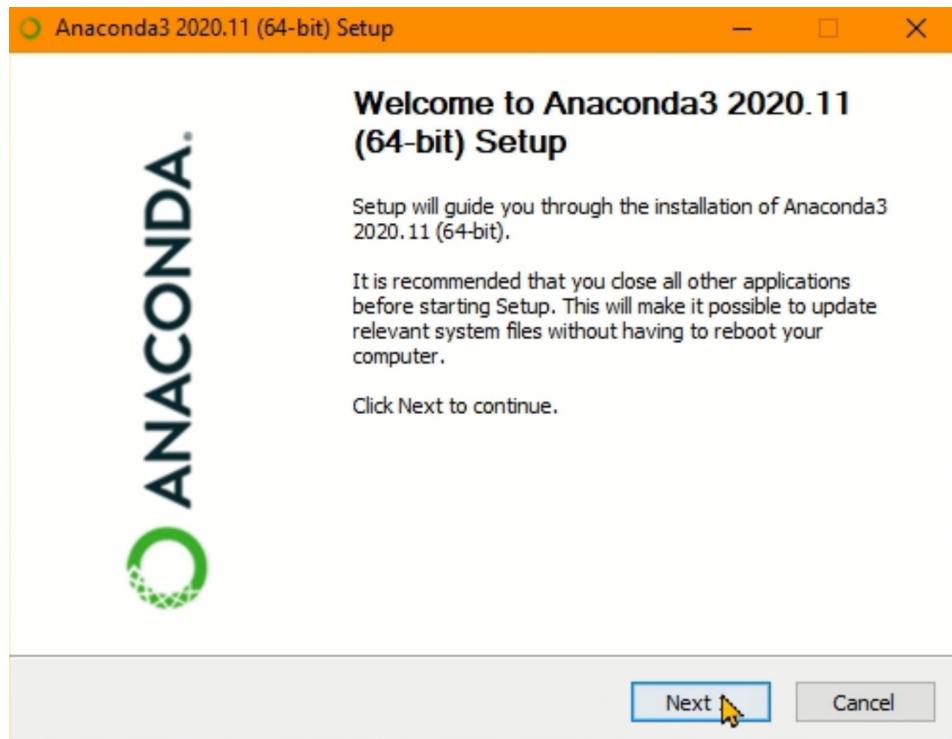


Figure 2-4: Anaconda Setup initial screen

The next screen is the License Agreement. You should click “I Agree” to proceed.

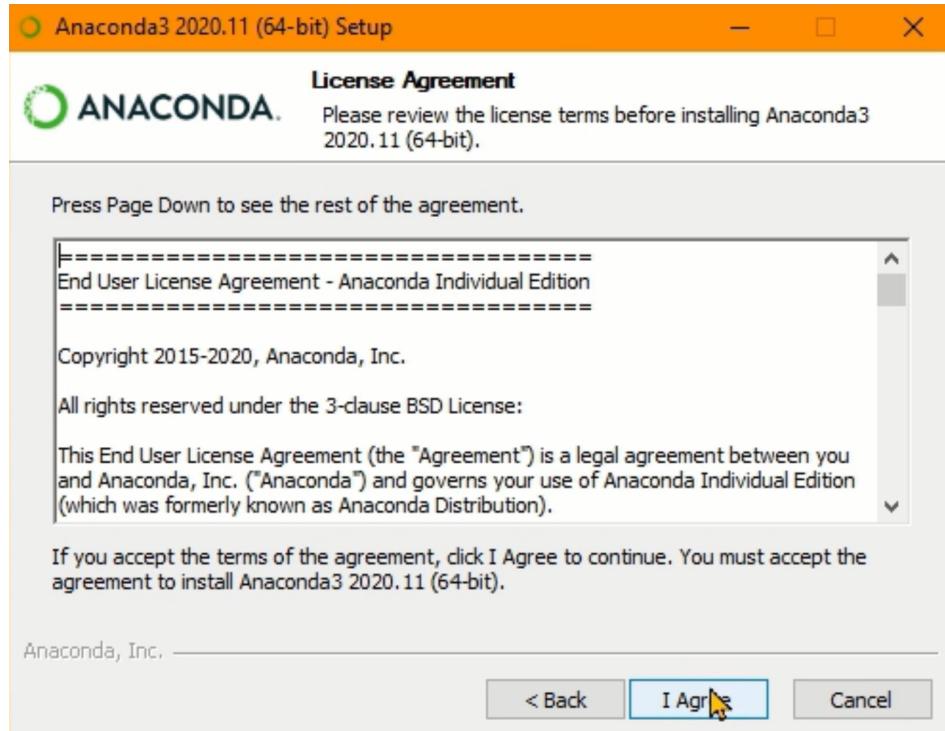


Figure 2-5: Anaconda License Agreement

The next screen is for selecting the installation type. You should decide the user group to install for. It's up to you to select either “Just Me” or “All Users”. And you should click Next again to proceed.

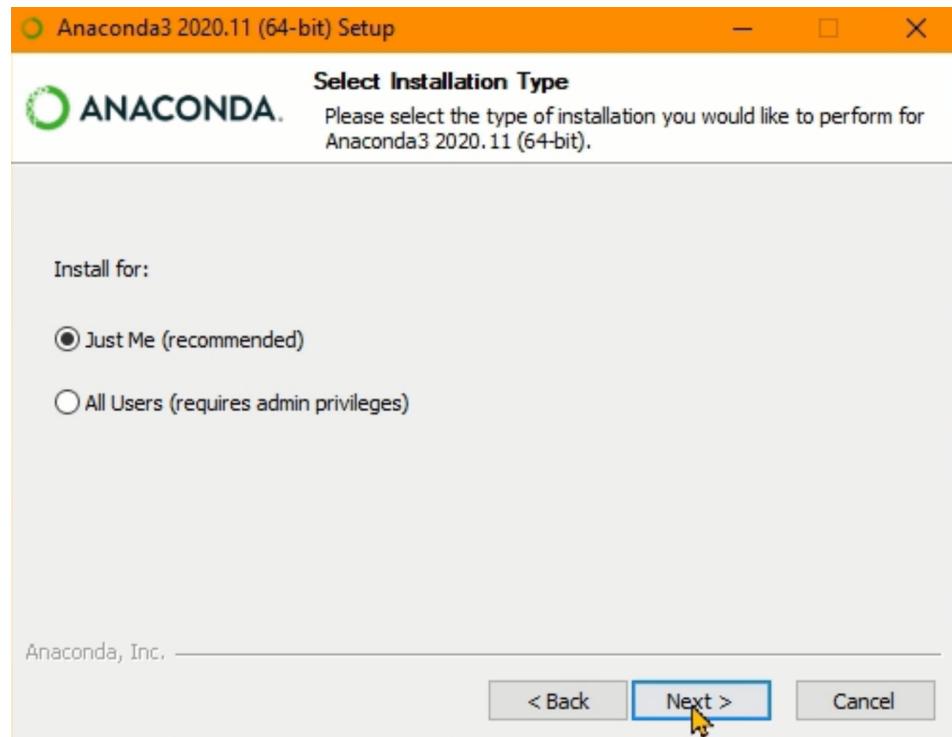


Figure 2-6: User Group selection

In the next screen you will select the installation folder. I do not recommend to change the location. It is up to you but I strongly recommend to move on with the default one.

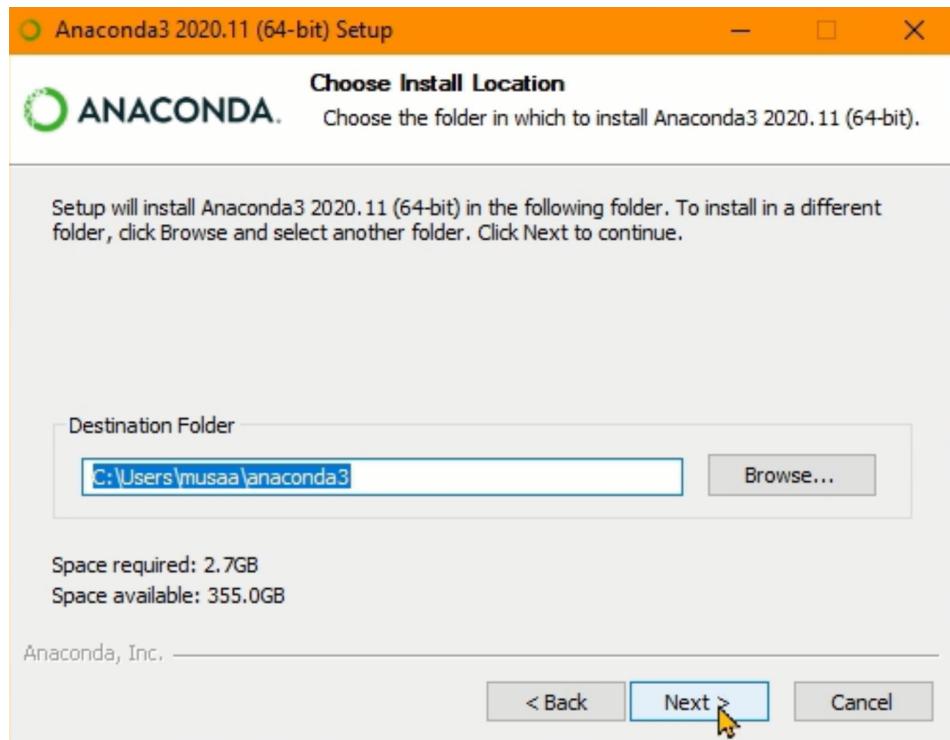


Figure 2-7: Location selection

The next screen is a very important step. There are two options here. The first one is about the PATH environment variable. It asks you to add Anaconda to your PATH variable. You should select it, because we want to run Anaconda prompt directly from command line interface. (See *Figure 2-2* above). To be able to run “**conda** ...” prompts we need to add Anaconda to the PATH.

The second option is about registering Anaconda as the default Python interpreter. You may have different installations of Python on your machine. You can either check this option or not, it’s up to you. And if you check, it will not affect other Python installations. I prefer to check it and continue.

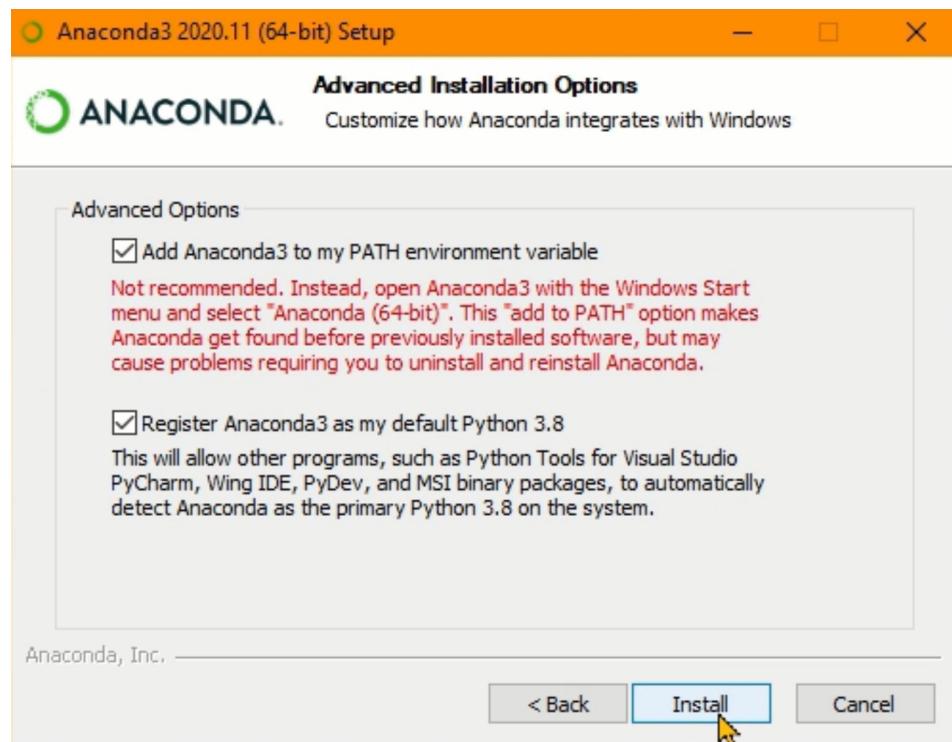


Figure 2-8: Advanced installation options

And now the installation will start. It may take several minutes depending on your machine configuration. After it finishes installing, just click on “Next and Finish” to finalize installation.

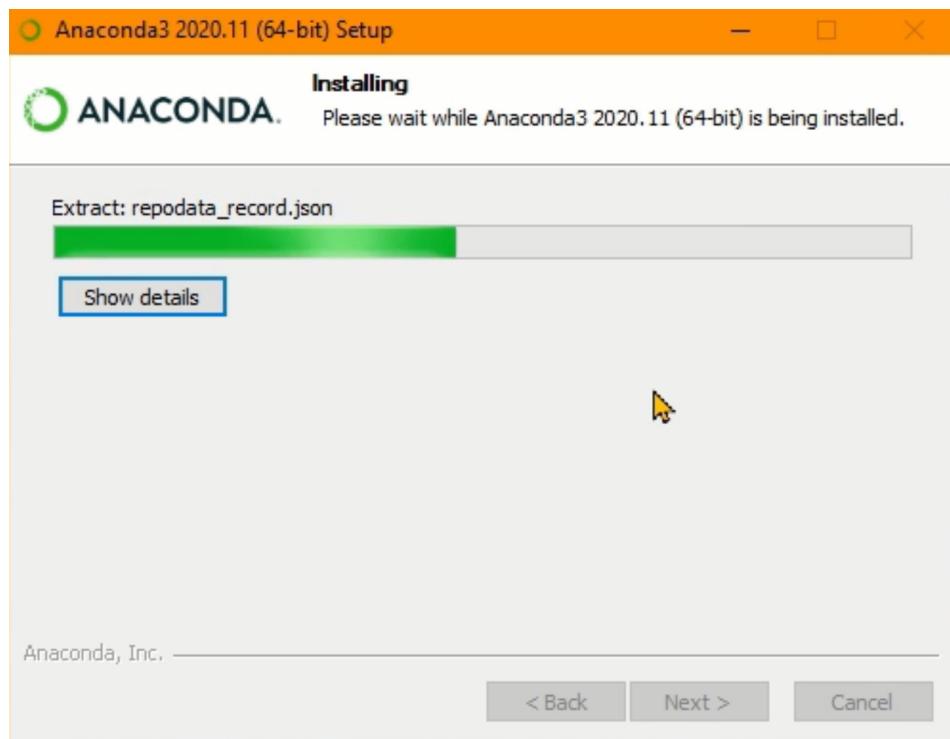


Figure 2-9: Installation progress

And when the installation is complete, Anaconda Navigator may not start automatically, so you have to start it from the start menu (on Windows). In the start menu search for “Anaconda” and launch Anaconda Navigator.

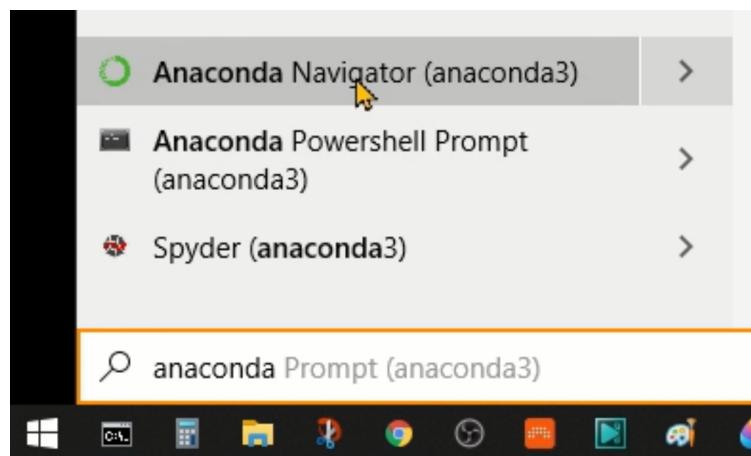


Figure 2-10: Anaconda Navigator in Windows start menu

When you launch Anaconda Navigator for the first time, it may popup some info screens, don't worry about them. And when it starts, you may see the main navigator screen as below.

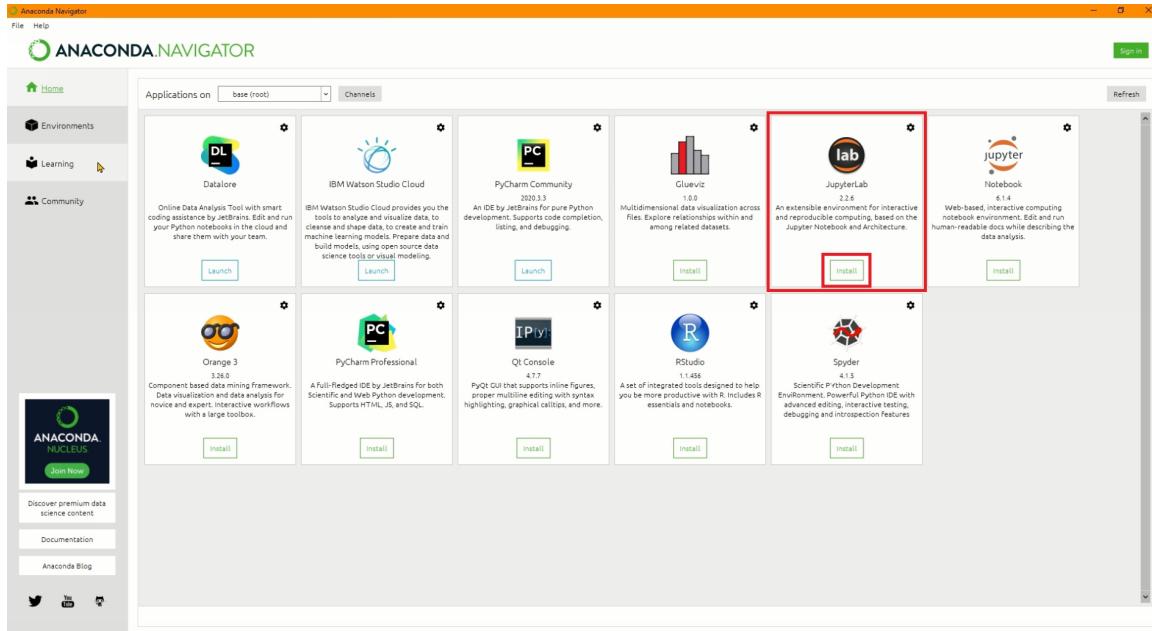


Figure 2-11: Anaconda Navigator main screen

On the left side menu of navigator you see Home, Environments, Learning and Community menus. We will talk about virtual environments in Python later in this chapter. In the Home page, you see the applications bundled in Anaconda Individual Edition. Some of them are already installed, and the others need to be installed. For this book we only need JupyterLab, so will install it.

Jupyter notebooks are documents that combine live runnable code with narrative text (Markdown), equations (LaTeX), images, interactive visualizations and other rich output. And JupyterLab is the web-based interactive development environment for Jupyter

notebooks. So when we install JupyterLab, we will have Jupyter notebooks too. Since JupyterLab gives us more ability to manage our folders and files more easily we will be using it as our IDE for this book.

On the main navigator screen, click on the “Install” button for the JupyterLab. Anaconda will install and setup JupyterLab for you. When the installation is completed you will see the “Launch” button for JupyterLab. And we are ready to start Python development now.

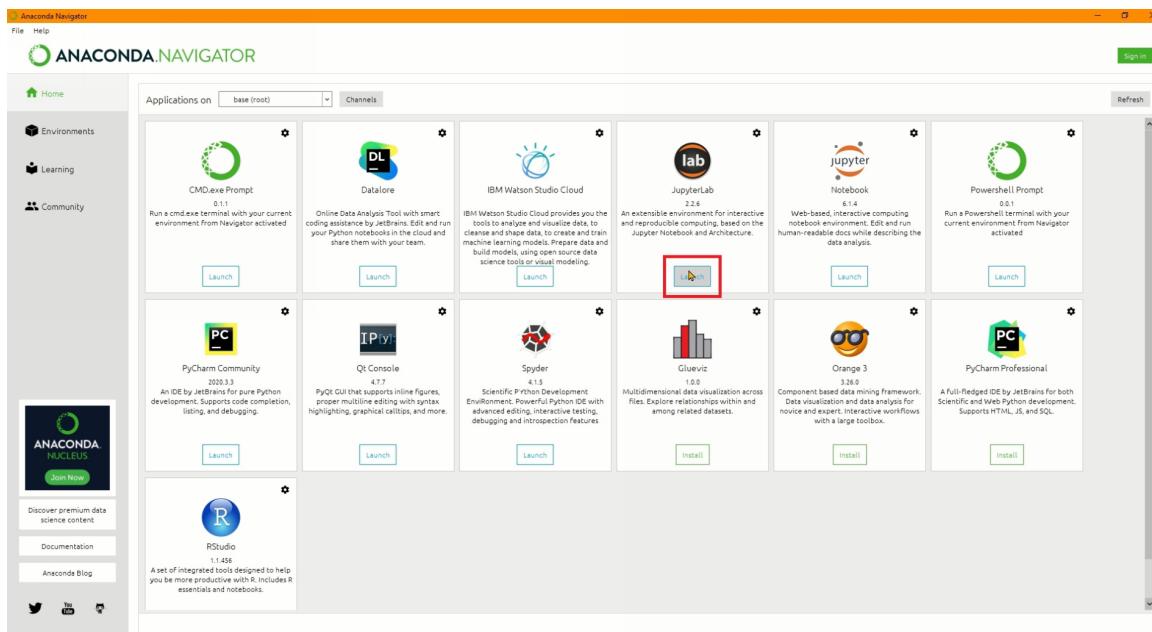


Figure 2-12: JupyterLab after successful installation

When you launch JupyterLab, it will open the web interface in your default web browser. By default it will start on port 8888. If this port is already in use on machine it will pick the next available port as 8889, 8890 or 8891 and so on. If we assume the port is 8888 then the url in the web browser will be: **localhost:8888/lab**.

When JupyterLab starts for the first time, it may ask for Password/Token. In the next section you will learn how to fix this.

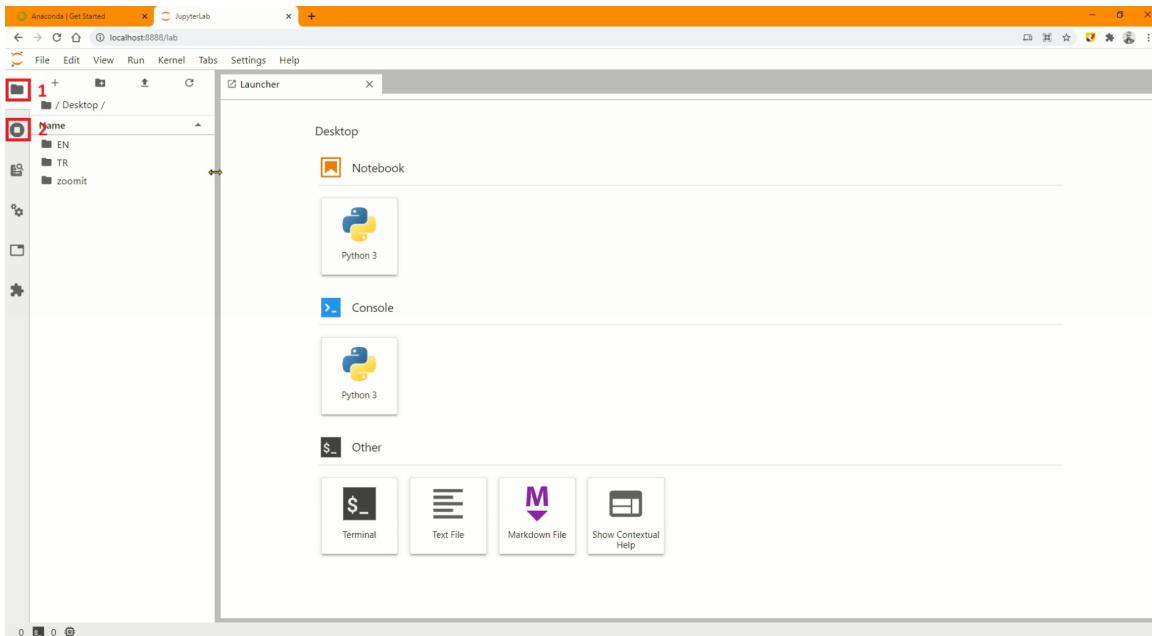


Figure 2-13: JupyterLab web based development environment

On the left side of JupyterLab, you see the File Browser (1) to organize your files and folders and Running Terminals and Kernels (2) to manage the kernel sessions, which we will see later. And on the right hand side you see the code editor, which we will be using to write Python code.

You can create, rename, copy or delete files and folders in the file browser. It works like the regular file operations on your operating system. To follow the topics more easily, I recommend you to copy the entire book content folders and files from [GitHub repository](#) on your machine. When you copy the files to your local machine, please make sure you can access them via JupyterLab file browser. And here is the complete folder structure for this book.

Name	Type
1_Introduction	File folder
2_Getting_Started	File folder
3_The_First_Program	File folder
4_Variables	File folder
5_Functions_I	File folder
6_Project_1_Functions	File folder
7_Assignment_1_Functions	File folder
8_Conditional_Statements	File folder
9_Functions_II	File folder
10_Loops	File folder
11.Strings	File folder
12_Project_2_Words	File folder
13_Assignment_2_Words	File folder
14_List	File folder
15_Dictionary	File folder
16_Tuple	File folder
17_Set	File folder
18_Comprehension	File folder
19_Project_3_Snake_Game	File folder
20_Assignment_3_Snake_Game	File folder
21_Final_Exam	File folder
22_Conclusion	File folder

Figure 2-14: Folder structure

To test whether your Anaconda and JupyterLab installation works let's create a Jupyter notebook file. Notebook files ends with **.ipynb** extension and are specific to Jupyter environment. You have to open them inside JupyterLab.

To create the **.ipynb** file in JupyterLab, navigate to **2_Getting_Started/1_Anacoda/** folder and on the right hand side click on “Python 3” button under the Notebook section. It will open up an empty notebook file, **Untitled.ipynb**, in a new tab.

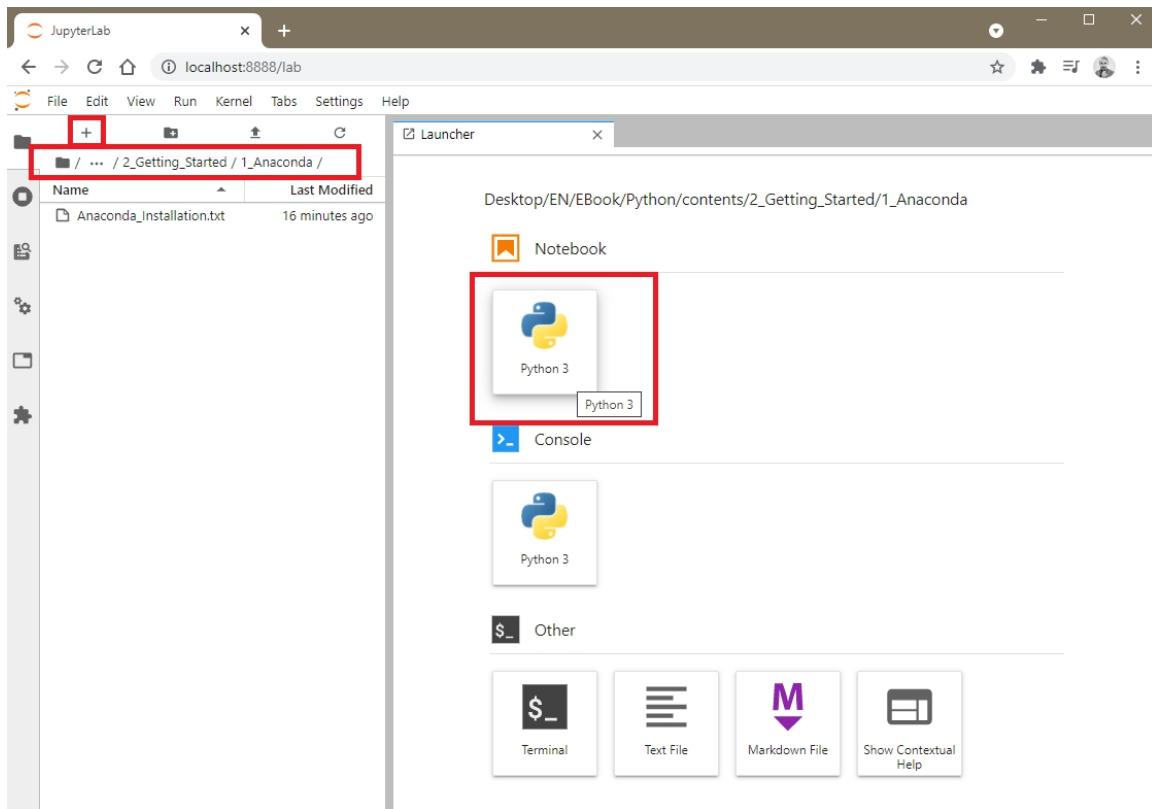


Figure 2-15: Create a new notebook file in the specified location

Right click on the **Untitled.ipynb** file and click on “Rename Notebook...” option. It will open the Rename File popup where you can enter the new name. Type **Anaconda_Install.ipynb** as the new name and click Rename button.

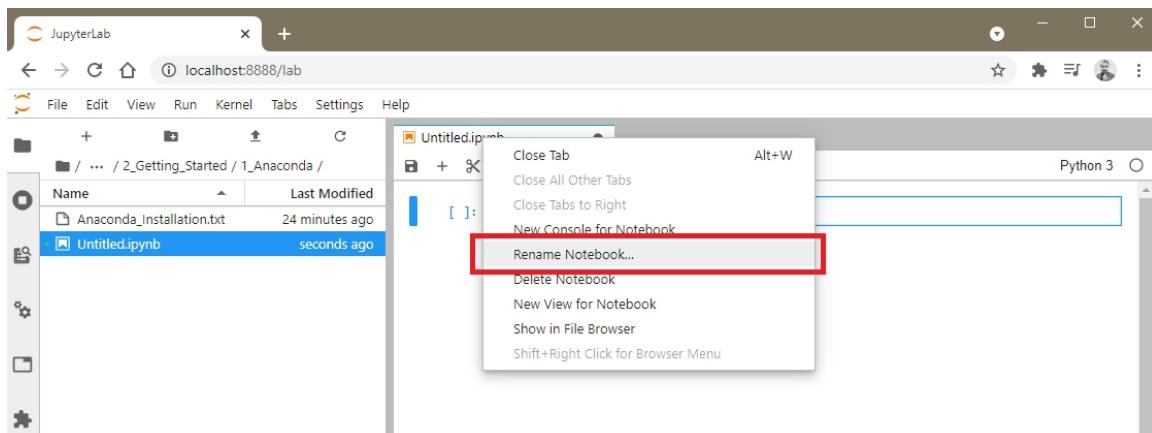


Figure 2-16: Rename notebook file

Now we can start to write our first line of Python to check if everything works correctly. In the first cell of the notebook type `print('Hello World')`. Don't worry, you will learn all about cells and Jupyter Notebook basics in this chapter. Once you type the given statement into the cell, click on Run button, which looks like a Play icon in the toolbar.

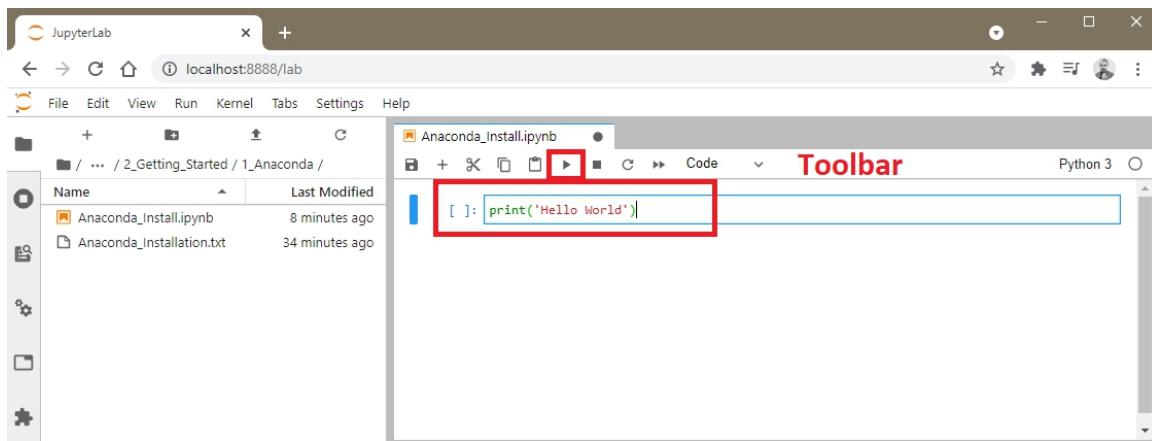


Figure 2-17: Type `print('Hello World')` and click on Run button

JupyterLab is going to run that Python command and it will print “Hello World” as the cell output.

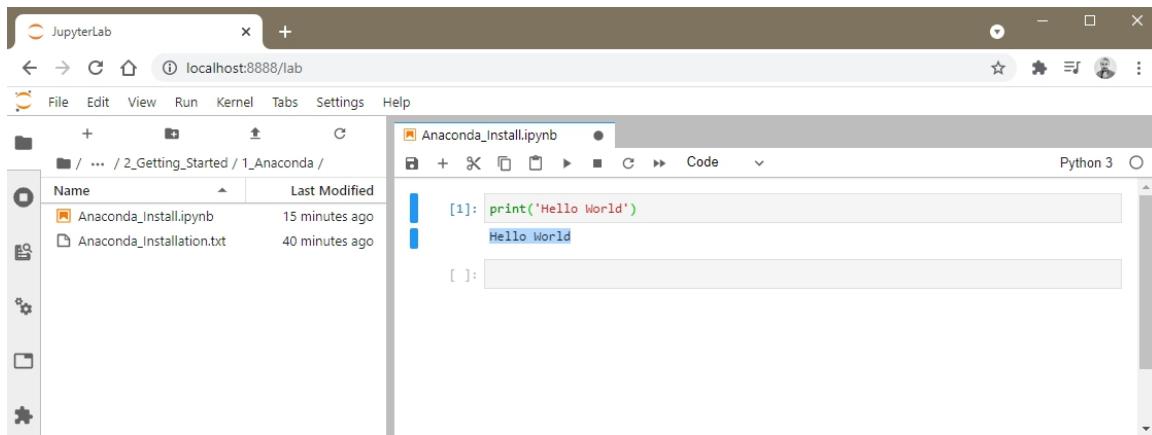
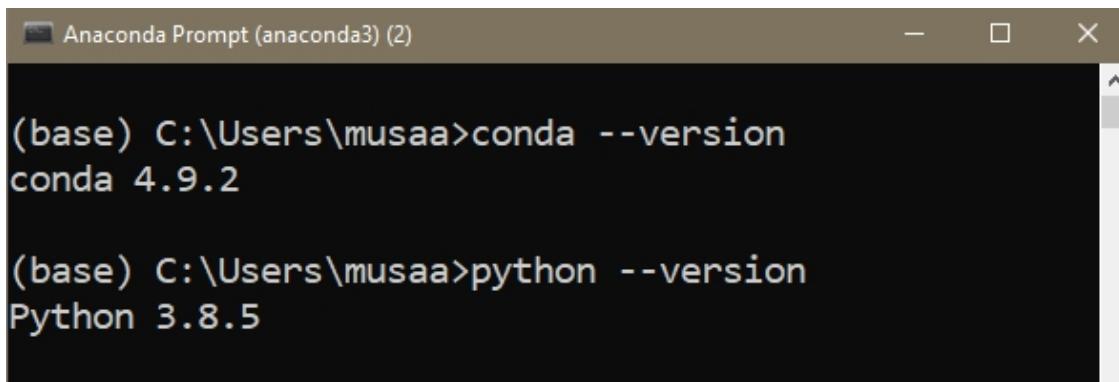


Figure 2-18: First cell output in JupyterLab

If you see the same output without getting any errors, then congratulations, you managed to setup Anaconda and JupyterLab successfully. Your development environment is up and running now.

Before moving to the next section, I want to show another way to check whether Anaconda is installed on your machine or not. This time we will check it from the Anaconda Prompt. From the start menu open Anaconda Prompt, and type in **conda --version**. It should display the Anaconda version installed on your machine. And in a new line this time type **python --version**. It should display the Python version which is installed with Anaconda.

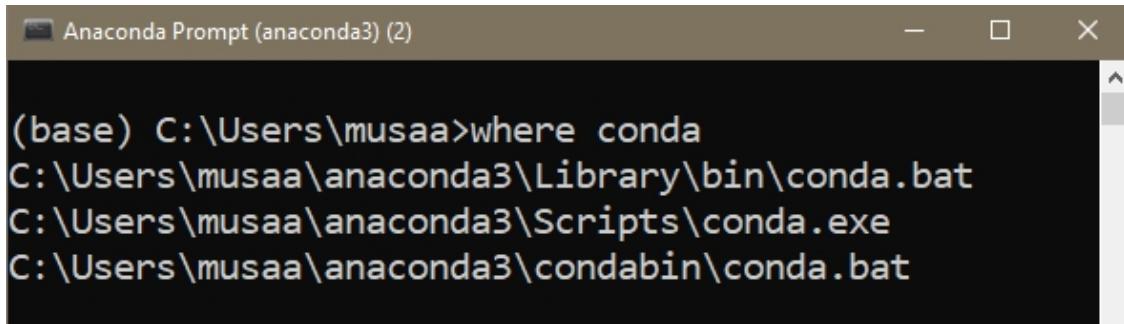


```
Anaconda Prompt (anaconda3) (2)
(base) C:\Users\musaa>conda --version
conda 4.9.2

(base) C:\Users\musaa>python --version
Python 3.8.5
```

Figure 2-19: Anaconda Prompt to check the installation

To see where Anaconda is located on your machine, type **where conda** in the Anaconda Prompt. It will display the location(s) of your Anaconda installation.



The screenshot shows a Windows-style terminal window titled "Anaconda Prompt (anaconda3) (2)". The command "where conda" is entered, and the output shows the paths where conda is located: C:\Users\musaa\anaconda3\Library\bin\conda.bat, C:\Users\musaa\anaconda3\Scripts\conda.exe, and C:\Users\musaa\anaconda3\condabin\conda.bat.

```
(base) C:\Users\musaa>where conda
C:\Users\musaa\anaconda3\Library\bin\conda.bat
C:\Users\musaa\anaconda3\Scripts\conda.exe
C:\Users\musaa\anaconda3\condabin\conda.bat
```

Figure 2-20: To see where Anaconda is installed on your machine

That's all about Anaconda installation. We set everything we need to start and learn Python programming. [Here](#) is the complete documentation for installation if you need more information or to learn how to install on macOS and Linux.^[1]

JupyterLab Password Issue

In recent years, there had been some [security updates](#) for Jupyter Notebooks.^[2] Due to this security updates, there is a possible issue that might occur when you launch JupyterLab for the first time. It may ask for a password or token when you launch JupyterLab. And you will not be able to proceed unless you enter the password.



Password or token:

Invalid credentials

Token authentication is enabled

If no password has been configured, you need to open the notebook server with its login token in the URL, or paste it above. This requirement will be lifted if you [enable a password](#).

The command:

```
jupyter notebook list
```

will show you the URLs of running servers with their tokens, which you can copy and paste into your browser. For example:

```
Currently running servers:  
http://localhost:8888/?token=c8de56fa... :: /Users/you/notebooks
```

or you can paste just the token value into the password field on this page.

See [the documentation on how to enable a password](#) in place of token authentication, if you would like to avoid dealing with random tokens.

Cookies are required for authenticated access to notebooks.

Setup a Password

You can also setup a password by entering your token and a new password on the fields below:

Token

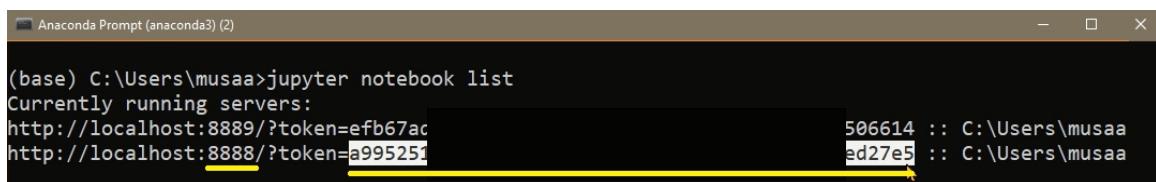
New password

Figure 2-21: JupyterLab asking for password or token

You may or may not see it depending on your system configuration. If you did not see the screen above, then you can skip this section. However if you see this screen when you start JupyterLab here is the solution.

The solution is quite easy. All you have to do is to find the token for the JupyterLab on your machine. To find the tokens and unlock the lock screen on JupyterLab, here are the steps you have to do:

- Open Anaconda Prompt and type this command: **jupyter notebook list**



```
Anaconda Prompt (anaconda3) (2)
(base) C:\Users\musaa>jupyter notebook list
Currently running servers:
http://localhost:8889/?token=efb67ac
http://localhost:8888/?token=a995251
506614 :: C:\Users\musaa
ed27e5 :: C:\Users\musaa
```

Figure 2-22: List the tokens based on ports

- This command will list all the JupyterLab instances with ports and tokens.
- Let's assume the port we are currently using is 8888. So the url for JupyterLab is <http://localhost:8888/lab>. We should copy the token for this port from the command prompt.
- The last step is to go to Password Screen (*Figure 2-21*) of JupyterLab and paste the token into "Password or token" field.

That's it. Now you provided the necessary token to JupyterLab and you can start using it.

Jupyter Notebook Basics

Before we start actual Python coding, we should learn about our development environment. In this section you will learn the basics of JupyterLab and feel comfortable using it.

Let's start with the screen elements. The JupyterLab interface consists of a main work area containing tabs of documents and activities, a collapsible left sidebar, and a menu bar.

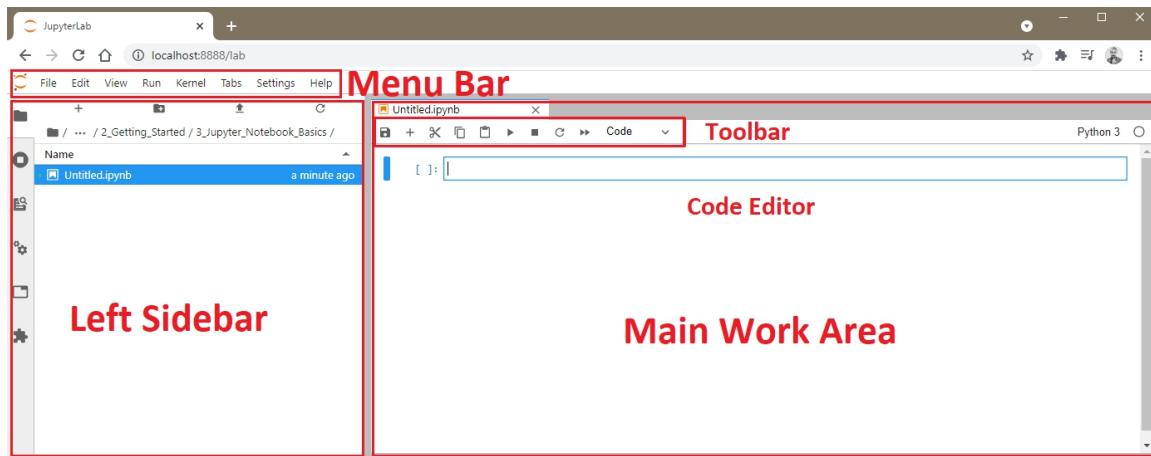


Figure 2-23: *JupyterLab user interface*

Menu Bar

The menu bar at the top of JupyterLab has top-level menus that expose actions available in JupyterLab with their keyboard shortcuts. The default menus are:

- **File:** actions related to files and directories
- **Edit:** actions related to editing documents and other activities
- **View:** actions that alter the appearance of JupyterLab

- **Run:** actions for running code in different activities such as notebooks and code consoles
- **Kernel:** actions for managing kernels, which are separate processes for running code
- **Tabs:** a list of the open documents and activities in the dock panel
- **Settings:** common settings and an advanced settings editor
- **Help:** a list of JupyterLab and kernel help links

Left Sidebar

The left sidebar contains a number of commonly-used tabs, such as a file browser, the list of running kernels and terminals, the command palette, the notebook cell tools inspector, the tab list (list of tabs in the main work area) and the extension manager.

The left sidebar can be collapsed or expanded by selecting “Show Left Sidebar” in the View menu or by clicking on the active sidebar tab.

Main Work Area

The main work area in JupyterLab enables you to arrange documents (notebooks, text files, etc.) and other activities (terminals, code consoles, etc.) into panels of tabs that can be resized or subdivided. When you double click on a file in the file browser, it opens up in the main work area. And here you can edit

the file content. Notebook files in the main work area consist of two parts, the toolbar and the code editor.

Let's start working in JupyterLab. First we will create a folder. To do this, navigate to *2_Getting_Started* folder inside JupyterLab. And inside this folder create a new folder named *3_Jupyter_Notebook_Basics*. Finally create a notebook file inside this folder and name it as **Jupyter_Basics.ipynb**.

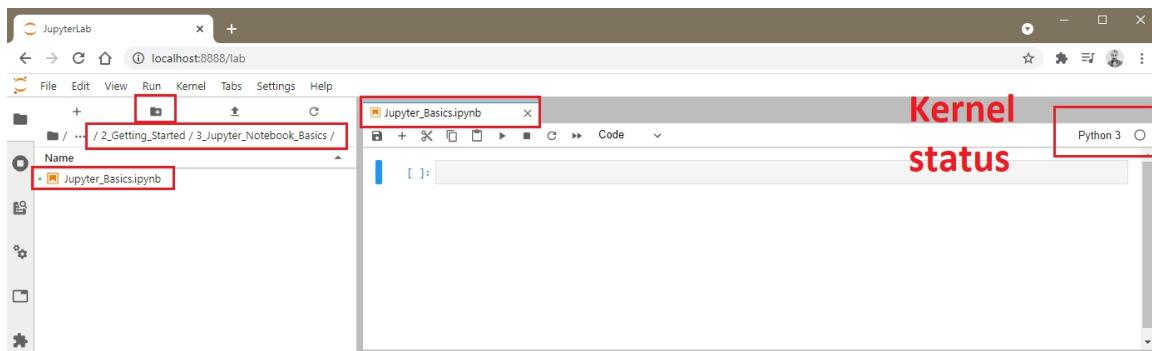


Figure 2-24: Create a folder and a notebook file

When you create a notebook file, behind the scenes, JupyterLab creates a kernel for it. Kernel is nothing but a session that running Python code. On the right hand side of Figure 2-24 you see the kernel status. Here kernel language is Python and it is idle for the time being.

Let's see all the active kernels on our machine. The second button on the left sidebar is the Running panel and it shows a list of all the kernels and terminals currently running across all notebooks, code consoles, and directories.

When you close a notebook document, code console, or terminal, the underlying kernel or terminal running on the server

continues to run. This enables you to perform long-running actions and return later. The Running panel enables you to re-open or focus the document linked to a given kernel or terminal. You can shut down any of them whenever you want.

Now let's create another notebook file. You can leave its name as **Untitled.ipynb**. And when you click on the Running panel you will see all the active kernels on your system.

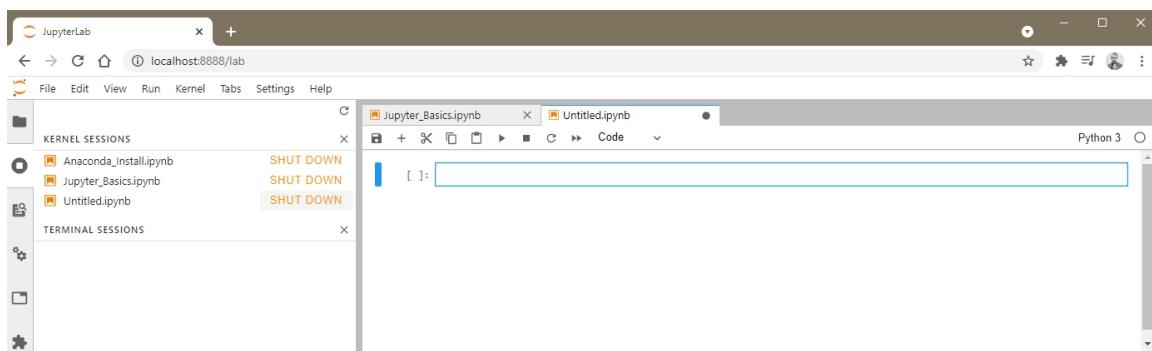


Figure 2-25: Active kernels

You can find the list of all operations related to kernels under the Kernel menu in the top menu bar.

The main work area is the place where you write, edit and run Python code. Two main parts in the work area are the Toolbar and the Code Editor. Toolbar gives a quick way of performing the most-used operations within the notebook, by clicking on an icon.

Here are the buttons in the toolbar and their functionalities:

-  **Save:** Save the notebook contents and create checkpoint
-  **Insert:** Insert a cell below the active cell
-  **Cut:** Cut the selected cells

-  **Copy:** Copy the selected cells
-  **Paste:** Paste the cells from the clipboard
-  **Run:** Run the selected cells and advance
-  **Stop:** Interrupt the kernel (stop running)
-  **Restart:** Restart the kernel
-  **Restart and Run All:** Restart the kernel and re-run the whole notebook
-  **Code:** **Cell Types:** Change the cell type for the active cells

Now we can move on to the code editor. Jupyter notebook is based on cells. Cells are the runnable individual code blocks. There are three main cell types in Jupyter notebooks: Code, Markdown and Raw.

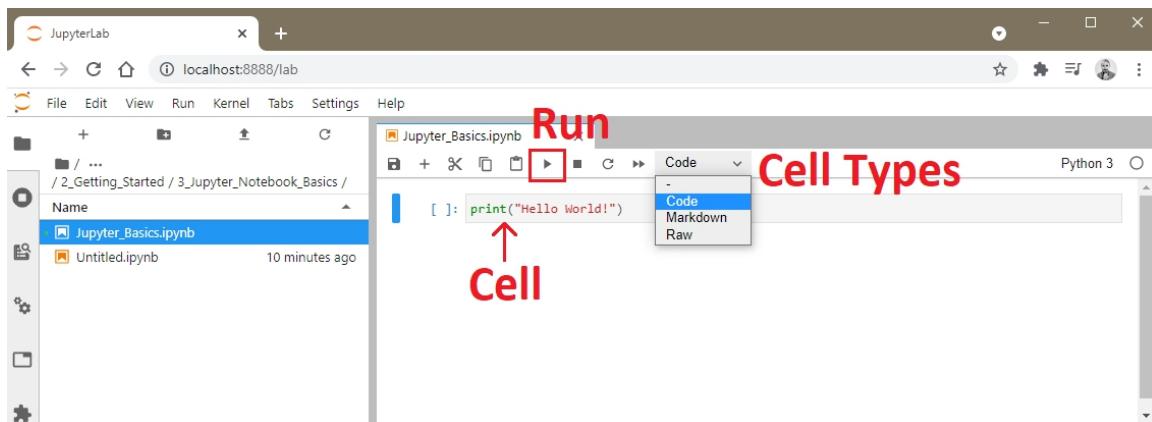


Figure 2-26: Cells and cell types

To run the cell or a block cells, you first select what you want to run and then click on the Run button in the toolbar. When you run a cell, you will see the cell output and a number in the square

brackets next to cell. That number is the running order of that particular cell in the current active kernel.

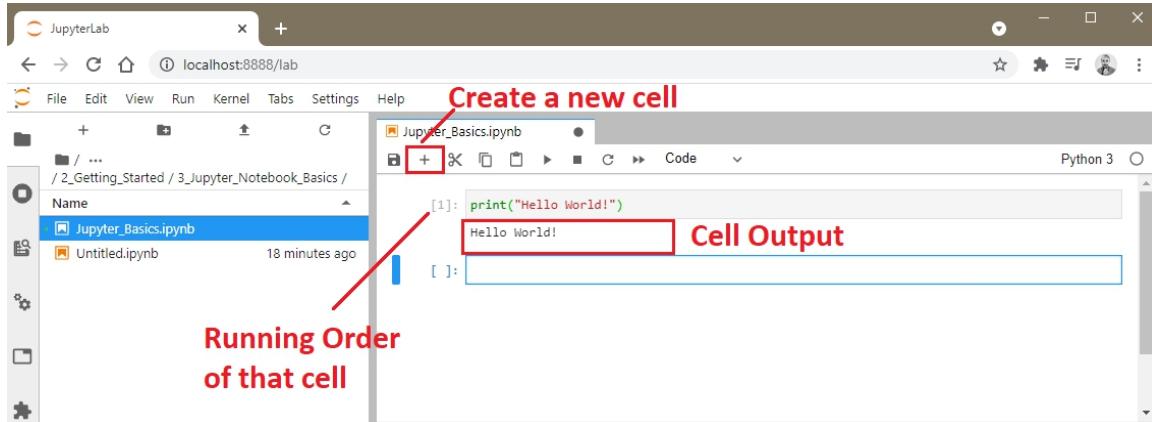


Figure 2-27: Cell output and the order number

To create a new cell we click on the Plus icon in the toolbar. And change the cell type as Markdown. Type **this is a ***markdown*** cell** in it and run the cell. You will see that the cell itself will become a plain text area. Markdown cell displays text which can be formatted using markdown language. You can also type HTML in the markdown cells. JupyterLab will parse that HTML code and format it. Here ****** are used to make the text bold.

Now create another cell and change its type as Raw. In this new cell, type **this is a Raw cell** text. Raw cells are plain text cells that do not allow any formatting.

If you run all the cells you will see the outputs as follows.

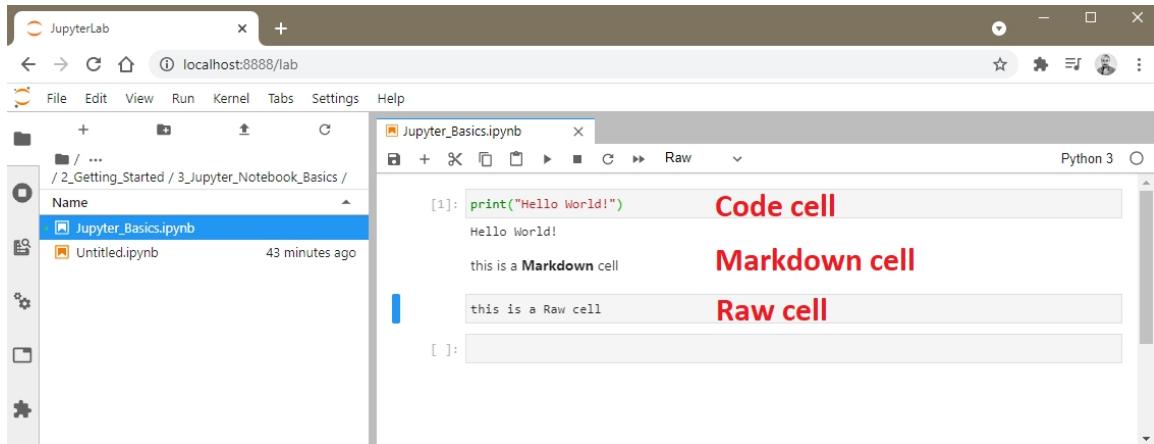


Figure 2-28: Cell types and outputs

That's all for JupyterLab basics. Now we can move on to Virtual Environments in Python. You can find the documentation about Jupyter Interface on the official [Jupyter Documents](#) web site. [3]

Python Virtual Environments

What is a virtual environment? Virtual Environment is an isolated runtime environment that allows Python users and applications to install and upgrade Python packages without interfering with other Python applications running on the same system.

Python applications will often use packages and modules that don't come as part of the standard library. Applications will sometimes need a specific version of a library, because the application may require that a particular bug has been fixed or the

application may be written using an obsolete version of the library's interface.

This means it may not be possible for one Python installation to meet the requirements of every application. If application A needs version 1.0 of a particular module but application B needs version 2.0, then the requirements are in conflict and installing either version 1.0 or 2.0 will leave one application unable to run.

The solution for this problem is to create a virtual environment, a self-contained directory tree that contains a Python installation for a particular version of Python, plus a number of additional packages.

Different applications can then use different virtual environments. To resolve the earlier example of conflicting requirements, application A can have its own virtual environment with version 1.0 installed while application B has another virtual environment with version 2.0. If application B requires a library be upgraded to version 3.0, this will not affect application A's environment.^[4]

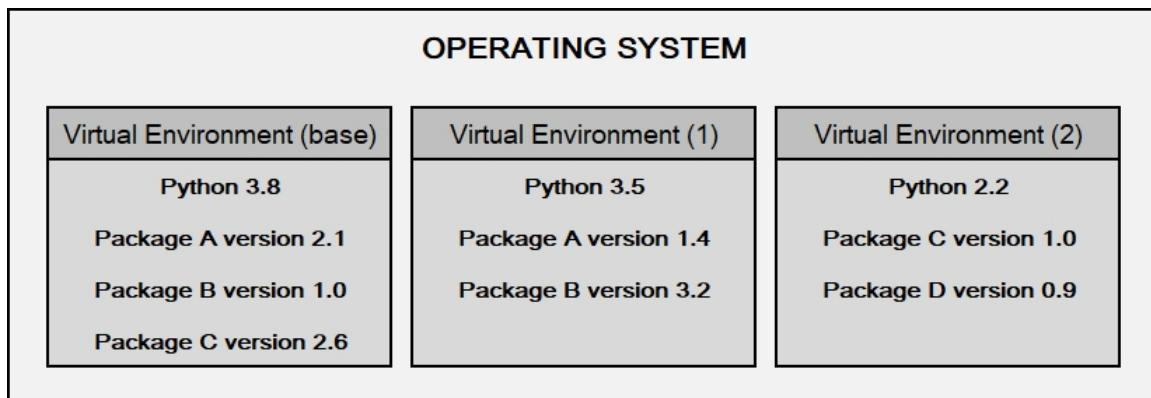


Figure 2-29: Different Virtual Environments on the same OS, completely isolated from each other

Now that we know what the virtual environments are, let's see them in Anaconda Navigator.

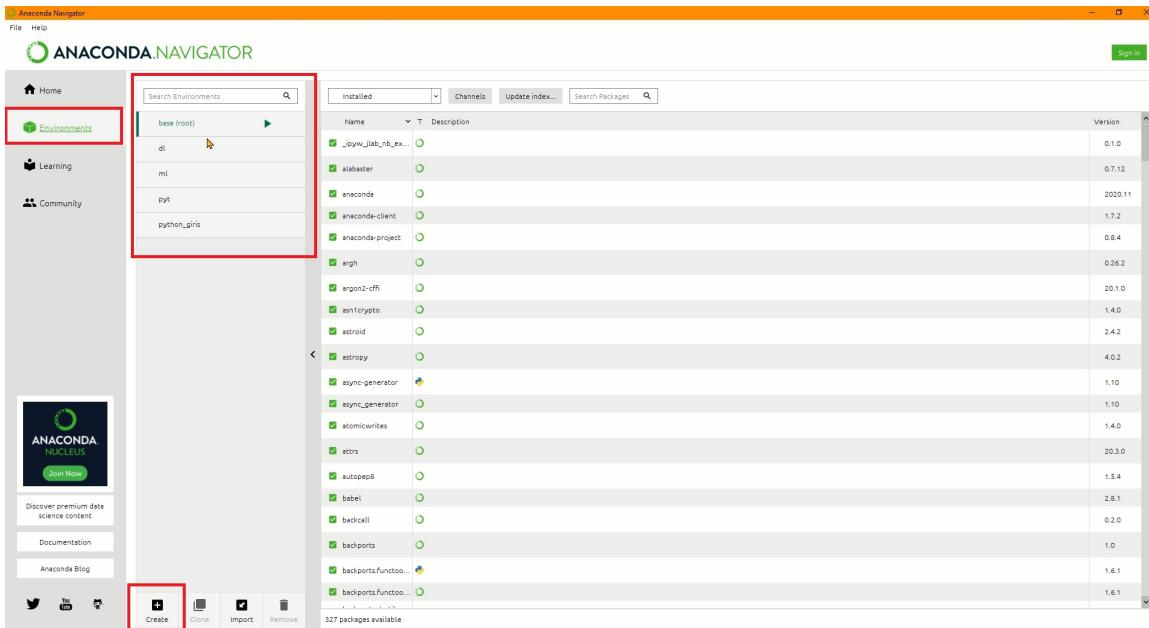


Figure 2-30: Virtual Environments List in Anaconda Navigator

When you install Anaconda, it will create the default virtual environment (venv for short) on your machine. This default environment is called **base** environment. And in *Figure 2-30* you see the base environment as the active one. You also see other environments which are created on my computer. You can create, delete or edit virtual environments on the same machine.

Now let's create a new venv inside the Navigator. On the bottom of the venv list, you see button named Create. Click on that and you will a popup screen to name your venv and select the

language and version. Name the venv as **python_intro** (no spaces) and select Python version 3.8 (or whatever is current on your machine). It will create the new venv when you click on Create button. (We will use **python_intro** as our virtual environment in this book.)

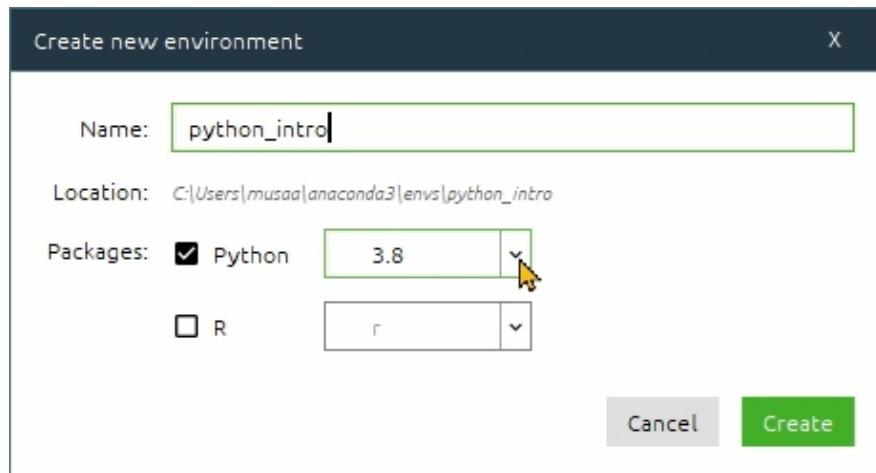


Figure 2-31: Create a new virtual environment

Now you can see your new virtual environment in the venv list. It has also been activated after creation. We will talk about activating a venv in this section. You will also see some packages been already installed in your venv. These are mandatory packages for Python to run. And the package named **pip** is the most important one. In standalone Python installation, we use **pip** to install other Python packages. Since we are using Anaconda, we will be using **conda** to install packages, instead of pip.

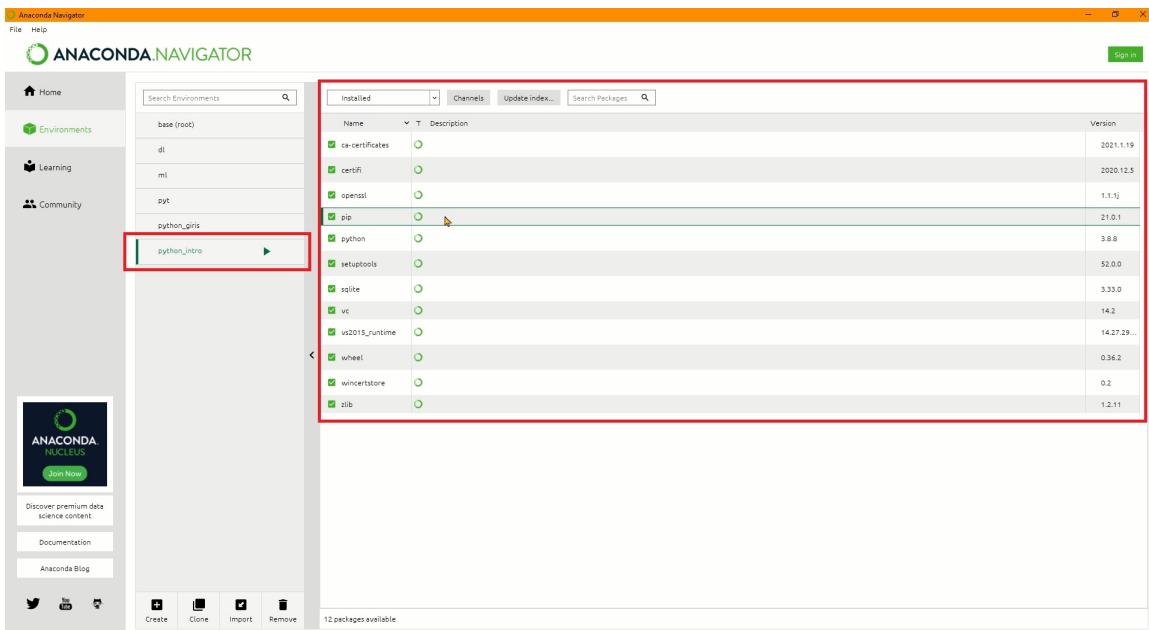


Figure 2-32: The new venv and important packages in it

Let's talk briefly on packages in Python. Python programming language is modular in nature, which means it's made up of packages and modules. A **Module** is the basic unit of code reusability in Python. Actually, it is nothing but a file including Python code in it. A **Package** is a Python module which can contain other modules or packages.^[5] You can think packages as extended modules. Python packages are kept mainly in central repositories like <https://pypi.org/> or Anaconda Package Manager, which we use in this book.

Python packages live in virtual environments. So, to install a package you need to declare a virtual environment. In other words, you have to **activate** the virtual environment that you want to add packages into it. To activate a venv in Navigator, just click on its name in the Environment list. Once it's activated, you can install

packages in it. You will see a green Play icon on the right hand side of the venv name.

If you now activate your newly created venv, **python_intro**, you may see that the packages differ from the base environment (depending on your configuration). You may also need to install JupyterLab again, because the last time we installed it in the **base** environment.

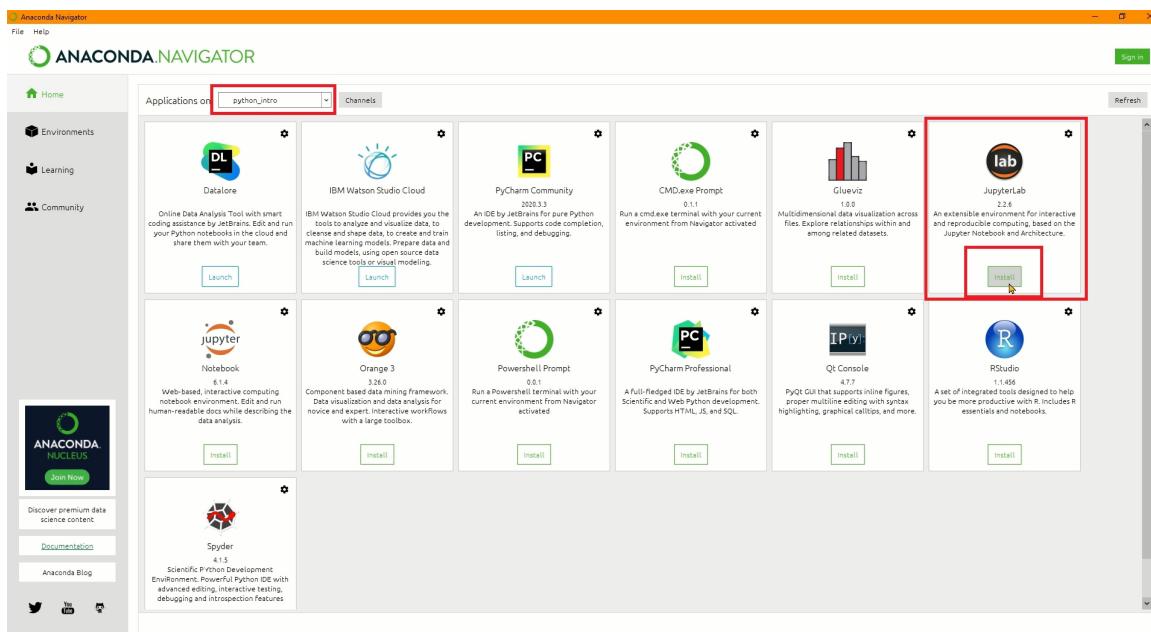


Figure 2-33: We need to install JupyterLab again in the new virtual environment

After JupyterLab installation is completed in the new venv, you can launch it and use it as we did before. All the rest is the same.

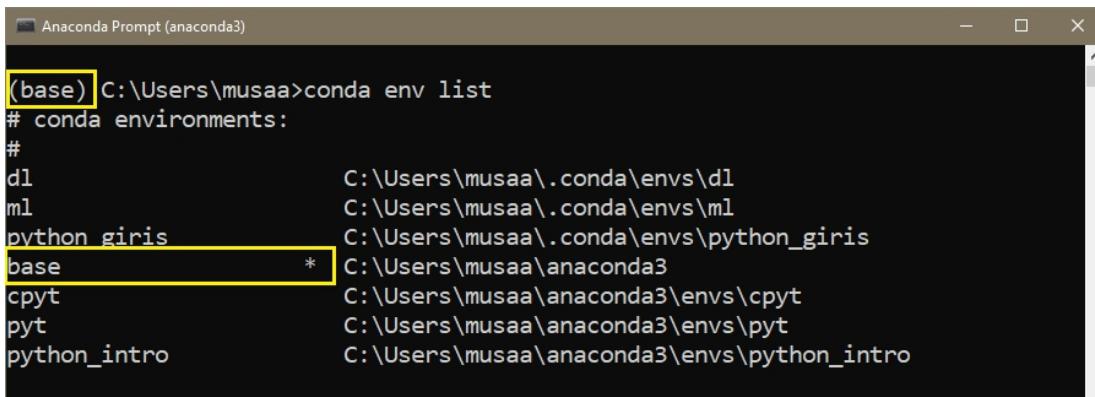
Open JupyterLab from Anaconda Prompt

So far we used Anaconda Navigator to create new virtual environments, to activate them or to launch JupyterLab. Anaconda Navigator is great but it has lots of applications bundled in it, which makes it quite heavy to run, in terms of performance. That's why we don't want to start Anaconda Navigator whenever we need JupyterLab. We have a better option, which is the Anaconda Prompt.

From now on, in this book we will be using Anaconda Prompt to manage virtual environments and launch JupyterLab. Let's see how it's easy to use Anaconda Prompt step by step:

Step 1: Open Anaconda Prompt in your computer. It will start with the **base** environment as active.

Step 2: See the list of virtual environments on your machine:
conda env list



```
(base) C:\Users\musaa>conda env list
# conda environments:
#
dl                         C:\Users\musaa\.conda\envs\dl
m1                         C:\Users\musaa\.conda\envs\m1
python_giris                C:\Users\musaa\.conda\envs\python_giris
base                        * C:\Users\musaa\anaconda3
cpyt                        C:\Users\musaa\anaconda3\envs\cpyt
pyt                         C:\Users\musaa\anaconda3\envs\pyt
python_intro                C:\Users\musaa\anaconda3\envs\python_intro
```

Figure 2-34: List of Virtual Environments on our machine with the base as the active one.

Step 3 (Optional): If you want to create a new virtual environment: **conda create --name <my_virtual_env>**

Step 4: Activate the virtual environment you want to work on:

```
conda activate python_intro
```

Step 5: Install JupyterLab, if it's not already installed: **conda install jupyterlab**

Or you can also use: **conda install -c conda-forge jupyterlab**

Step 6: Now you can [start JupyterLab](#): **jupyter lab**

That's it, if type **jupyter lab** in Anaconda Prompt and hit enter, JupyterLab will open in a new browser tab.

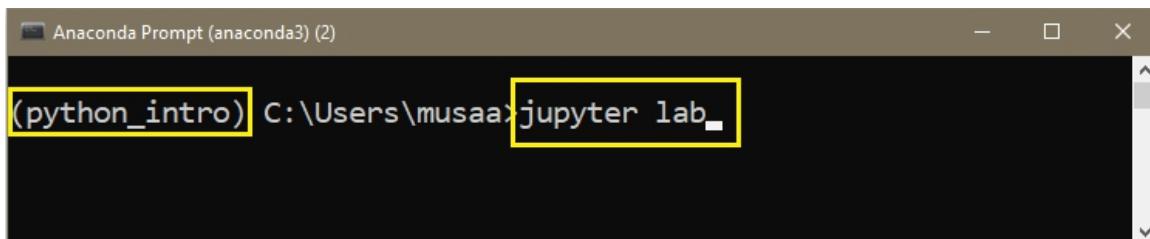
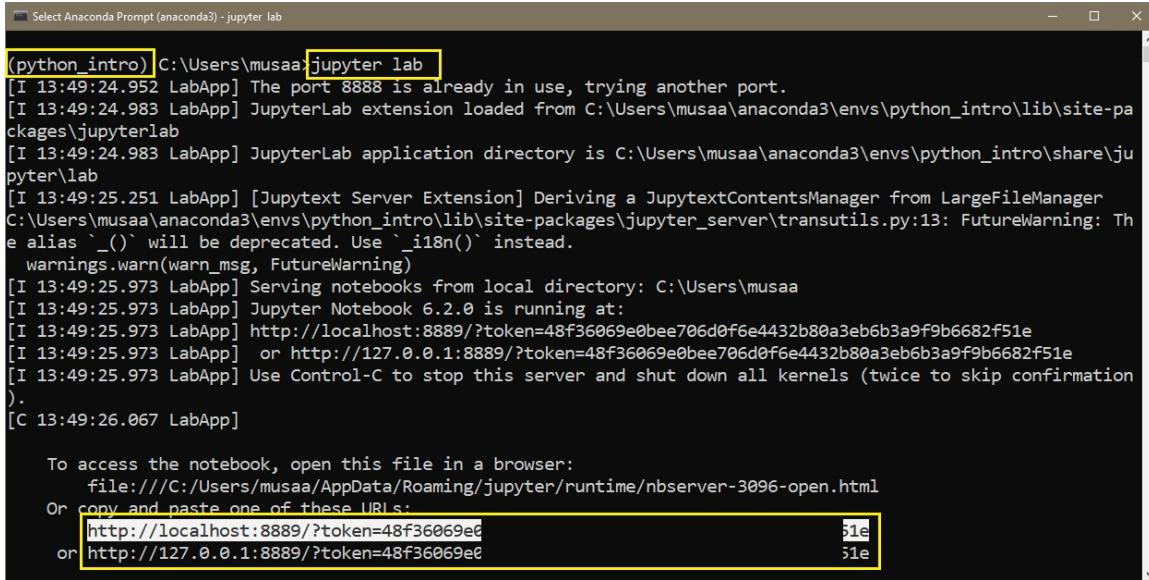


Figure 2-35: Open JupyterLab with Anaconda Prompt (make sure python_intro venv is activated)

In general, JupyterLab will open in your default web browser. But in some cases that might not happen. If it doesn't start automatically then you can type the url in your browser. For example, <http://localhost:8889> or the one with token <http://localhost:8889/?token=48f360.....>



```
(python_intro) C:\Users\musaa>jupyter lab
[I 13:49:24.952 LabApp] The port 8888 is already in use, trying another port.
[I 13:49:24.983 LabApp] JupyterLab extension loaded from C:\Users\musaa\anaconda3\envs\python_intro\lib\site-packages\jupyterlab
[I 13:49:24.983 LabApp] JupyterLab application directory is C:\Users\musaa\anaconda3\envs\python_intro\share\jupyter\lab
[I 13:49:25.251 LabApp] [JupyterText Server Extension] Deriving a JupyterTextContentsManager from LargeFileManager
C:\Users\musaa\anaconda3\envs\python_intro\lib\site-packages\jupyter_server\transutils.py:13: FutureWarning: Th
e alias `__()` will be deprecated. Use `__i18n__()` instead.
  warnings.warn(warn_msg, FutureWarning)
[I 13:49:25.973 LabApp] Serving notebooks from local directory: C:\Users\musaa
[I 13:49:25.973 LabApp] Jupyter Notebook 6.2.0 is running at:
[I 13:49:25.973 LabApp] http://localhost:8889/?token=48f36069e0bee706d0f6e4432b80a3eb6b3a9f9b6682f51e
[I 13:49:25.973 LabApp] or http://127.0.0.1:8889/?token=48f36069e0bee706d0f6e4432b80a3eb6b3a9f9b6682f51e
[I 13:49:25.973 LabApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation
).
[C 13:49:26.067 LabApp]

  To access the notebook, open this file in a browser:
    file:///C:/Users/musaa/AppData/Roaming/jupyter/runtime/nbserver-3096-open.html
  Or copy and paste one of these URLs:
    http://localhost:8889/?token=48f36069e0bee706d0f6e4432b80a3eb6b3a9f9b6682f51e
    or http://127.0.0.1:8889/?token=48f36069e0bee706d0f6e4432b80a3eb6b3a9f9b6682f51e
```

Figure 2-36: JupyterLab url and token after it launches

Now that we have everything we need, we can start actual Python coding. In the next chapter we will write our first Python program and start learning Python basics.

OceanofPDF.com

3. The First Program

Starter Notebook File

At the beginning of each chapter we will start with a Starter Notebook File. This file includes only the chapter title. And you are expected to write your code in this file while learning the concepts.

You can either download it from [Github Repository](#) or you can create it manually. If you choose to download it you can skip this section go to the next one, Hello World. If you want to create it from scratch, here is how you can do this.

As you may remember from the previous chapter, in Python everything starts with virtual environments (venv). That's why we have to activate the related venv (python_intro). To do this, let's first see the venv list on our machine. Open Anaconda Prompt, type **conda env list** and hit enter. If you see **python_intro** in the list it means you have successfully created that venv. If not, then you have to repeat Python Virtual Environments section in Chapter 2.

Let's move on and activate the python_intro virtual environment. Type **conda activate python_intro** into the Anaconda Prompt and hit enter. Now this venv should be active. You should see its name in the parentheses at the beginning of the command line.

Now, as the final step let's launch JupyterLab. In the Anaconda Prompt, type **jupyter lab** and hit enter. The JupyterLab should open up in a new tab in your default web browser.

Inside JupyterLab, navigate to folder *3_The_First_Program*, create a new notebook file and rename it as **First_Program.ipynb**. This will be the file we will be using in this chapter. Select the first cell and change its type as Markdown. We will write the chapter title in this cell. Type the text **# First Program** in it. The **#** symbol means **<h1>** tag in HTML. Which means it will be large enough to serve as the title cell. If you run the cell, you will see the formatted chapter title.

Here is the starter notebook file for this chapter:

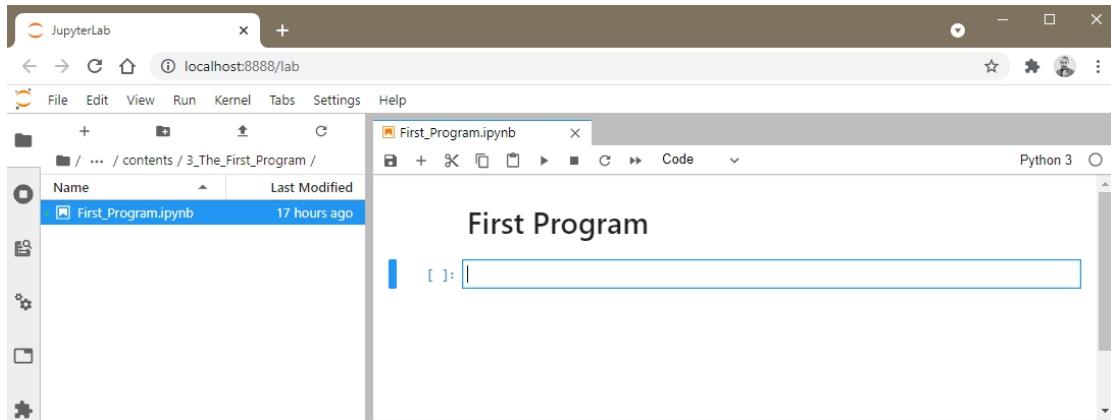


Figure 3-1: *First_Program.ipynb* file in JupyterLab

Hello World!

Finally we are ready to write our first program. A **program** is a procedure that tells the computer how to succeed a given task. Probably the one below will be the easiest Python program that you

can write. It will just print **Hello World!** on the screen. And here it is:

```
[  
1  1 print("Hello World!")  
]:
```

```
[  
1  Hello World!  
]:
```

Here, the statement is `print("Hello World!")` which uses Python's built-in function `print()`. You will learn more about functions in later chapters. It is used to print some text as the output, which is **Hello World!** here.

Let's print another text. This time we will print as **This is second the line of code**.

```
[  
2  1 print("This is the second line of code")  
]:
```

```
[  
2  This is second the line of code  
]:
```

Now let's print a numeric value. In cells 1 and 2, we print plain text but this time we will print an integer.

```
[  
3  1 print(2345)  
]:
```

```
[  
3  2345  
]:
```

Let's just do an ordinary operation. We will see arithmetic operations in the next section but here let's see an example. Let's type `5 + 2` in the next cell and see the result by running it.

```
[  
4  1  5 + 2  
]:
```

```
[  
4  7  
]:
```

Arithmetic Operations

Arithmetic Operations in Python are as follows:

- Addition: `+`
- Subtraction: `-`
- Multiplication: `*`
- Division: `/`
- Exponential: `**`
- Modulo Operator: `%`
- Floor Division: `//`

Let's see them in action.

```
[  
5  1  # Addition  
]:  
2  24 + 16
```

```
[  
5  40  
]:
```

```
]: [1 # Subtraction
]: [2 47 - 17

[6 30
]: [7 1 # Multiplication
]: [2 5 * 8

[7 40
]: [8 1 # Division
]: [2 80 / 10

[8 8.0
]: [9 1 # Division
]: [2 9 / 2

[9 4.5
]:
```

Exponential means the power operator. `3**2` means “3 to the power of 2” which is 9. Here the base is 3 and the exponent is 2.

```
[1  
0]: 1 # Exponential  
2 3**2
```

```
[1  
0]: 9
```

```
[1  
1]: 1 # Exponential  
2 4**3
```

```
[1  
1]: 64
```

The Modulo Operator (`%`) returns the remainder of dividing the left hand operand by right hand operand. It's used to get the remainder of a division. For example when you divide 64 by 5, the remainder will be 4.

```
[1  
2]: 1 # Modulo Operator  
2 64 % 5
```

```
[1  
2]: 4
```

```
[1  
3]: 1 # Modulo Operator  
2 9 % 2
```

```
[1  
3]: 1
```

The `//` is called the **Floor Division** operator in Python. To understand the floor division, you first need to understand the floor of a real number. The floor of a real number is the largest integer that is less than or equal to that number. For example floor of **4.5** is **4** and floor of **2.4** is **2**. Be careful about the floor of negative numbers. Floor of **-4.5** is **-5** and floor of **-2.4** is **-3**.

```
[1  
4]: 1 # Floor Division  
     2 9 // 2
```

```
[1  
4]: 4
```

```
[1  
5]: 1 # Floor Division  
     2 12 // 5
```

```
[1  
5]: 2
```

```
[1  
6]: 1 # Floor Division  
     2 -9 // 2
```

```
[1  
6]: -5
```

```
[1  
7]: 1 # Floor Division  
     2 -12 // 5
```

```
[1  
7]: -3
```

Values and Types

In your first steps of programming there are two basic but important concepts you should learn. Values and types. **Values** are anything in the program which holds meaning. Here the term meaning simply means data. So values are the data in your program. **Types** are the format of that values in computer's memory. In other words, type is shape of that data.

For example the text “**Hello World!**” is just a value. If you type “**Hello World!**” in a cell in JupyterLab and run it, you will see the output as the same text. JupyterLab just prints the value, that's all.

```
[1] 8]: 1 "Hello World!"
```

```
[1] 8]: Hello World!"
```

Now if you write `type("Hello World!")` in the cell and run it, now it will give you the type of the text “**Hello World!**”. And since it is a plain text its type `str`. `str` means **String** and we will see strings in detail later on.

```
[1] 9]: 1 type("Hello World!")
```

```
[1] 9]: str
```

`type()` is the second function we learned so far. It's another built-in function in Python and it returns the type of the parameter you pass in the parentheses.

Now let's pass another parameter to it. Let's pass number **2** this time. And since it is an integer, Python will give you **int** as its type.

```
[2  
0]: 1 type(2)
```

```
[2  
0]: int
```

Let's pass **4.6** now, a floating point number. As you may guess Python will give you **float** as its type. Don't worry we will see all of these types and more in detail.

```
[2  
1]: 1 type(4.6)
```

```
[2  
1]: float
```

You have completed your first program now. You have some basic understanding about types and values and how to print those using Python functions. Now it's time to test what you have learned so far. First you will have True/False questions and then a quiz of 10 questions.

True / False - First Program

For each questions below, decide whether it's true or false.

Q1:

print(Hello World) statement is going to print "Hello World" on the screen.

- True
- False

Q2:

The output of **type(12)** statement, is different from the output of **type("12")** statement.

- True
- False

SOLUTIONS - True / False - First Program

Here are the solutions for True / False questions.

S1:

print(Hello World) statement is going to print "Hello World" on the screen.

- True
- False

It's **false** because the statement inside the **print()** function should be in quotes like **print("Hello World")**.

S2:

The output of **type(12)** statement, is different from the output of **type("12")** statement.

- True
- False

It's **true** because **type(12)** will print **int** but **type("12")** will print **str**.

QUIZ - The First Program

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_First_Program.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *3_The_First_Program*.

At the end of each chapter you will have a quiz. There will be 10 questions in each quiz. Below each question, you will find the expected answer for it. And you will try to find that answer with Python. Quizzes are distributed as Jupyter Notebook files. After you download the quiz file, you should put it in a folder where you can access it inside the JupyterLab. You will also find the quiz questions in this section. But remember, you cannot learn a programming language just by reading it. You should code and try it yourself.

In the next section you will find the quiz solutions. But before reading the solutions, you are expected to solve the questions on your own. Every question in the quiz aims to teach you a specific topic, so make sure you try to solve each one of them. After you try and even after you solve them, I strongly recommend to check the solutions provided here. It is very important because the solutions in this book aim to teach you different ways to approach Python problems. Here are the questions for this chapter:

QUIZ - First Program:

Q1:

How many seconds are there in 24 minutes and 36 seconds?

Answer: 1476

Q2:

The Batmobile, having average speed of 80 km/hour will travel from Gotham to Arkham. We know the distance between Gotham and Arkham, which is 450 km's. How many minutes will it take for Batmobile to travel from Gotham to Arkham?

Answer: 337.5

Q3:

What is 14th power of 2?

Answer: 16384

Q4:

Can you print "Python, I love you :)" on the screen?

Answer: Python, I love you :)

Q5:

Can you find the type of number 15?

Answer: <class 'int'>

Q6:

Can you print the type of text "15"?

Answer: <class 'str'>

Q7:

What is the remainder when we divide 49 by 5?

Answer: 4

Q8:

What is the floor division (integer division) result of dividing 49 by 5?

Answer: 9

Q9:

What is the average of sum of squares of integers from 1 to 5?

(average of this sum -> $1^2 + 2^2 + 3^2 + 4^2 + 5^2$)

Answer: 11.0

Q10:

Solve this equation for x with Python:

$$x^2 = 4^3 + 17$$

What is the value of x?

Answer: 9.0

OceanofPDF.com

SOLUTIONS - The First Program

Here are the solutions for the quiz for Chapter 3 - The First Program.

SOLUTIONS - First Program:

S1:

Since there are 60 seconds in one minute, there will be $24 * 60$ seconds in 24 minutes. Then we add 36 seconds. And the total is $24 * 60 + 36$ seconds.

```
[  
1  1  24 * 60 + 36  
]:
```

```
[  
1  1476  
]:
```

S2:

The number of hours that the Batmobile will get to Arkham is distance / average speed, which is $450 / 80$ hours. And if we convert this result into minutes it will be $(450 / 80) * 60$.

```
[  
2  1  (450 / 80) * 60  
]:
```

```
[  
2  337.5  
]:
```

S3:

The result of 2 to the power of 14 is 2^{14} which is denoted as `2**14` in Python syntax.

```
[ 1  2**14  
3
```

```
]: [3]: 16384
```

S4:

In Python, to print text we use **print()** built-in function. Inside the parentheses of the function, we pass the text we want to print. This is called the parameter and here our parameter is the text of "Python, I love you :)". So the syntax is **print("Python, I love you :)")**.

```
[4]: 1 print("Python, I love you :)")
]: [4]: Python, I love you :)
```

S5:

To get the type of any object in Python we use, **type()** built-in function. The parameter will be the object which want to get the type. Here it is an integer value of 15. So the syntax is **type(15)**.

```
[5]: 1 type(15)
]: [5]: int
```

S6:

In this question we will print the type of the text “15”. Be careful, it is not an integer now. It is a text, because it is surrounded with quotation marks. Actually it is called a **String (str)**, and you will learn Strings in very detail later

on. Here I just wanted to show you the difference between, 15 and “15” for Python. The code to get its type is `print(type("15"))`.

```
[  
6  1  print(type("15"))  
]:
```

```
[  
6      <class 'str'>  
]:
```

S7:

To get the remainder we use Modulo Operator (%). It is used just like ordinary math operators. So the result of 49 % 5 is 4.

```
[  
7  1  49 % 5  
]:
```

```
[  
7      4  
]:
```

S8:

The floor division (integer division) is a special kind of division operation. It gives you the floor of the division result (dividend). The floor of a real number is the largest integer that is less than or equal to that number. In this question we need the result of `49 // 5`. If it was a normal division then the result would be 9.8. But the operation here is the floor division, so we will take the largest integer that is less than or equal to 9.8, which is 9.

```
[  
8  1  49 // 5  
]:
```

```
[  
8      9  
]:
```

]:

S9:

In this question, we need the average of this summation: $1^2+2^2+3^2+4^2+5^2$. Since there are 5 terms here we will divide this sum into 5. So the solution is $(1^{**2} + 2^{**2} + 3^{**2} + 4^{**2} + 5^{**2}) / 5$ which is 11.0.

[

9

1 $(1^{**2} + 2^{**2} + 3^{**2} + 4^{**2} + 5^{**2}) / 5$

]:

[

9

11.0

]:

S10:

Throughout this book, we will have some mathematical operations. But the aim is to get you hands-on with problem solving in Python, not to learn mathematics. In this question we have a math equation to solve for x. To find x, we need to get the square root of the right hand side, $4^{**3} + 17$. And the square root is nothing but the power of 1/2. So the solution is $(4^{**3} + 17)^{**}(1/2)$.

[1

0]:

1 $(4^{**3} + 17)^{**}(1/2)$

[1

0]:

9.0

4. Variables

What is a Variable?

Inside JupyterLab, navigate to folder *4_Variables*. Then create a new notebook file and rename it as **Variables.ipynb**. This will be the starter file that we will be using in this chapter. You can either download it from [Github Repository](#) or you can create it manually. If you want to create it from scratch, revisit the section Starter Notebook File in Chapter 3.

Variables are one the main building blocks of programming. A **Variable** is a name referring to a value. It is the identifier of a specific part of the computer memory which holds that value in it. In other words, it is the **name of the address** for that **value** in the memory.

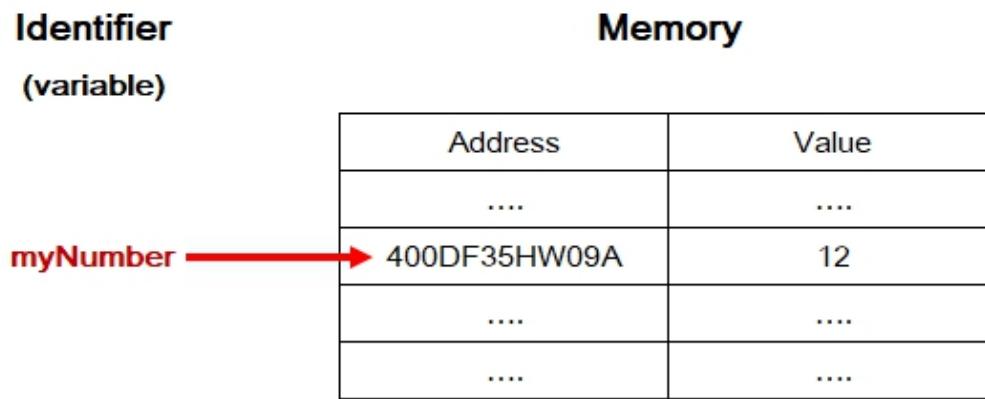


Figure 4-1: Variables and values in the computer memory

Assigning:

How do we link a variable to a value? We do this with **assignment** operator. **Assignment** creates (defines) a variable and binds it to the given value. Assignment is denoted by the equal sign `=`.

Let's see an example:

```
[  
1  myNumber = 12  
]:
```

Here we create a **variable**, give it the name **myNumber**, and assign the value **12** to it. Now let's print its value.

```
[  
2  print(myNumber)  
]:  
  
[  
2  12  
]:
```

As you see here **myNumber** is the variable and **12** is its value. When you write `print(myNumber)`, Python gives you the **value** which exists in the address named as **myNumber** in the memory.

Let's define another variable. This time let's give some text as its value.

```
[  
3  info = "Python name comes from Monty Python's  
  Flying Circus"  
]:
```

In cell 3, we define a variable named **info** and assign the text **“Python name comes from Monty Python's Flying Circus”**

to it. Now if we print its value you will see the text you assigned while defining it.

```
[  
4  1  print(info)  
]:
```

```
[  
4  Python name comes from Monty Python's Flying  
]:  Circus
```

Let's define another variable. This time let's make its value as a floating point number. And we will print its value right after we define it.

```
[  
5  1  pi = 3.1415  
]:  
2  print(pi)
```

```
[  
5  3.1415  
]:
```

Now let's define a variable which has an integer value. And print its value.

```
[  
6  1  n = 460  
]:  
2  print(n)
```

```
[  
6  460  
]:
```

Let's say you want to print a variable which you haven't define yet. We know that we have no variable named `a`. Let's see what happens if we try to print its value.

```
[  
7 1 print(a)  
]:  
[  
7  
]:  
-----  
-----  
NameError Traceback (most  
recent call last)  
<ipython-input-7-bca0e2660b9f> in <module>  
----> 1 print(a)  
NameError: name 'a' is not defined
```

As you see in the output of cell 7, we get an error. This is how the programming errors look like in JupyterLab. The error type is **NameError** and it says **name 'a' is not defined**. Which is true because we didn't define a variable with name `a` yet.

Let's define it, assign **1000** to it, and then print its value. In JupyterLab to print a variable, you can just write the variable name, provided that it's in the last line. In other words, if you are at the last line in a cell, and you want to print a variable, you just write its name. You don't have to explicitly type `print()` statement. JupyterLab prints the value in the last line, implicitly.

```
[  
8 1 a = 1000  
]:
```

```
2  a  
[  
8  1000  
]:
```

Reassigning:

In Python, you can **reassign** a variable. Reassigning means that you can connect a different value with a previously assigned variable. In other words, you can change the value which lives in that memory address. The address which your variable is linked to.

Remember our first variable was **myNumber** and we assigned to value **12** to it. Now let's reassign it. We want its new value to be **“Python Hands-On”**.

```
[  
9  1  myNumber = "Python Hands-On"  
]:  
2  myNumber  
[  
9  Python Hands-On'  
]:
```

That's it. We reassigned a variable in cell 9. It's that easy and simple. No matter the previous value or the new value, you are allowed to reassign variables in Python. Let's see what happens in the memory when you reassign a variable.

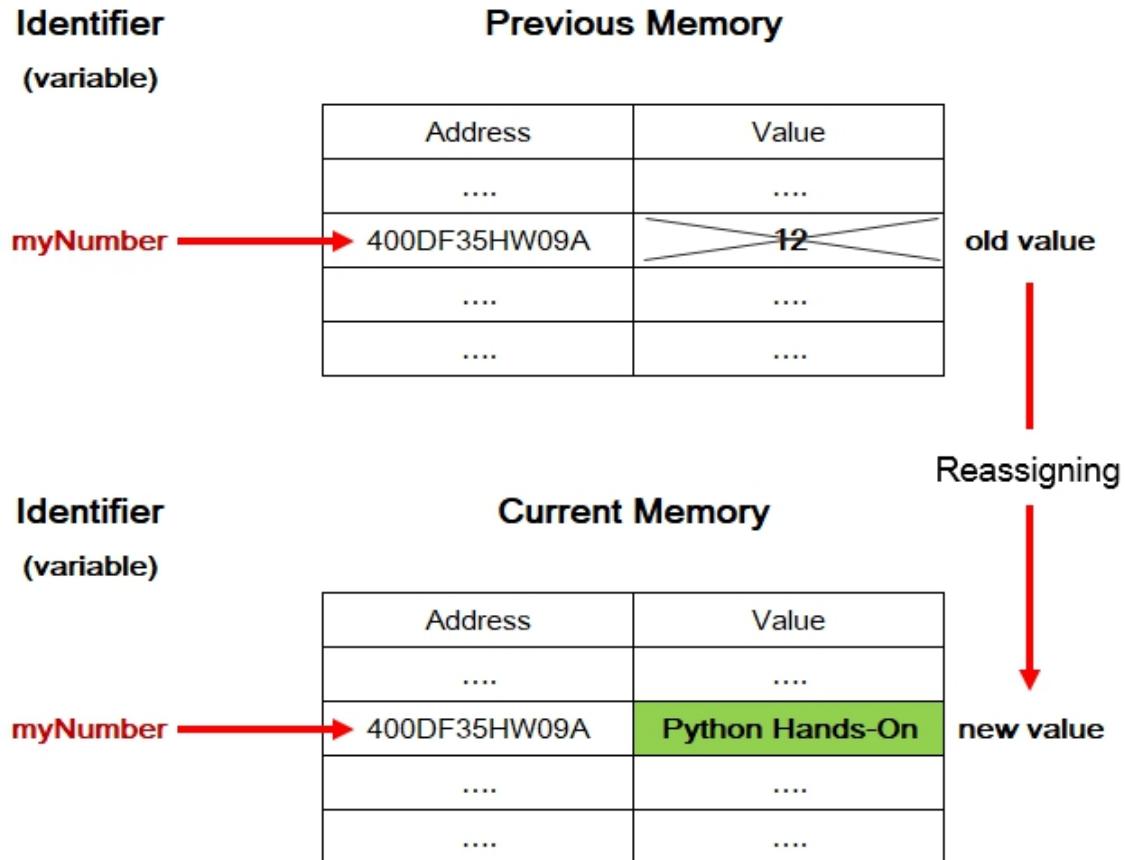


Figure 4-2: Reassigning a variable changes its value in the same memory address

Let's have a True/False quiz on this section.

OceanofPDF.com

True / False - What is a Variable?

For each questions below, decide whether it's true or false.

Q1:

Variable is a name referring to a **value**.

- True
- False

Q2:

Value is the name of the **address** of a **variable**.

- True
- False

SOLUTIONS - True / False - What is a Variable?

Here are the solutions for True / False questions.

S1:

Variable is a name referring to a **value**.

- True
- False

It's **true** because "Variable is a name referring to a value".

S2:

Value is the name of the **address** of a **variable**.

- True
- False

It's **false** because the opposite is true: Variable, is the name of the address of a value.

OceanofPDF.com

Variable Names

In Python we have some conventions while declaring the variables. In general variable names:

- should be meaningful and self-explanatory
- should be in snake case (like `snake_case`)

Snake case (stylized as `snake_case`) refers to the style of writing in which each space is replaced by an underscore (`_`) character, and the first letter of each word written in lowercase.

Here are some examples of good variable names you can define in Python:

average_speed: This name is composed of two words, it's all in lower case letters and it is in snake case form. Its name is quite self-explanatory. You can understand which data it will hold by just reading its name.

starting_index: This name is also self-explanatory and in snake case. It is obvious that it will refer to the starting index of a sequence.

number_of_visitors: Another example of good naming for variables. You can easily understand what it refers in its context.

We saw how a variable name should look like in Python. Now let's see the rules you need to keep in mind when declaring variables.

In Python, variables:

- cannot start with a number

You are not allowed to declare variables like:

- 1number
- 3_rd_position
- cannot include spaces

You are not allowed to declare variables like:

- average speed
- step size
- cannot include special characters. Variable names must be made up of only letters, numbers, and underscore (_)

You are not allowed to use the chars like: @, -, +, *, etc.

- cannot include Python Keywords

You are not allowed to declare variables like:

- and, for, return, True, if ... all of these are Python Keywords

Now let's try to define variables with names against the rules and see what happens.

```
[1 0]: 1 1number = 1245
```

```
[1 0]: File "<ipython-input-10-a2d0bf6727ee>", line 1
      1number = 1245
      ^
SyntaxError: invalid syntax
```

As you we get **SyntaxError** which tells us we cannot define a variable starting with a number.

Let's see another example. This time we will try to use space character in the variable name.

```
[1 1]: 1 average speed = 85
```

```
[1 1]: File "<ipython-input-11-233d5a2dc3ff>", line 1
      average speed = 85
      ^
SyntaxError: invalid syntax
```

We get **SyntaxError** again. Which tells us that we cannot use spaces in variable names. The right way to define this variable is to name it as **average_speed**.

```
[1 2]: 1 average_speed = 80
2 average_speed
```

[1
2]:

80

Let's try to use a forbidden character, @, in the variable name.

[1
3]:

```
1 @tenth = 10
2 @tenth
```

[1
3]:

```
File "<ipython-input-13-0b2a129a6ecf>", line 1
@tenth = 10
^
SyntaxError: invalid syntax
```

You will get the same error no matter where you use @ char in the variable name. Let's see it in the middle of the name.

[1
4]:

```
1 ten@th = 10
2 ten@th
```

[1
4]:

```
File "<ipython-input-14-7f06a1bf1aba>", line 1
ten@th = 10
^
SyntaxError: cannot assign to operator
```

The error description is different but it's still a **SyntaxError**. Let's try another special character (*) which you shouldn't be using in variable names.

[1
5]:

```
1 *a = 45
2 *a
```

[1
5]:

File "<ipython-input-15-96dea197531b>", line 1

***a** = 45

^

SyntaxError: starred assignment target must be in a list or tuple

Not let's try to use a Python Keyword as the variable name. Keywords are reserved words that have specific meaning in Python. They cannot be used as variable names, function names, or any other identifiers. We will see Python Keywords later in this chapter.

[1
6]:

1 **for** = 4

2 **for**

[1
6]:

File "<ipython-input-16-ece5be1d6013>", line 1

for = 4

^

SyntaxError: invalid syntax

The word **for** is a reserved Python keyword, so you are not allowed to use it as a variable name. That's why you get **SyntaxError**.

Important Note:

Python is CASE SENSITIVE. Case sensitive means that **a** is different from **A**. The variable **average_speed** is different from the variable **Average_Speed**. Let's prove this with an example.

We will define a variable with small letter `a` and assign value `10` to it. Then we will define another variable with capital letter `A` and assign value `45` to it. And finally we will print the values of both variables. And you will see they are different.

```
[1  
7]: 1 a = 10  
2 A = 45
```

```
[1  
8]: 1 print(a)  
2 print(A)
```

```
[1  
8]: 10  
45
```

As you see in cells 17 and 18, `a` is not equal to `A` in Python. Let's see this in another example.

```
[1  
9]: 1 num = 5  
2 Num = 12  
3 print(num)  
4 print(Num)
```

```
[1  
9]: 5  
12
```

Unicode Characters:

Although most Python environments support Unicode Characters in variable names, I strongly recommend not use them while declaring your variables. You should only use English

Alphabet letters. The safest way is only using ASCII (American Standard Code for Information Interchange) characters, which are allowed.

Let's say you want to define a variable with name omega (Ω), which is a letter in Greek alphabet. Instead of using its Greek symbol, write the full name, like **omega**.

```
[2
0]: 1 Ω_omega = 80
      2 print(Ω_omega)
```

```
[2
0]: 80
```

You may not get an error if you use the symbol, but to be on the safe side, avoid using non-ASCII characters.

```
[2
1]: 1 omega = 80
      2 print(omega)
```

```
[2
1]: 80
```

True / False - Variable Names

For each questions below, decide whether it's true or false.

Q1:

We can declare a variable named **True** as **True = correct.**

- True
- False

Q2:

We can create a variable named **10_mults** as **10_mults = [10, 20, 30, 40, 50].**

- True
- False

SOLUTIONS - True / False - Variable Names

Here are the solutions for True / False questions.

S1:

We can declare a variable named **True** as **True = correct.**

- True
- False

It's **false** because: True is a Python Keyword, so you cannot use it as a variable name.

S2:

We can create a variable named **10_mults** as **10_mults = [10, 20, 30, 40, 50].**

- True
- False

It's **false** because: Variable Names cannot start with a number.

Python Data Types

As you learned in the previous chapter, every value has a type. Variables can store data of different types, and different types can do different things. You will learn more about the types in the next chapters.

Python has the following built-in data types:

Text Type	<code>: str</code>
Numeric Types	<code>: int, float, complex</code>
Sequence Types	<code>: list, tuple, range</code>
Mapping Type	<code>: dict</code>
Set Types	<code>: set, frozenset</code>
Boolean Type	<code>: bool</code>
Binary Types	<code>: bytes, bytearray, memoryview</code>

And we already know that, to get the type of a variable we use the built-in function `type()`. Let's see each of the data types with an example.

Text Type:

Strings are text types in Python. In other words they are sequences of character data. The string type in Python is called `str`. String literals may be delimited using either single (‘) or double quotes (“). All the characters between the opening delimiter and matching closing delimiter are part of the string:

Let's see a string with single quotes:

```
[2] 1 # str (string) with single quotes
```

```
2 text = 'This is a single quoted string'  
3 type(text)
```

```
[2]  
2]: str
```

And now let's define another string but with double quotes this time.

```
[2]  
3]: 1 # str (string) with double quotes  
2 text = "This is a double quoted string"  
3 type(text)
```

```
[2]  
3]: str
```

Numeric Types:

There are three numeric types in Python: **int**, **float**, **complex**

Int: int, or integer, is a whole number, positive or negative, without decimals of unlimited length. Examples: 1, 400, -500780, 999999999999 etc.

```
[2]  
4]: 1 # int (integer)  
2 integer_number = 16  
3 type(integer_number)
```

```
[2]  
4]: int
```

Float: float, or "floating point number" is a number, positive or negative, containing one or more decimals. Examples: 1.25, 4.6, 8.0, -20.758, -1.0 etc.

```
[2 5]: 1 # float
        2 floating_point_number = 4.6
        3 print(type(floating_point_number))
```

```
[2 5]: <class 'float'>
```

Complex: complex numbers are written with a "j" as the imaginary part. Examples: 2+3j, 6j, -8j etc.

```
[2 6]: 1 # complex
        2 complex_number = 2+3j
        3 print(type(complex_number))
```

```
[2 6]: <class 'complex'>
```

Sequence Types:

In Python, sequence types are: **list, tuple, range**

List: Lists are used to store multiple items in a single variable.

Lists are created using square brackets, [].

```
[2 7]: 1 # list
        2 my_list = [1, 2, 3, 4, 5]
        3 print(my_list)
```

```
[2 7]: [1, 2, 3, 4, 5]
```

Let's also print the type of **my_list** variable.

```
[2 8]: 1 type(my_list)
```

[2
8]:

list

Tuple: Tuples are used to store multiple items in a single variable. A tuple is a collection which is ordered and unchangeable. Tuples are written with round brackets, ().

[2
9]:

```
1 # tuple
2 three_letters = ('A', 'B', 'C')
3 print(three_letters)
4 type(three_letters)
```

[2
9]:

```
('A', 'B', 'C')
tuple
```

Range: The range type represents a sequence of numbers and is commonly used for looping a specific number of times in for loops. Range type variables are created with the **range()** function.

[3
0]:

```
1 # range
2 up_to_five = range(5)
3 print(up_to_five)
4 type(up_to_five)
```

[3
0]:

```
range(0, 5)
range
```

Let's see how we loop over a range. You haven't learned the loops yet but don't worry, it is very easy to understand.

[3
1]:

```
1 for i in up_to_five:
```

```
2     print(i)
```

```
[3 1]:
```

```
0
1
2
3
4
```

And that's it. You told Python to loop over **up_to_five** range and it did. It printed the integers from 0 to 5 (not included).

Mapping Type:

Mapping Type is **dictionary (dict)** in Python.

Dictionary: dict, or dictionary, is used to store data values in **key:value** pairs. A dictionary is a collection which is does not allow duplicate keys. Dictionaries are written with curly brackets, {}, and have keys and values.

```
[3 2]:
```

```
1 # dictionary
2 user = {
3     "name": "Musa",
4     "lastname": "Arda",
5     "language": "Python"
6 }
7
8 print(user)
```

```
[3 2]:
```

```
{'name': 'Musa', 'lastname': 'Arda', 'language': 'Python'}
```

We define a dictionary variable named **user**. In this variable, “name”, “lastname” and “language” are keys, “Musa”,

“Arda” and “Python” are values. A dictionary is a mapping between keys and values.

Let’s print the type of `user` variable we defined in cell 32.

```
[3] 1 type(user)
```

```
[3] 1 dict
```

Set Types:

In Python set types are: **set** and **frozenset**. We will only cover **set** in this book.

Set: Sets are used to store multiple items in a single variable. A set is a collection of unique items. In other words, set items do not allow duplicate values.

```
[3] 1 # set
[4] 2 set_of_numbers = {1, 2, 2, 3, 1, 3, 4, 4, 1}
[3] 3 print(set_of_numbers)
```

```
[3] 1 {1, 2, 3, 4}
[4]
```

In cell 34, we created a set variable, `set_of_numbers`. And as you see, there were duplicate numbers in the sequence we provide while defining it. But when we print the variable, we see no duplicates in it. And that’s exactly how a set works in Python.

Let’s print the type for `set_of_numbers` variable.

```
[3] 1 type(set_of_numbers)
[5]
```

```
[3  
5]: set
```

Boolean Type:

Boolean type is simply **bool** in Python.

In programming, most of the time, you need to evaluate an expression. Evaluation is determining if the expression is either **True** or **False**. This is where Booleans came into the picture. Boolean values are two constant objects **True** and **False**. They are used to represent truth values in Python.

Let's see how we define bool variables in Python.

```
[3  
6]: 1 # booleans  
     2 correct = True  
     3 wrong = False
```

And let's print their values:

```
[3  
7]: 1 print(correct)  
     2 print(wrong)
```

```
[3  
7]: True  
     False
```

Finally let's see their types:

```
[3  
8]: 1 type(correct)
```

```
[3  
8]: bool
```

We will not cover Binary Types here. They are out of the scope of this book. Now that we know almost all the basic types in Python let's see how we can do type conversion in the next section.

OceanofPDF.com

True / False - Python Data Types

For each questions below, decide whether it's true or false.

Q1:

Numeric Types are **int**, **float** and **complex**.

- True
- False

Q2:

We use **type()** function to see the type of a variable.

- True
- False

OceanofPDF.com

SOLUTIONS - True / False - Python Data Types

Here are the solutions for True / False questions.

S1:

Numeric Types are **int**, **float** and **complex**.

- True
- False

It's **true**. In Python Numeric Types are: int, float and complex.

S2:

We use **type()** function to see the type of a variable.

- True
- False

It's **true**. We use **type()** function to see the type of a variable.

Type Conversion

You can easily convert one type to another in Python. All you have to do is to call special functions like `int()`, `float()`, `str()` etc. These are built-in functions for type conversion. There are many of them, but for now we will start with the first three of them.

int(): `int()` function converts strings and floating point numbers into integers.

Let's define a string first then convert it to an integer.

```
[3  
9]: 1 # a string variable  
      2 str_num = "24"  
      3 type(str_num)
```

```
[3  
9]: str
```

We defined a string variable `str_num` in cell 39. It is a string because it is defined with quotes (“). And we get `str` when we print its type. Now let's convert this variable to an integer.

```
[4  
0]: 1 # convert it into integer  
      2 int_num = int(str_num)  
      3 type(int_num)
```

```
[4  
0]: int
```

As you see in cell 40 line 2, we convert the string variable, `str_num`, to an integer with `int()` function. The result is a new

variable, `int_num`, which is an integer. And we see its type as `int` when we call the `type(int_num)` on it.

Let's convert a floating point number to integer this time.

```
[4 1]: 1 # create a floating point number first
        2 my_decimal = 3.71486
        3
        4 # convert it into integer
        5 my_int = int(my_decimal)
        6
        7 # print the type of my_int
        8 type(my_int)
```

```
[4 1]: int
```

As you see in cell 41, we do all the operations in one cell. First we create a floating point variable `my_decimal`. Then we convert it to integer. The `int()` function creates a new variable for us and the type of this new variable is integer. We named this new variable as `my_int`. And finally we print the type of `my_int` which is `int`.

Here you should note that, the type conversion in Python doesn't affect the original variable. In other words, the variable you pass into the parentheses of `int()` function does not change. `int()` function gives you a new variable. And this new variable is independent of the original one.

float(): `float()` function converts strings and integers to floating point numbers.

```
[4 2]: 1 my_float = float(4)
2 my_float
```

```
[4 2]: 4.0
```

In cell 42, we convert an integer value, 4, to a floating point number. And we name this new float variable as `my_float`. Then we print its value.

Here is another example for float conversion. This time we will convert a string to float.

```
[4 3]: 1 pi = float("3.14159")
2 print(pi)
3 print(type(pi))
```

```
[4 3]: 3.14159
<class 'float'>
```

str(): `str()` function converts almost every type to string.

```
[4 4]: 1 number_in_quotes = str(188)
2 print(type(number_in_quotes))
3 number_in_quotes
```

```
[4 4]: <class 'str'>
'188'
```

```
[4 5]: 1 str(3.14159)
```

[4
5]:

'3.14159'

Type Conversion is also called **Type Casting**. You may see both terms in this book. They both refer to the same type operations.

OceanofPDF.com

True / False - Type Conversion

For each questions below, decide whether it's true or false.

Q1:

The result of the code line below is **float**.

```
type("3.84")
```

- True
- False

Q2:

If you run the code lines below it will print **int** as the output.

```
num = int(3.14159)  
print(num)
```

- True
- False

SOLUTIONS - True / False - Type Conversion

Here are the solutions for True / False questions.

S1:

The result of the code line below is **float**.

```
type("3.84")
```

- True
- False

It's **false**. The value in the **type()** function is a string, not a floating point number. It is surrounded with double quotes, which makes it a string. And the result of **type("3.84")** is **str** not **float**.

S2:

If you run the code lines below it will print **int** as the output.

```
num = int(3.14159)  
print(num)
```

- True
- False

It's **false**. The **int()** function will give you a variable, **num**, with value of 3. And when you call print function as **print(num)**, Python will print its value not its type. And its value is 3.

Python Comments

When writing code in Python, it's important to make sure that your code can be easily understood by other developers. One of the main ways to increase readability of your code is by using **comments**.

Comments are lines of code that are ignored by the **Python Interpreter**. Python Interpreter is the application to run your Python code. It is included in the Anaconda installation. Interpreter runs your Python code line by line, from the beginning to the end. But it does not run comment lines in the code. It just ignores them.

You can write comments in two ways in Python: One Line Comment or Multi-Line Comments.

One Line Comment:

One Line Comment starts with a `#` sign. Python will ignore anything following the `#` sign. Like the lines below:

```
[4
6]: 1 # One Line Comment
      2
      3 # This is one line comment
```

Let's say you want to create a variable to keep the miles traveled. You can write the purpose of this variable in a comment line above it.

```
[4
7]: 1 # keeps the miles traveled
```

```
2 miles_traveled = 90000
3
4 print(miles_traveled)
```

```
[4
7]: 90000
```

And here is a sample program to demonstrate how you can improve code readability. We write the formula for average speed and we add comments for each step. One can completely understand what we are doing here even if he/she don't know Python.

```
[4
8]: 1 # average speed = displacement / change in time
2
3 # first get the displacement
4 displacement = 450 # km
5
6 # then get the change in time
7 change_in_time = 6 # hr
8
9 # calculate the average speed
10 average_speed = displacement / change_in_time
11
12 # print the result
13 print(average_speed)
```

```
[4
8]: 75.0
```

Multi-Line Comments:

Multi-Line Comments are created with a **pair of triple quotes**, either single quote or double quotes.

```
[4  
9]: 1     """  
2  
3     Multi-Line Comment  
4  
5     In Python, Multi-Line comments,  
6     are created via a pair of triple quotes: """ """  
7  
8     As in this case.  
9  
10    """
```

If you run cell 49 in JupyterLab the output will be as follows:

```
[4  
9]: '\n\nMulti-Line Comment\n\nIn Python, Multi-Line  
comments,\nare created via a pair of triple quotes:  
\n\nAs in this case.\n\n'
```

Here you see a special set of characters: `\n`. `\n` is the new line character in Python. It means an empty blank line. You can use it in strings to create an empty line and to format a plain string into multiple lines.

```
[5  
0]: 1 multi_line_text = "Line 1 \nLine 2 \nLine 3"  
2 print(multi_line_text)
```

```
[5  
0]: Line 1  
     Line 2  
     Line 3
```

```
[5  
1]: 1 print("Hello my dear \nPython")
```

```
[5  
1]: Hello my dear
```

Python

You can format any string inside a multi-line comment as you wish. You can use new lines, tabs etc., as below:

```
[5] 1     """
2]: 2     This is line 1
3     This is line 2
4     This is line 3
5     """
```

```
[5] 1\n2]: 2\n\n      This is line 1\n      This is line\n      This is line 3\n\n'
```

OceanofPDF.com

True / False – Python Comments

For each questions below, decide whether it's true or false.

Q1:

If you run the code below it will print **10** as the output.

```
# define a variable as x
# x = 10

# print its value
print(x)
```

- True
- False

Q2:

In Python;

for one line comment we use a pair of tree double

quotes: " " ",

for multi-line comments we use: # symbol.

- True
- False

SOLUTIONS - True / False - Python Comments

Here are the solutions for True / False questions.

S1:

If you run the code below it will print **10** as the output.

```
# define a variable as x
# x = 10

# print its value
print(x)
```

- True
- False

It's **false**. You will get an error if you run this code. The error will be: "**NameError**: name 'x' is not defined". It means you have not defined a variable named x yet. As you see in the code the definition of x is a comment line, **# x = 10**. Which means Python Interpreter will ignore that line. So there is no variable named x in your code. And the **print(x)** statement will raise an error.

S2:

In Python;
for one line comment we use a pair of tree double quotes: " " ",
for multi-line comments we use: # symbol.

- True
- False

It's **false**. It should be just the opposite. We use the # sign for **one line comment** and " " " " for **multi-line comments**.

Numeric Operations

We have already seen Arithmetic Operations in Chapter 3. But we didn't know much about variables at that time. Now that we know how to define variables and what the numeric type variables are, we will revisit the Arithmetic Operations again. But this time we will call it as Numeric Operations.

Numeric Operations means the operations we do on numeric data types in Python, which are **int**, **float** and **complex**. We will be using int and float types in this section. Let's see numeric operations one by one.

To begin with, we will create two numeric variables: **a** and **b**.

```
[5  
3]: 1 # define two variables  
     2  
     3 a = 30  
     4 b = 7
```

Now let's see all the arithmetic operations with these variables: addition, subtraction, multiplication, division, floor division (integer division), modulo, and power (exponentiation).

```
[5  
4]: 1 print(a + b) # addition
```

```
[5  
4]: 37
```

```
[5  
5]: 1 print(a - b) # subtraction
```

```
[5] 23
[5]: 1 print(a * b)    # multiplication
[6] 210
[6]: 1 print(a / b)    # division
[7] 4.285714285714286
[7]: 1 print(a // b)    # floor division (integer division)
[8] 4
[8]: 1 print(a % b)    # modulo
[9] 2
[9]: 1 print(a**b)    # power
[10] 21870000000
```

While dealing mathematical operations, one of the thing you should always keep in mind is the **order of operations**. Which operation you should compute first. And here is the hint to make it easy to remember: **PEDMAS**. It stands for Parenthesis, Exponents, Division, Multiplication, Addition, and Subtraction. The first letter from each operation in order.

Order of Operations: Parenthesis, Exponents, Division, Multiplication, Addition, Subtraction

Here is an example for the order of operations.

```
[6  
1]: 1 print(4 * 5 - 9 + 14 / 7)
```

```
[6  
1]: 13.0
```

Inside the `print()` function in cell 61, the first operation is the division, `14 / 7` which results in `2.0`. The second one is the multiplication, `4 * 5` which is `20`. And then addition and subtraction. The final result is `13.0`. Since we have one floating point number in the calculation, the whole result becomes a float.

Shorthand Operations:

In Python, for most of the arithmetic operations we have the shorthand form. Let's see them in action.

```
[6  
2]: 1 a = 9  
2  
3 # increment a by 1  
4 a = a + 1  
5 a
```

```
[6  
2]: 10
```

In cell 62, we first define a variable, `a`, and assign value 9 to it. Then we increment `a` by 1 in line 4. And then we print its final value. As you see, the result is 10.

Let's increment it by 4, this time.

```
[6  
3]: 1 a = a + 4  
2 a
```

```
[6  
3]: 14
```

We add 4 to the old value of `a`, which was 10. And now the final value of `a` is 14. This is ordinary way for the addition operation. Now let's see the **shorthand increment** (`+=`).

```
[6  
4]: 1 # Shorthand Increment -> +=  
2  
3 a += 1 # a = a + 1  
4 a
```

```
[6  
4]: 15
```

In line 3 of cell 64, you see the shorthand increment operation. It adds 1 to the value of `a` and then reassign the new value to `a`. So `a += 1` is exactly the same as `a = a + 1`.

Now let's see **shorthand decrement** (`-=`).

```
[6  
5]: 1 # Shorthand Decrement -> -=  
2  
3 a -= 6 # a = a - 6  
4 a
```

```
[6  
5]: 9
```

As you see in cell 65, we decreased the value of `a` by 6 (`a -= 6`). And its final value is `9`.

Now let's see **shorthand multiplication** (`*=`).

```
[6
6]: 1 # Shorthand Multiply -> *=
      2
      3 a *= 5      # a = a * 5
      4 a
```

```
[6
6]: 45
```

We used shorthand multiplication (`a *= 5`) to multiply the current value of `a`, by 5. Its value was 9, so we get 45 when we multiply it with 5. And finally we reassigned this value to `a`, which is now 45.

Finally let's see **shorthand division** (`/=`).

```
[6
7]: 1 # Shorthand Division -> /=
      2
      3 a /= 11      # a = a / 11
      4 a
```

```
[6
7]: 4.090909090909091
```

In cell 67, we use shorthand division (`a /= 11`) to divide the current value of `a`, by 11. Its value was 45 and we get `4.090909090909091` when we divide it by 11. We then assign this value back to `a`.

OceanofPDF.com

True / False – Numeric Operations

For each questions below, decide whether it's true or false.

Q1:

The result of Shorthand Addition Operator `+=`, will print **20** on the screen.

```
x = 20
```

```
x += 20
```

```
print(x)
```

- True
- False

Q2:

The last value of `p` will be **12.0**.

```
p = 8
```

```
p = p * 3
```

```
p /= 2
```

- True
- False

SOLUTIONS - True / False - Numeric Operations

Here are the solutions for True / False questions.

S1:

The result of Shorthand Addition Operator `+=`, will print **20** on the screen.

```
x = 20
x += 20
print(x)
```

- True
- False

It's **false**. The last value of **x** is **40**.

S2:

The last value of **p** will be **12.0**.

```
p = 8
p = p * 3
p /= 2
```

- True
- False

It's **true**. First **p** is 8. Then it will become 24 when multiplied by 3. And finally it will become 12.0 via `/=` operation.

String Operations

In Python, Strings are displayed via **double quotes (" ")** or **single quotes (' ')**. There is one important point that you should note for using the quotes. **Opening and Closing Quotes should match.** Which means, if you start with a double quote, then you should end with a double quote. Same goes for single quotes also.

```
[6  
8]: 1 string_double = "This is a double quotes string."  
2 print(string_double)
```

```
[6  
8]: This is a double quotes string.
```

In cell 68 we define a string with double quotes. We start with double quotes and finish with the same. Now let's define another string with single quotes.

```
[6  
9]: 1 string_single = 'This is a single quote string'  
2 print(string_single)
```

```
[6  
9]: This is a single quote string
```

What if, you need to use both types of quotes in the same string? Let's see it with an example:

```
[7  
0]: 1 # If you need to use both in the same string  
2  
3 single_in_double = "Python's the best language."  
4 print(single_in_double)
```

```
[7  
0]: Python's the best language.
```

In cell 70, we use single quotes inside the double quotes. Now let's see how we can use double quotes inside the single ones.

```
[7  
1]: 1 double_in_single = 'Jenny said "Run, Forest, run",  
2 and he did.'  
2 double_in_single
```

```
[7  
1]: 'Jenny said "Run, Forest, run", and he did.'
```

Now let's see if we confuse the single quotes with double ones in the same string. We will start with double quotes but end up with single ones.

```
[7  
2]: 1 false_usage = "This is a 'double quote'
```

```
[7  
2]: File "<ipython-input-72-8168303684c0>", line 1  
      false_usage = "This is a 'double quote'  
                           ^  
SyntaxError: EOL while scanning string literal
```

As you see in cell 72, we get **SyntaxError**, which tells us that we didn't define the string in a proper way. Starting quote was a double one, but the ending one was a single quote.

In general, you cannot do arithmetic operations on Strings, in Python. I used the word “in general” because there are some arithmetic operations which works on Strings.

Let's start with some of the forbidden operations for Strings.

Which are:

- subtraction (-)
- division (/)

Let's see what happens if you try to subtract a string from another string. We will use two strings '4' and '3'. Although they look integers to the eye, they are strings, because of the quotes.

```
[7] 3]: 1 '4' - '3'  
[7] 3]: -----  
-----  
TypeError Traceback (most recent call last)  
<ipython-input-73-76ad527f5a7e> in <module>  
----> 1 '4' - '3'  
  
TypeError: unsupported operand type(s) for -: 'str' and 'str'
```

As you see in the output of cell 73, we get: **TypeError**: unsupported operand type(s) for -: 'str' and 'str'. Which says you cannot do subtraction (-) on string types.

Let's try division too.

```
[7] 4]: 1 '4' / '3'  
[7] 4]: -----  
-----
```

```
TypeError                                 Traceback (most
recent call last)
<ipython-input-74-6b4018763e91> in <module>
----> 1 '4' / '3'

TypeError: unsupported operand type(s) for /: 'str'
and 'str'
```

Again we get **TypeError**: unsupported operand type(s) for /: 'str' and 'str'. Which says, you are not allowed to do division on strings.

We see that operations like subtraction and division are forbidden for strings. But there are operations which we are allowed to use with strings.

Exceptional Case: addition (+) and multiplication (*) are possible with strings. But with its own logic, not as the way we see in numbers. Let's these operations with examples.

First create two string variables, **first** and **second**. Then try addition and multiplication operations on them.

```
[7] 1 # define two string variables
5]: 2 first = "Sunny"
     3 second = "Days"
```

Now let's try to add them and see the result. Before reading the next lines and running it on your computer, try to make a guess. What do you expect if you add two strings together?

```
[7] 1 # Addition in Strings (Concatenation)
6]: 2
```

```
3 first + second
```

```
[7] 6: 'SunnyDays'
```

As you see in the output of cell 76, addition for strings is nothing but concatenation. Python adds the string characters one after another. No matter how many strings you add.

```
[7] 7: 1 first + second + first + first + second
```

```
[7] 7: 'SunnyDaysSunnySunnyDays'
```

Now let's see how multiplication works with strings. We already have a variable named, `first`. Let's multiply it with 4.

```
[7] 8: 1 # Multiplication in strings
      2
      3 4 * first
```

```
[7] 8: 'SunnySunnySunnySunny'
```

And as you see, multiplying a string by a number means, repeating the string by that amount. Here are two more examples.

```
[7] 9: 1 6 * '5'
```

```
[7] 9: '555555'
```

```
[8] 0: 1 3 * '8'
```

[8
0]:

‘888’

OceanofPDF.com

True / False – String Operations

For each questions below, decide whether it's true or false.

Q1:

In Python, we can use **double quotes** (" ") and **single quotes** (' ') to create Strings.

- True
- False

Q2:

The Interpreter will run the code below without an error.

```
print("Python's Syntax allow create strings with both ' and " quotes.")
```

- True
- False

OceanofPDF.com

SOLUTIONS - True / False - String Operations

Here are the solutions for True / False questions.

S1:

In Python, we can use **double quotes** (" ") and **single quotes** (' ') to create Strings.

- True
- False

It's **true**. In Python, we can use double quotes (" ") and single quotes (' ') to create Strings.

S2:

The Interpreter will run the code below without an error.

```
print("Python's Syntax allow create strings with both '  
and " quotes.")
```

- True
- False

It's **false**. The double (" ") quotes in the middle will finalize the string. So the last part which is, " **quotes.**", will raise **SyntaxError**.

Keywords

Python has a set of **keywords** that are reserved words and cannot be used as variable names, function names, or any other identifiers. All the Python keywords are in the **keyword** package.

To be able to use a package in Python, we use **import** statement. **Import** itself is a keyword and it's used to import packages in your project. We will be using import statement quite a lot during this book. So, don't worry, you will get used to it.

```
[8
1]: 1 # first import the keyword package
      2
      3 import keyword
```

And that's it. You import the keyword package and it's ready to use. Let's see the complete list of Python keywords. The keyword list is in a variable called **kwlist** inside keyword package. To get it, we use the syntax of **<package_name>.<variable_name>**, like **keyword.kwlist**. We will get it and assign it to a local variable.

```
[8
2]: 1 # Get keyword list
      2 # And assign this list to a variable
      3
      4 python_keywords = keyword.kwlist
      5
      6 # print keywords
      7 print(python_keywords)
```

[8
2]:

```
['False', 'None', 'True', 'and', 'as', 'assert', 'async',  
'await', 'break', 'class', 'continue', 'def', 'del', 'elif',  
'else', 'except', 'finally', 'for', 'from', 'global', 'if',  
'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or',  
'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']
```

And in the output of cell 82, you see the complete list of keywords in Python. All you have to do is to call `keyword.kwlist` variable, assign it to a variable and then print it.

You are not allowed to use any of these keywords as variable names. It's forbidden and you will get `SyntaxError` if you try. Let's assume we want to create a variable with name “**import**”.

[8
3]:

```
1 import = 45
```

[8
3]:

```
File "<ipython-input-83-82fef50b7975>", line 1
```

```
    import = 45
```

```
    ^
```

```
SyntaxError: invalid syntax
```

As you see in cell 83, `import` is a keyword, so Python does not allow using it as a variable name. Let's try it one more time. This time, we will try to name our variable as “**global**”.

[8
4]:

```
1 global = 'ABC'
```

[8
4]:

```
File "<ipython-input-84-f145bd44b7db>", line 1
```

```
    global = 'ABC'
```

```
    ^
```

```
SyntaxError: invalid syntax
```

`SyntaxError` again, `global` is also a keyword which is reserved.

Since keywords are not allowed, you should always be careful while naming your variables. If you are not sure, whether a name is a reserved keyword or not, there is an easy way to check it. You can use `iskeyword()` method in the `keyword` package.

Let's say you want to learn if the word “`not`” is a Python keyword or not:

```
[8  
5]: 1 # To check -> keyword.iskeyword()  
2  
3 keyword.iskeyword('not')
```

```
[8  
5]: True
```

In cell 85, we check whether the word “`not`” is a keyword or not. To do this, we just pass it into the `iskeyword()` method as: `keyword.iskeyword('not')`. The result of the cell is `True`, which means, yes the word “`not`” is a Python keyword.

None:

Let's check another one. This time check for the word “`None`”.

```
[8  
6]: 1 keyword.iskeyword('None')
```

```
[8  
6]: True
```

As you may guessed, the word “**None**” is also a Python keyword. **None** is the special keyword used to define a null value, or no value at all. Simply it means ‘nothing’. **None** is not the same as 0, **False**, or an empty string. It is a data type of its own (**NoneType**) and only **None** can be **None**.

Let’s try another one. We will check the word “**return**” this time.

```
[8  
7]: 1 keyword.iskeyword('return')
```

```
[8  
7]: True
```

The word “**return**” is another keyword for Python. Let’s check something funny. I wonder if the word “**python**” itself is a keyword for Python or not.

```
[8  
8]: 1 keyword.iskeyword('python')
```

```
[8  
8]: False
```

Interesting, right? The word “**python**” is not a reserved word for Python. How did we know that? Because the output of cell 88 prints False. False means, it’s not a reserved word.

As a final check, let’s look for the word “**word**” itself. Is “**word**” a reserved keyword for Python?

```
[8  
9]: 1 keyword.iskeyword('word')
```

[8
9]:

False

No, it's not. You can use “**word**” as a variable name.

And here, we finalize Chapter 4, Variables. We learned quite a lot about variables; how to declare them, how to name them, what are the data types, how to do type conversion, comments, numeric and string operations and keywords in Python.

After the True/False questions on Keywords section, you will have a quiz on Variables. The author of this book is a firm believer of “**learning by doing**”. So, you are expected to solve the quiz questions by yourself. Before checking the solutions make sure you try and solve all the questions. Or at least, you tried hard.

OceanofPDF.com

True / False - Keywords

For each questions below, decide whether it's true or false.

Q1:

We import the **keyword** package to see Python Keywords as below:

```
import keyword
```

- True
- False

Q2:

To see whether word **for** is a keyword for Python or not we can use:

```
for.iskeyword()
```

- True
- False

SOLUTIONS - True / False - Keywords

Here are the solutions for True / False questions.

S1:

We import the **keyword** package to see Python Keywords as below:

```
import keyword
```

- True
- False

It's **true**. **import keyword** is the right syntax to import the keyword package.

S2:

To see whether word **for** is a keyword for Python or not we can use:

```
for.iskeyword()
```

- True
- False

It's **false**. To check if **for** is a Python keyword the right syntax is: **keyword.iskeyword('for')**.

QUIZ - Variables

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_Variables.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *4_Variables*. Here are the questions for this chapter:

QUIZ - Variables:

Q1:

Fill in the blanks to print "Hello World!":
.....("Hello World!")

Q2:

Fix the typo (coding error) below:
print("Alice's Adventures')

Q3:

Convert the line below into a Comment Line. (One line comment)

I am a Comment Line :)

Q4:

Convert the lines below into comment lines. (multi-line comment):

This comment line

Is multiple lines comment

Not just one...

Q5:

Print the text below exactly as it is shown:

Who in the world am I?

 Ah, that's the great puzzle!

 'Alice in Wonderland'

Q6:

Create a variable named **car_model** and assign the value 'Volvo' to it.

Q7:

Create a variable named **x** and set its value to 50.

Q8:

Create two variables **x** and **y**. And assign them as 20 and 50 respectively.

Create a variable **z**, and assign **x + y** to it.

Then print **z**.

Complete the missing codes below according to the declarations.

```
1 x = 20
2 y = 80
3
4 ... = x + y
5 print(...)
```

Q9:

Remove forbidden characters from the variable name, and print its value.

2my-first_variable*β = "Beta"

Answer: Beta

Q10:

Fill in the blanks to assign three variables at the same time. And then print their values.

Hint: In Python you can do multiple assignments at once.

Python assigns variables from right to left.

x ... y ... z = "Python"

Answer:

Python

Python

Python

SOLUTIONS - Variables

Here are the solutions for the quiz for Chapter 4 - Variables.

SOLUTIONS - Variables:

S1:

It asks to fill the blanks. We have to use **print()** function to be able to print something on the screen. And the solution is:

```
[  
1  1  # S 1:  
]:  
2  
3  print("Hello World!")
```

```
[  
1  Hello World!  
]:
```

S2:

The typo here is about the quotes in strings. Since the text starts with double quotes, it should end also with them. But in the question the quote is a single quote.

```
[  
2  1  # S 2:  
]:  
2  
3  print("Alice's Adventures")
```

```
[  
2  Alice's Adventures  
]:
```

S3:

Single line comments are created with # sign. So to convert any line or code segment into the Python comments we simply put # sign in front of them.

```
[  
3 1 # S 3:  
]:  
2  
3 # I am a Comment Line :)
```

S4:

To convert a text into multi-line comments, we surround it with a pair of triple quotes, which can be either double or single quotes.

```
[  
4 1 # S 4:  
]:  
2  
3 """  
4 This comment line  
5 Is multiple lines comment  
6 Not just one...  
7 """
```

S5:

We will solve this questions in two different ways.

The first way is to use three separate **print()** statements.

The second way is more elegant. Into a single **print()** function, we will pass the text as it is, inside a pair of triple quotes, which will make it a formatted text. A pair of triple quotes (either single or double) is also used for printing formatted text.

```
[  
5 1 # S 5:  
]:  
2  
3 # 1st Way  
4 print("Who in the world am I?")  
5 print(" Ah, that's the great puzzle!")  
6 print(" 'Alice in Wonderland'")
```

```
7
8  # 2nd Way
9  print("""
10 Who in the world am I?
11     Ah, that's the great puzzle!
12         'Alice in Wonderland'
13     """)
```

```
[  Who in the world am I?
5   Ah, that's the great puzzle!
]:    'Alice in Wonderland'
```

S6:

```
[ 1 # S 6:
]: 2
3 car_model = 'Volvo'
4 car_model
```

```
[ 6 'Volvo'
]:
```

S7:

```
[ 1 # S 7:
]: 2
3 x = 50
4 x
```

```
[ 7 50
]:
```

S8:

```
[  
8 1 # S 8:  
]:  
2  
3 x = 20  
4 y = 80  
5  
6 z = x + y  
7 print(z)
```

```
[  
8 100  
]:
```

S9:

Here, there are multiple mistakes in the variable name. First of all it cannot start with a number, so we will delete the number 2 at the beginning. The new name is **my-first_variable*β**. Now the next problem is the dash sign, or minus, (-). You are not allowed to use it in the variable name because it has a special meaning as a math operation. You can use underscore (_) character instead of minus sign. So let's replace “-“ with “_”. The new name is **my_first_variable*β**. And there are two special characters which are forbidden, * and β. We will remove them. And finally our variable name is **my_first_variable**.

```
[  
9 1 # S 9:  
]:  
2  
3 my_first_variable = "Beta"  
4 print(my_first_variable)
```

```
[  
9 Beta  
]:
```

S10:

In Python you can assign multiple variables with the same value at once, by using = consecutively. They will all get created and assigned simultaneously. Before doing a multiple assignment, let us first create every variable, **x**, **y** and **z** one by one. You will see that it is exactly the same thing to create them all at once. Here is the solution:

```
[1
0]: 1  # S 10:
      2
      3  # create them one by one
      4  # x = 'Python'
      5  # y = 'Python'
      6  # z = 'Python'
      7
      8  # create them all at once with multiple
      9  # assignment
      9  x = y = z = 'Python'
      10
      11 # now print their values
      12 print(x)
      13 print(y)
      14 print(z)
```

```
[1
0]: Python
      Python
      Python
```

5. Functions I

What is a Function?

In Chapter 4, we learned one of the main building blocks of programming, **Variables**. In this chapter we will learn another building block which is **Functions**. Functions are crucial to every computer program. They are very useful and have many advantages. One of the most prominent advantage of functions is, they prevent code repetitions. In this book, we have two chapters dedicated to functions.

In computer programming, a **Function** is a special code fragment written to perform a specific task. A function only runs when it is called. It may have arguments (parameters). And it may return data as a result.

Calling a Function:

To call a function you just write the function name followed by parentheses: **function_name()**. Here the function takes no arguments, because the parentheses are empty. This is how we call a function without any arguments.

If we want to call a function with arguments we have to pass them inside the parentheses: **function_name(argument_1, argument_2, ...)**

So far, we have called some of the Python built-in functions. Like `print()`, `type()`, `int()`, `float()`, `str()`, etc. Let's call them again and this time our focus is how we call them and how we pass arguments to them.

We will start with the `type()` function:

```
[  
1  1  type(42)  
]:
```

```
[  
1  1  int  
]:
```

As you see in cell 1, we call the `type()` function with an integer argument 12 as: `type(42)`. And it returns `int`. The output of cell 1, is the return value of `type()` function. JupyterLab prints it automatically for us.

```
[  
2  1  type('a sunny day')  
]:
```

```
[  
2      str  
]:
```

In cell 2, we pass a string 'a sunny day' as an argument. And the `type()` function returns `str` as the result.

Let's move on with the `int()` function:

```
[  
3  1  # int() -> converts the given argument to int  
]:
```

```
2
3 int(5.68)
[3]: 5
```

We pass a floating point number, 5.68, into the `int()` function. The function converts (casting) that float to an integer, 5, and returns it back.

What if we pass a value which cannot be converted to an integer? A arbitrary string for example. Let's try:

```
[4]: 1 int('value')
[4]: -----
ValueError: invalid literal for int() with base 10: 'value'
```

Traceback (most recent call last)
<ipython-input-4-8ea25d043ddd> in <module>
----> 1 int('value')

Obviously, the text “value” cannot be converted to an integer. That's why we get **ValueError** in cell 4. Which tells us, Python cannot convert “value” to ant `int`.

If you try a string which **can be** converted to an integer, then Python will do it. A string like “82” for example. Yes it is a string,

because it is in quotes, but it can be converted to a number. Let's try:

```
[  
5  1 int("82")  
]:
```

```
[  
5    82  
]:
```

As you see, a string of “82” is easily converted to an integer, because it **convertible**.

Let's move on with the `str()` function:

```
[  
6  1 # str() -> converts (casts) the given argument into  
]:  
    string  
2  
3 str(45)
```

```
[  
6    '45'  
]:
```

In cell 6, we pass an integer, 45, into the `str()` function and it converts the `int` to a string (`str`) and returns that string as ‘45’.

```
[  
7  1 str(4.57)  
]:
```

```
[  
7    '4.57'  
]:
```

```
[8]: 1 str('Gotham')
]:
```

```
[8]: 'Gotham'
]:
```

As you see in cells 7 and 8, the `str()` function can convert anything to string.

Finally, let's see the `float()` function:

```
[9]: 1 # float() -> casts the given argument into float
]: 2
3 float(12)
```

```
[9]: 12.0
]:
```

In cell 9, we pass an `int`, `12`, into the `float()` function and it returns a floating point number, `12.0`.

Now let's pass a string which can be converted to a floating point number.

```
[1
0]: 1 float('4')
```

```
[1
0]: 4.0
```

In cell 10, Python converts `'4'`, which is a string, to `4.0` which is a float. This also works if we pass a negative number in quotes,

like “-78”. Let’s see:

```
[1] 1 num = '-78'  
1]: 2 float(num)
```

```
[1] 1 -78.0  
1]:
```

As you see in cell 11, “-78” is converted into a float because it’s convertible. Let’s try another string, “78-”, which is obviously not a number.

```
[1] 1 num = '78-'  
2]: 2 float(num)
```

```
[1] 1 -----  
2]: 2 -----  
      ValueError                                     Traceback (most  
      recent call last)  
      <ipython-input-12-e61f3f90eb4d> in <module>  
          1 num = '78-'  
          ----> 2 float(num)  
  
      ValueError: could not convert string to float: '78-'
```

As you might guess, we get a **ValueError** from Python Interpreter. **ValueError**: could not convert string to float: '78-'.

Now that we know how we call a function, either with or without arguments, let’s move on with some Math Functions. After you get accustomed to calling functions, we will define our own functions.

OceanofPDF.com

True / False - What is a Function?

For each questions below, decide whether it's true or false.

Q1:

Functions can take arguments but they **can not** return a value.

- True
- False

Q2:

You will get **-18.0** if you run **float("-18")**.

- True
- False

SOLUTIONS - True / False - What is a Function?

Here are the solutions for True / False questions.

S1:

Functions can take arguments but they **cannot** return a value.

- True
- False

It's **false**. Functions can take parameters and **can** return a value.

S2:

You will get **-18.0** if you run **float("-18")**.

- True
- False

It's **true** because the **float("-18")** expression will return **-18.0**.

Math Functions

We use the **math** module to do mathematical operations in Python. A **module** is any Python file that includes executable Python code. Most of the math related functions can be found in the **math** module. In this section we will import the **math** module and use some of the functions in it.

Let's import **math** module first:

```
[1] 1 # import math module
[3]: 2 import math
```

Let's print the **math** module to see basically what it looks like:

```
[1] 1 # see the module info
[4]: 2 print(math)

[1] 4: <module 'math' (built-in)>
```

The module info in cell 14 is just the basic data. Now let's see the details. To get the detailed info about any module or function we use **help()** function. Let's call it for math module.

```
[1] 1 # see detailed docs
[5]: 2
3 # help() -> gives you the detailed docs
4 help(math)
```

```
[1] 5: Help on built-in module math:
```

NAME

math

DESCRIPTION

This module provides access to the mathematical functions defined by the C standard.

FUNCTIONS

acos(x, /)

Return the arc cosine (measured in radians) of x.

acosh(x, /)

Return the inverse hyperbolic cosine of x.

asin(x, /)

Return the arc sine (measured in radians) of x.

...

....

The detailed information in the output of cell 15, is too much that I didn't put all of the lines here. Whenever you need detailed information about an object in Python you can use `help()` function.

Now let's start using the math module. Let's say we need to use the famous pi number.

```
[1] 1 # we can call any function from module -> .
6]: 1 notation
     2 math.pi
```

```
[1] 6]: 3.141592653589793
```

Now let's see how we can use the math module on a real example. The example details are in the cell below as comment lines.

```
[1  
7]: 1 # Example:  
2  
3 # What is the perimeter (circumference) of a  
4 # circle having radius of 10 cm?  
5 # Perimeter = 2 * pi * r  
6  
7 # radius  
8 r = 10  
9  
10 # perimeter  
11 perimeter = 2 * math.pi * r  
12 print(perimeter)
```

```
[1  
7]: 62.83185307179586
```

As you see in cell 17, line 10 we use the pi number in the math module, to calculate the circumference of a circle. We define a variable named `r` to keep the radius. Then we calculate the circumference by the formula of `2 * math.pi * r` and we assign the result to a variable named `perimeter`.

Let's see another example. This time we will calculate the sine of **30 degrees**. We will be using the `math.sin()` function. If you check its documentation via `help(math.sin)`, you will see that its argument should be radians, not degrees. Here is the

documentation: `sin(x, /)` Return the sine of `x` (measured in radians). So to be able to use `math.sin()` function we need the radians of 30 degrees. First we will calculate the radians of 30 degrees, then we will pass that value into `math.sin()` function. See the steps below:

```
[1
8]: 1      # Example:
      2
      3      # Calculate the sine of 30 degrees -> sin(30)
      4
      5      degree = 30
      6
      7      # calculate radian
      8      radian = math.radians(degree)
      9
      10     # calculate the sine
      11     sine = math.sin(radian)
      12
      13     print(sine)
```

```
[1
8]: 0.4999999999999994
```

Composition of Functions:

In cell 18 above, we define a variable named `degree` to keep the value of 30 degrees (line 5). Then we calculate the radian of the variable `degree` and assign to a variable named `radian` (line 8). The purpose of `radian` variable is to keep the radians returned from `math.radians(degree)`. Then finally in line 11 we get the

sine value via the function call `math.sin(radian)`. And we assign the result to a variable named `sine`.

The two variables, `degree` and `radian`, in cell 18, are only there to keep some values to be used in the next step. We call these kind of variables as **temporary variables**. Most of the time they just serve to keep some temporary values.

Instead of using temporary variables, we can **chain the function calls**. Chaining the functions is called **composition**. We can write more compact code via composition of functions. You will see plenty of examples on function chaining throughout this book.

Chaining is very simple, you just use the function call whenever you need a temporary variable. In the cell 19 below, we will chain the function calls we did in cell 18. And the result will be the same.

```
[1  
9]: 1 # chain the functions  
2 # first call math.radians(degree)  
3 # then pass it into math.sin()  
4  
5 degree = 30  
6 sine = math.sin(math.radians(degree))  
7 print(sine)
```

```
[1  
9]: 0.4999999999999994
```

This is the chained function call:
`math.sin(math.radians(degree))`

OceanofPDF.com

True / False - Math Functions

For each questions below, decide whether it's true or false.

Q1:

We can use **help()** function to get information about a function or a module.

- True
- False

Q2:

To get **sine** value of variable named **radian** and assign this value to a variable named **sine_rad** we can do:

```
sine_rad.math.sin(radian)
```

- True
- False

SOLUTIONS - True / False - Math Functions

Here are the solutions for True / False questions.

S1:

We can use **help()** function to get information about a function or a module.

- True
- False

It's **true**. **help()** function will give info about the object you asked for.

S2:

To get **sine** value of variable named **radian** and assign this value to a variable named **sine_rad** we can do:

```
sine_rad.math.sin(radian)
```

- True
- False

It's **false**. It should be **sine_rad = math.sin(radian)**. Assignment is done via "=".

Defining Functions

How do we define a function in Python? It's very easy, we use the **def** keyword. Let's define our first function:

```
[20]: 1 # Without Parameters
      2
      3 def my_first_function():
      4     # print line
      5     print("My first function")
```

After the **def** keyword we write the function name, followed by parentheses and a colon (:). The function name in cell 20 is **my_first_function**. And we have no parameters for this function that's why the parentheses are empty. The colon (:) starts the function block (scope). Every line of code which is **under the colon and indented**, belongs to this function.

To run a function we defined, we have to call it. For example, the function in cell 20, **my_first_function**, will not print anything on the screen until we call it. Now let's call it.

```
[21]: 1 # call the function
      2
      3 my_first_function()
```

```
[21]: This is my first function
```

And now we call the function and it prints the text “This is my first function”.

Indentation:

Indentation in Python is crucial. Actually it's a part of the syntax. Python uses indentation to indicate a block of code (scope). You can either use tabs or 4 spaces to create an indent.

Let's assume we want to print a student's data, which is the name, age and the language.

```
[2] 1 # Student Data
[2]: 2 print("Name: John Doe")
      3 print("Age: 24")
      4 print("Language: Python")
```

```
[2] 1 Name: John Doe
[2]: 2 Age: 24
      3 Language: Python
```

What if we need to print it again? We have to rewrite all the three print lines again right?

```
[2] 1 # we need student data again
[2]: 2
      3 print("Name: John Doe")
      4 print("Age: 24")
      5 print("Language: Python")
```

```
[2] 1 Name: John Doe
[2]: 2 Age: 24
```

Language: Python

We write exactly the same lines of code just because we need to print student data again. That's not very efficient. We need a better way. What if we had functions which does the printing for us? Whenever we need to print the student data, we can just call these functions. Let's define them:

```
[2
4]: 1 # Define a function for student name
      2
      3 def student_name():
      4     print("Name: John Doe")
```

Now that we defined the `student_name()` function we can call it.

```
[2
5]: 1 # call the function student_name
      2 student_name()
```

```
[2
5]: Name: John Doe
```

We call it and it prints “Name: John Doe” as expected. Now let's define the other functions for the age and language.

Define the function `student_age()` and call it right after the definition.

```
[2
6]: 1 # Define a function for student age
      2
      3 def student_age():
      4     print("Age: 24")
```

```
[2  
7]: 1 # call the function student_age  
2 student_age()
```

```
[2  
7]: Age: 24
```

Define the function `student_language()` and call it right after the definition.

```
[2  
8]: 1 # Define a function for student language  
2  
3 def student_language():  
4     print("Language: Python")
```

```
[2  
9]: 1 # call the function student_language  
2 student_language()
```

```
[2  
9]: Language: Python
```

Now that we have three functions for student name, age and language, whenever we need to print the student data we can call these functions.

```
[3  
0]: 1 # Print Student with functions  
2  
3 student_name()  
4 student_age()  
5 student_language()
```

```
[3  
0]: Name: John Doe  
Age: 24
```

Language: Python

As you see we get all the student data printed. It's not bad, but still not very efficient. What if we need to print the student data one more time? We have to call all three functions again, which makes three lines of code each time. Can't we find a better way?

```
[3
1]: 1 # print student data again
      2 # we have to call all 3 functions again
      3
      4 student_name()
      5 student_age()
      6 student_language()
```

```
[3
1]: Name: John Doe
      Age: 24
      Language: Python
```

The better way is to define a wrapper function (**student_data**) that will call three student functions. Whenever you need student data, you can just call this wrapper function. Let's define it:

```
[3
2]: 1 # one wrapper function to call all three
      2
      3 def student_data():
      4     student_name()
      5     student_age()
      6     student_language()
```

Now that we have a single wrapper function for all the student data, we can call it anytime we want. It's just a single line of code.

```
[3  
3]: 1 # print student data in just one function call  
2 student_data()
```

```
[3  
3]: Name: John Doe  
      Age: 24  
      Language: Python
```

As you see it gives the same result with less code. This is the power of functions and you should use them to write better code. If you need to print the student data one more time, just call the wrapper function again:

```
[3  
4]: 1 # print student data again  
2 student_data()
```

```
[3  
4]: Name: John Doe  
      Age: 24  
      Language: Python
```

So far we print the students first and last name together, like “John Doe”. Now let's assume we want to print the students first and last names separately. Something like “Name: John”, “Lastname: Doe”. To do this we will define two separate functions, one for the first name and the other for the last name:

```
[3  
5]: 1 # create two separate functions for firstname and  
      lastname
```

```
2
3 def student_firstname():
4     print("Name: John")
5
6 def student_lastname():
7     print("Lastname: Doe")
```

Remember, earlier we had `student_name` to print the first and the last names together. Now we will redefine that function. You can always redefine a function in Python. This new `student_name` function will not print anything itself. Instead it will call `student_firstname` and `student_lastname` functions.

```
[3
6]: 1 # redefine student_name function
      # it will print students name by calling two separate
      # functions
3
4 def student_name():
5     student_firstname()
6     student_lastname()
```

Let's call the redefined `student_name` function. You will see it will print the first name and the last names on separate lines.

```
[3
7]: 1 # call the redefined student_name function
      student_name()
```

```
[3
7]: Name: John
      Lastname: Doe
```

And if we call the wrapper `student_data` function again you will see it will print the first and last names on separate lines. That's because it has `student_name` function in it (cell 32), and we redefined that function.

```
[3 8]: 1 student_data()  
  
[3 8]: Name: John  
        Lastname: Doe  
        Age: 24  
        Language: Python
```

You learned how to define functions, how to call them and how to nest them in other functions. Remember that we have nested `student_name` function in `student_data`. Before moving on to the next section I want to show you the execution flow of code. In which order, the Python Interpreter runs your code when you call functions.

Execution Flow:

- Python Interpreter runs the code from the first line
- And moves down
- If it encounters a function call
 - First it goes into that function
 - Executes it
 - Waits it to finish
 - Returns back

- Moves down

Let us draw the execution flow for `student_data` function. The numbers indicate the order of execution by Python Interpreter.

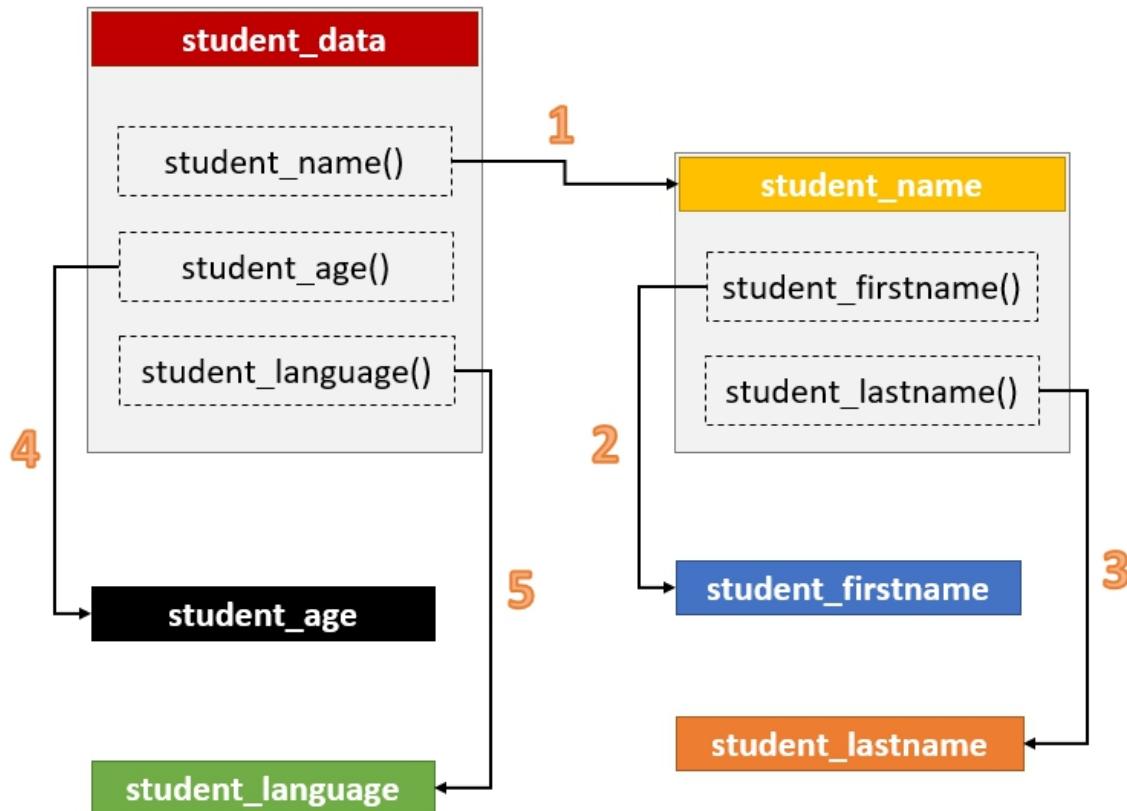


Figure 5-1: Execution flow for `student_data` function

True / False - Defining Functions

For each questions below, decide whether it's true or false.

Q1:

We can use the expression;

print(student_data)

to call **student_data** function and print its result.

- True
- False

Q2:

```
def A():
    print("A")
```

```
def B():
    print("B")
```

```
def C():
    A()
```

```
def D():
    A()
    B()
    C()
```

We have functions as defined here.

If we call function D as **D()**, what will be printed on the screen is:

```
A
B
A
```

- True
- False

SOLUTIONS - True / False - Defining Functions

Here are the solutions for True / False questions.

S1:

We can use the expression;

print(student_data)

to call **student_data** function and print its result.

- True
- False

It's **false**. To call a function, you have to add parentheses just after function name as: **student_data()**.

S2:

```
def A():
    print("A")

def B():
    print("B")

def C():
    A()

def D():
    A()
    B()
    C()
```

We have functions as defined here.

If we call function D as **D()**, what will be printed on the screen is:

A
B
A

- True
- False

It's **true**. **D** will call **A** first, then it will call **B**, then **C**. And **C** will call **A** again.

OceanofPDF.com

Parameters & Arguments

So far, we defined functions with no parameters. Now we will see how to define functions that take one or more parameters. By the way, it's time to talk a little bit about **Parameters** vs. **Arguments**.

In computer programming, the terms parameter and argument can be used for the same thing which is the information that you pass into a function. So they refer to the same thing with a slight difference.

A **Parameter** is the variable listed inside the parentheses in the function definition. An **Argument** is the value that are sent to the function when it is called.

Now it's time to define a function that takes a parameter.

```
[3 9]: 1 # define a function which takes a parameter
      2
      3 def print_square(number):
      4
      5     # get the square
      6     sqr = number**2
      7
      8     # print sqr
      9     print(sqr)
```

In cell 39, we define a function named **print_square** which takes a parameter named **number**. The function uses this parameter to calculate its square and then prints it. So whenever

you call the function you can pass a different number to print its square.

We will call our `print_square` function with different arguments (parameters) but first let's see what happens if we do not pass any arguments to it.

```
[4 0]: 1 # call it without an argument
2 print_square()

[4 0]: TypeError: print_square() missing 1 required
       positional argument: 'number'
```

As you see in the output of cell 40, we get **TypeError**, which says the function `print_square()` needs an argument but we didn't pass any. So if you define your function with parameters then you need to pass a value for them while calling the function. There is solution for this case, but we will see it later.

Now let's pass different arguments into our function and see how it works:

```
[4 1]: 1 print_square(6)

[4 1]: 36
```

In cell 41, we pass the number 6 as the argument to the function and it prints 36. Now let's pass another number.

```
[4 2]: 1 print_square(8)
```

[4
2]: 64

And this time it prints the square of number 8 which is 64. Let's call one more time with another number, 3.

[4
3]: 1 print_square(3)

[4
3]: 9

And not surprisingly, we saw number 9 in the output of cell 43, when we call the function with 3.

Now let's define a function with more than one parameter. The idea is very simple. We just separate the parameters with commas.

Let's say we want to define a function that calculates the area of a rectangle. The parameters will be the length and the width of the rectangle. It will calculate the area and then print it.

[4
4]: 1 *# Example:*
2
3 *# Define a fn that takes two parameters*
4 *# Parameters: short, long*
5 *# Function will print the area of the rectangle*
6
7 **def** area_of_rectangle(short, long):
8
9 *# area -> assign to a variable*
10 area = short * long
11
12 *# print the area*
13 **print**(area)

Now that we have our function let's call it for rectangles with different sizes.

```
[4 5]: 1 # call with two parameters
2 area_of_rectangle(4, 6)
```

```
[4 5]: 24
```

We call the `area_of_rectangle` function with 4 and 6 as the sides and it prints the area as 24. We could also call the same function with variables. Instead of passing directly the values, we could first define two variables for the length and the width and then pass them into the function. You will see exactly the same result:

```
[4 6]: 1 # define two variables first
2 s = 4
3 u = 6
4
5 # pass the variables into to the function
6 area_of_rectangle(s, u)
```

```
[4 6]: 24
```

Now let's do another example. Remember we had `student_data`, `student_firstname`, `student_lastname`, `student_age` and `student_language` in the previous section. But they were all static functions. They could only print static data. We will make them parametric now. They will be able to take any

student information as parameters and print them. Here is the code for this:

```
[4
7]: 1  # Example:
2
3  # Let's make student_data function parametric
4
5  # first name
6  def student_firstname(first):
7      print("Name: " + first)
8
9  # last name
10 def student_lastname(last):
11     print("Lastname: " + last)
12
13 # age
14 def student_age(age):
15     print("Age: " + str(age))
16
17 # language
18 def student_language(lang):
19     print('Language: ' + lang)
20
21 def student_data(firstname, lastname, age,
language):
22     student_firstname(firstname)
23     student_lastname(lastname)
24     student_age(age)
25     student_language(language)
```

Here is what we do in cell 47:

- We redefine `student_firstname` function to take a parameter as: `student_firstname(first)`.
- We redefine `student_lastname` function to take a parameter as: `student_lastname(last)`.
- We redefine `student_age` function to take a parameter as: `student_age(age)`.
- We redefine `student_language` function to take a parameter as: `student_language(lang)`.
- Finally we redefine `student_data` function to take four parameters as: `student_data(firstname, lastname, age, language)`.

Now all the functions are parametric. The `student_data` function will pass the related parameter to any of the functions it calls. Let's now call it with different parameter sets.

[4
8]:

```
1 # define variables to pass as arguments
2 first = 'Clark'
3 last = 'Kent'
4 age = '28'
5 lang = 'Python'
6
7 # call the function
8 student_data(first, last, age, lang)
```

[4
8]:

```
Name: Clark
Lastname: Kent
Age: 28
```

Language: Python

Before calling the `student_data` function with a new parameter set, I want to clarify an important point. If you inspect the `student_age` function carefully you will see that it is a little bit different than the other functions. The print function in it is `print("Age: " + str(age))`. Why do we have `str(age)` not just `age`? The reason is simple but crucial. When the age parameter is integer, you will get an error if you try to concatenate it like `print("Age: " + age)`. Because in Python you cannot concatenate `str` with `int`. Both of the operands must be string when doing addition operation on strings. That's why we converted the age to string via `str(age)`.

We can now call the `student_data` with different arguments including an integer age value.

```
[4
9]: 1 first = 'Peter'
      2 last = 'Parker'
      3 age = 22
      4 lang = 'JavaScript'
      5
      6 student_data(first, last, age, lang)
```

```
[4
9]: Name: Peter
      Lastname: Parker
      Age: 22
      Language: JavaScript
```

True / False - Parameters & Arguments

For each questions below, decide whether it's true or false.

Q1:

You can define a function having **two parameters** as follows:

```
def two_param_function(a b):  
    print(a)  
    print(b)
```

- True
- False

Q2:

In Python, a function **must** return the parameters it received.

- True
- False

SOLUTIONS - True / False - Parameters & Arguments

Here are the solutions for True / False questions.

S1:

You can define a function having **two parameters** as follows:

```
def two_param_function(a b):  
    print(a)  
    print(b)
```

- True
- False

It's **false**. There must be a comma between the parameters.

S2:

In Python, a function **must** return the parameters it received.

- True
- False

It's **false**. A function can return parameters back but this is **not a must**.

Scope

Scope is the region where the variables exist. In other words, a variable will only be visible to and accessible by the code in its scope. Scope is determined by **indentation** in Python.

When we create a function, it has its own scope. The function scope starts right after the colon (:) and includes any line indented in the function. Let's see it with an example:

```
[5
0]: 1 # ----- not function scope -----
2 # ----- not function scope -----
3
4 # function scope example
5 def scope_fn():
6     # + + + function scope starts here + +
7
8     scope_var = 100
9     print(scope_var)
10
11    s2 = scope_var * 2
12    print(s2)
13
14    # + + + function scope ends here + +
15
16    # ----- not function scope -----
17    # ----- not function scope -----
```

In cell 50, you can see the function scope. Lines from 6 to 14 are in the function scope. The lines from 1 to 4 and 15 to 17 are outside the function scope. Now let's call this function:

```
[5 1]: 1 # call the function
2 scope_fn()
```

```
[5 1]: 100
200
```

In cell 50, there are two variables inside the function scope, `scope_var` and `s2`. They exist only inside this function scope. Let's see if we can reach them outside the function.

```
[5 2]: 1 # reach the variable inside the function
2 # try to print scope_var
3 print(scope_var)
```

```
[5 2]: NameError: name 'scope_var' is not defined
```

In cell 52, we get `NameError` when we try to print `scope_var` variable. Why? Because it was defined in the `scope_fn` function and only exists in this functions scope. But the print statement in cell 52 is **not** in the function scope. That's why we cannot reach the `scope_var` variable.

Scoping is very helpful because it avoids name collisions and unpredictable behaviors. For example no one outside the `scope_fn` function can modify `scope_var` variable.

What if you need to use or modify `scope_var` variable? In that case, you have to define this variable in the **global scope**. In general we have two types of scopes:

- **Global Scope:** The variables defined in global scope are available to all of your code.
- **Local Scope:** The variables defined in local scope are only available within that scope. The `scope_var` is only available within the `scope_fn` function.

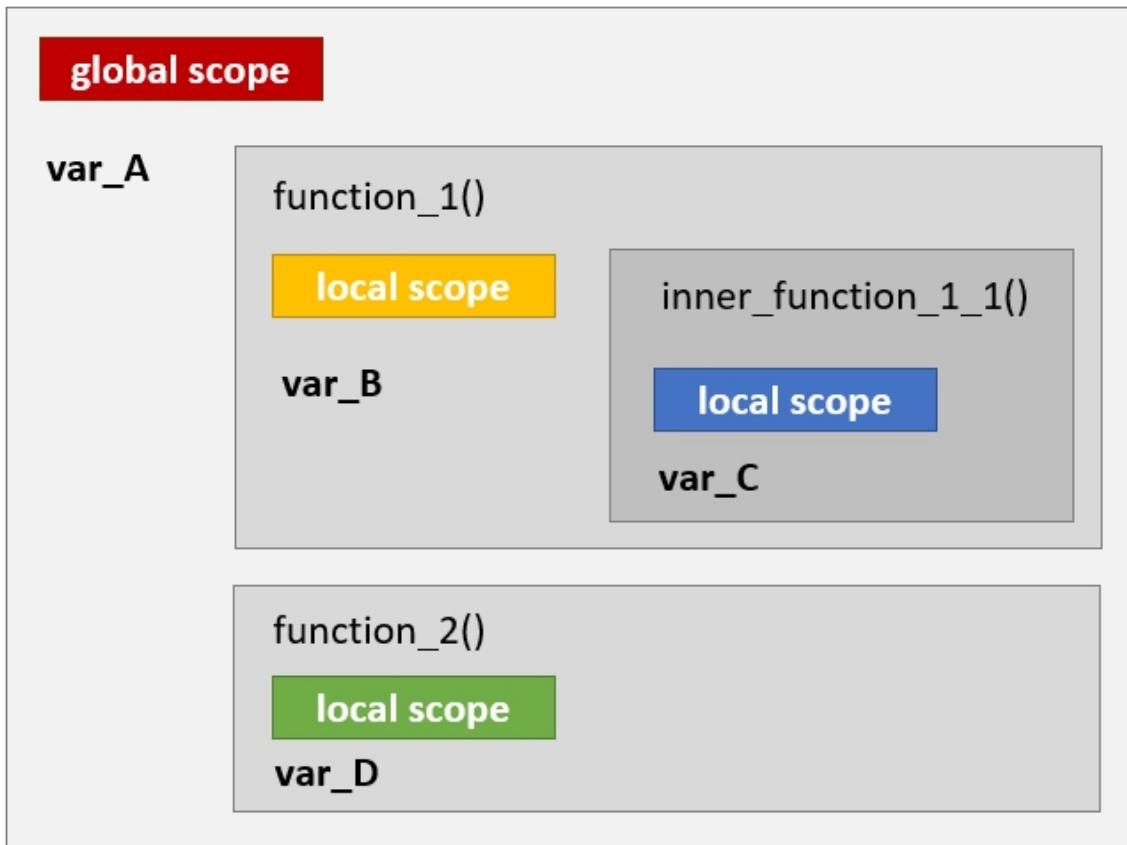


Figure 5-2: Global and local scopes

As a rule of thumb, a function can use any variable which is defined **in or above** of its scope. Let's examine Figure 5-2 to see which variable is visible to which function:

- `var_A` is visible to all functions: `function_1`, `function_1_1` and `function_2`

- **var_B** is visible to **function_1** and **function_1_1**.
- **var_C** is only visible to **function_1_1**.
- **var_D** is only visible to **function_2**.

Let's define two variables in the global scope and then access them from within a function.

```
[5  
3]: 1 # global scope  
2 short = 40  
3 long = 60
```

In cell 53, we defined **short** and **long** variables in the global scope. So we can use them almost anywhere inside this Python file.

```
[5  
4]: 1 # call the variable in global scope  
2 print(short)
```

```
[5  
4]: 40
```

```
[5  
5]: 1 # define a function to use global scope variables  
2  
3 def perimeter():  
4     # reached the global scope  
5     area_of_rect = short * long  
6     print(area_of_rect)
```

In cell 55, we defined a function, **perimeter**, and inside it we used the global variables **short** and **long**. Let's call it and see the result.

```
[5] 1 perimeter()
```

```
[5] 2400
```

We learned that we can access global variables from inside our functions. What if we wanted to modify (reassign) them? Can we do this?

Let's define a function named `change_globals` and see if we can reassign a global variable in it.

```
[5] 1 # try to change the global variables
[7] 2
[3] 3 def change_globals():
[4] 4     short = 50
[5] 5     print(short)
```

In cell 57, we reassigned the variable `short` with a value of 50. And then we print it. Let's call this function and see the result.

```
[5] 1 change_globals()
```

```
[5] 50
```

In the output of cell 58, we can see that the variable `short` has a new value of 50. Now comes the question. What happened to the global variable `short`? Let's see its value, by printing it.

```
[5] 1 # print global short variable again
[9] 2 print(short)
```

[5
9]:

40

As you see in the output of cell 59, the value of the global variable **short** is still **40**. It hasn't changed. Interesting right? We changed its value inside the **change_globals** function. Or "we think" we changed. Actually we didn't change the global variable. When we reassign the variable **short** in line 4 inside the **change_globals** function, Python creates a new variable with the same name. This new variable is just a local duplicate. It only exists inside the **change_globals** function. The new value you assign to it, will not affect the global variable **short**.

How do we change the global variables? The answer is simple. We have to use the **global** keyword with the variable name. Let's see:

```
[6  
0]: 1 # try to change the global variables -> global  
      keyword  
2  
3 def change_globals():  
4     global short  
5     short = 5000  
6     print(short)
```

In line 4 of cell 60, we type **global short** to tell Python Interpreter that, we will be using the actual global variable **short** in this function scope. So whatever modification we do inside the function will affect the global variable. We assigned a new value of **5000** to it inside the function and now let's see if the global

variable also get changed? But first we have to call the function to make it work.

```
[6 1]: 1 change_globals()
```

```
[6 1]: 5000
```

```
[6 2]: 1 # print global short variable again
2 print(short)
```

```
[6 2]: 5000
```

As you see in the output of cell 62, our global variable, `short`, is changed now. That's how we can modify global variables in the inner scopes.

True / False - Scope

For each questions below, decide whether it's true or false.

Q1:

In Python, a function **can only use** the variables which are defined in **its own scope**.

- True
- False

Q2:

In Python, **scope** is determined via the **indentation**.

- True
- False

SOLUTIONS - True / False - Scope

Here are the solutions for True / False questions.

S1:

In Python, a function **can only use** the variables which are defined in **its own scope**.

- True
- False

It's **false**. A function can use the variables in its own scope, moreover can also use the variables of its parent scopes. It can use global scope variables for instance.

S2:

In Python, **scope** is determined via the **indentation**.

- True
- False

It's **true**. Python uses **indentation** for scoping.

Return

A function takes arguments (if any), performs some operations, and returns a value, if it's needed. The value that a function returns to the caller is known as the function's **return value**. This is achieved via a special keyword named **return**. The **return** keyword, exits the function and tells Python to run the rest of the main program. In more simple words; we use the **return** keyword to **exit** a function and **return a value**.

```
[6
3]: 1 # define a fn to return a value
      2
      3 def cube(n):
      4
      5     # calculate the cube
      6     n_cube = n**3
      7
      8     # return this n_cube
      9     return n_cube
```

In cell 63, we define a function, **cube**, with **return** keyword in it. The function takes an argument, **n**, calculates its cube and returns the cube value back to the caller. And as soon as Python executes line 9, it will exit the function.

Now that we what the **return** keyword does, let's call the **cube** function and get the returned value from it.

```
[6
4]: 1 # call the function and get the returned value
```

```
2
3 num = 5
4
5 cube_of_num = cube(num)
6
7 print(cube_of_num)
```

```
[6
4]: 125
```

In cell 64, line 5, we call the `cube` function as `cube(num)` and assign the returned value to the `cube_of_num` variable. Let's do it one more time.

```
[6
5]: 1 # call the function again
2 n = 4
3
4 returned_value = cube(n)
5 print(returned_value)
```

```
[6
5]: 64
```

We could also chain the `cube` and the `print` functions. Remember we learned about composition of functions.

```
[6
6]: 1 # chain the functions
2 print(cube(3))
```

```
[6
6]: 27
```

As you see in cell 66, we chain the `cube` and the `print` functions. And we get the cube of 3, which is 27, in the output.

Functions which do not return any value are generally called as **Void Functions.**

OceanofPDF.com

True / False - Return

For each questions below, decide whether it's true or false.

Q1:

The function below **will return** the square of num parameter back:

```
def get_square(num):  
    square= num**2  
    print(square)
```

- True
- False

Q2:

```
def cube(n):  
    n_cube = n**3  
    return n_cube
```

```
print(cube(3))
```

cube function above is a function which returns a value.

Since we did not assign that returning value to any variable the expression:

```
print(cube(3))
```

will cause **NameError**.

- True
- False

SOLUTIONS - True / False - Return

Here are the solutions for True / False questions.

S1:

The function below **will return** the square of num parameter back:

```
def get_square(num):  
    square= num**2  
    print(square)
```

- True
- False

It's **false**. For a function to return a value back, it must have "**return**" statement at the end.

S2:

```
def cube(n):  
    n_cube = n**3  
    return n_cube  
  
print(cube(3))
```

cube function above is a function which returns a value.

Since we did not assign that returning value to any variable the expression:

```
print(cube(3))  
will cause NameError.
```

- True
- False

It's **false**. We do not have to assign the return value of a function to a variable. **print()** function will print the returning value from the **cube** function.

Docstring

Function **docstrings** are the string literals that appear right after the definition of a function. It tells about the purpose of the function, input and output parameters, return value (if any) and any data about the function. Let's see how we define a function that has a docstring.

```
[6
7]: 1      # define a function with docstring
      2
      3      import math
      4
      5      def get_power(num, p):
      6          """
      7              Calculates the power of number.
      8              Parameters: int num, p
      9              Returns: the give power of the number
      10             """
      11
      12      # calculate the power
      13      power = math.pow(num, p)
      14
      15      # return the power
      16      return power
```

The text enclosed with a pair of triple quotes (""""") is the docstring. It is helpful for documentation and code readability. Anyone who wants to use this function can check this docstring to learn about it. Let's see how we can get the docstring of a function.

```
[6 8]: 1 # get help for this function
        2 # help()
        3
        4 help(get_power)
```

```
[6 8]: Help on function get_power in module __main__:
```

```
get_power(num, p)
    Calculates the power of number.
    Parameters: int num, p
    Returns: the give power of the number
```

We call the `help()` function to get the docstring of any function in Python. As you see in the output of cell 68, it returns the docstring we write when we define the `get_power` function.

If you need more information about a function, you can use question mark (?) after the function name:

```
[6 9]: 1 # detail help
        2 # ?
        3
        4 get_power?
```

```
[6 9]: Signature: get_power(num, p)
```

Docstring:

Calculates the power of number.

Parameters: int num, p

Returns: the give power of the number

File: c:\ebook\python\contents\5_functions_i\ref\
<ipython-input-67-a3fa>

Type: function

You see the detailed information in the output of cell 69. Here are Signature, Docstring, File and Type data about the function.

Signature: Signature is how we call the function:
function_name(parameter_1, parameter_2, ...)

If you want even more detailed information about a function you can use double question marks (??) after the function name.

```
[7  
0]: 1 # most detailed help  
      2 # ??  
      3  
      4 get_power??
```

[7
0]:

Signature: get_power(num, p)

Source:

```
def get_power(num, p):  
    """
```

Calculates the power of number.

Parameters: int num, p

Returns: the give power of the number

"""

```
    # calculate the power  
    power = math.pow(num, p)  
    # return the power  
    return power
```

File: c:\ebook\python\contents\5_functions_i\ref\
<ipython-input-67-a3fa>

Type: function

As you see in the output of cell 70, we can even get the source code of a function with double question marks (??).

Built-in Methods and Attributes:

Everything in Python is an object, and almost every object has built-in methods and attributes. They are called **dunder** methods (or attributes). We can get them via shortcuts. The word “**dunder**” simply means “**double underscores: __**”. Functions are objects too, so they have built-in attributes. Let’s print some of the dunder attributes for our `get_power` function.

```
[7 1]: 1 # Docstring -> .__doc__  
2 get_power.__doc__
```

```
[7 1]: '\n      Calculates the power of  
      number.\n      Parameters: int num,  
      p\n      Returns: the give power of the number\n      '
```

As you see in cell 71, `__doc__` is used to get the docstring of a function as: `get_power.__doc__`

Let’s print it to see it as formatted text:

```
[7 2]: 1 print(get_power.__doc__)
```

```
[7 2]:      Calculates the power of number.  
      Parameters: int num, p  
      Returns: the give power of the number
```

We can also get the module in which our `get_power` function runs. It is another dunder method: `__module__`

```
[7 3]: 1 # module
```

```
2
3 # .__<TAB>
4 get_power.__module__
```

[7]
3: '__main__'

`__main__` Module is the default module when you launch a Python file. It is the name of the scope in which top-level code executes, the global scope in our case.

As a final exercise, let's get the name of our function via dunder attributes: `__name__`

```
[7]
4: 1 # name
  2
  3 get_power.__name__
```

[7]
4: 'get_power'

How to get input from the user in JupyterLab?

Before moving on to the True / False questions and the Quiz for this chapter, there is one last function we should learn. It is the built-in `input()` function in Python. The `input()` function is used to get any input from the user. And the user input will always be in type of **string**. Keep this in mind when using the `input()` function. Now let's see how to get input from the user:

```
[*]: # ask for the user name  
  
      input("Please enter your name:")
```

Please enter your name:

```
[ ]:
```

Figure 5-3: The `input()` function in JupyterLab.

When you run the cell with the `input()` function, JupyterLab will display a text box for the user input. And the kernel will stay busy until the user presses **Enter**. After pressing the Enter key, Python will execute the next line after the `input()` function. As you see in cell 75 below:

```
[7  
5]: 1 # ask for the user name  
2  
3 input("Please enter your name:")
```

```
[7  
5]: Please enter your name: Musa Arda  
'Musa Arda'
```

Since the `input()` function returns the user input, you can assign it to a variable, and use it whenever you need. See the code in cell 76, line 3. We assign the user input to a variable named `user_name`.

```
[7  
6]: 1 # ask for the user name and assign to variable  
2  
3 user_name = input("Please enter your name:")  
4 print("The user name is: " + user_name)
```

[7
6]:

Please enter your name: Musa Arda

The user name is: Musa Arda

OceanofPDF.com

True / False - Docstring

For each questions below, decide whether it's true or false.

Q1:

In Python, the docstring of a function is declared above the function name:

```
"""
```

This function calculates the power.

Parameters: int num, int p

Returns: the power p of num

```
"""
```

```
def get_power(num, p):  
    power = math.pow(num, p)  
    return power
```

- True
- False

Q2:

To see the docstring of a function, we can use the **dunder** attribute `"_doc_"` as follows:

```
<function_name>._doc_
```

- True
- False

SOLUTIONS - True / False - Docstring

Here are the solutions for True / False questions.

S1:

In Python, the docstring of a function is declared above the function name:

```
"""
This function calculates the power.
Parameters: int num, int p
Returns: the power p of num
"""

def get_power(num, p):
    power = math.pow(num, p)
    return power
```

- True
- False

It's **false**. Docstring must be inside the function scope, not above of it.

S2:

To see the docstring of a function, we can use the **dunder** attribute `"_doc_"` as follows:

```
<function_name>.__doc__
```

- True
- False

It's **false**. **Dunder**'s are used with two underscores (`__`), not one.

QUIZ - Functions I

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_Functions_I.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *5_Functions_I*. Here are the questions for this chapter:

QUIZ - Functions I:

Q1:

Define a function named **first_function** that prints "Hello World!".

```
[  
1  1 # Q 1:  
]:  
2  
3 # --- your solution here --- #  
4  
5 # call the function you defined  
6 first_function()
```

```
[  
1  Hello World!  
]:
```

Q2:

Define a function named **say_hello** that takes two string parameters.

We will assume this parameters be two separate names.
(name_1 and name_2).

Function will concatenate these names with "and".

Then it will print as:

"Hello name_1 and name_2."

Expected Output: Hello Clark Kent and Superman.

```
[  
2 1 # Q 2:  
]:  
2  
3 # --- your solution here --- #  
4  
5 # call the function  
6 say_hello("Clark Kent", "Superman")  
7 say_hello("Bruce Wayne", "Batman")  
8 say_hello("Peter Parker", "Spiderman")
```

```
[  
2 Hello Clark Kent and Superman  
]:  
Hello Bruce Wayne and Batman  
Hello Peter Parker and Spiderman
```

Q3:

Define a function named **say_my_name** that asks user to enter his/her name.

This function will print as: "Hi [user_name]."

Hint: to get input from user -> **input()**

```
[  
3 1 # Q 3:  
]:  
2  
3 # --- your solution here --- #  
4  
5 # call the function you defined  
6 say_my_name()
```

```
[  
3 Please enter your name: musa arda  
]:  
Hi musa arda
```

Q4:

Define a function named `get_max` that takes 3 parameters as integer.

Function will return the maximum of these three numbers.

Add a docstring for this function.

Hint: Python has `max()` built-in function.

You can use `help(max)` to see how to use the `max()` function.
Or you can find the documentation [here](#).

```
[4]: 1      # Q 4:  
      :  
      2  
      3      # --- your solution here --- #  
      4  
      5      # call function -> get the returned value and print  
      6      max_number = get_max(4, 82, 19)  
      7      print(max_number)  
      8  
      9      # docstring  
      10     help(get_max)  
      11     print(get_max.__doc__)
```

```
[4]: 82  
      :  
      Help on function get_max in module __main__:  
      get_max(num_1, num_2, num_3)  
          This function gets 3 numbers and selects the  
          maximum.  
          Parameters: int num_1, num_2, num_3  
          Returns: Value from Python's max function  
  
          This function gets 3 numbers and selects the  
          maximum.  
          Parameters: int num_1, num_2, num_3  
          Returns: Value from Python's max function
```

Q5:

Define a function named **strip_and_lower** that takes a string parameter.

Function will remove spaces at the beginning and at the end of the string, then it will convert the string into lower case.

Hints:

- *to remove spaces: strip()*
- *to convert to lowercase: lower()*

```
[  
5 1 # Q 5:  
]:  
2  
3 # --- your solution here --- #  
4  
5 # function call  
6 txt = " PyThon MaCHine LeARning "  
7 final_txt = strip_and_lower(txt)  
8 print(final_txt)
```

```
[  
5 python machine learning  
]:
```

Q6:

Define a function named **add** which takes two integer parameters.

The function will return the sum of these parameters.

```
[  
6 1 # Q 6:  
]:  
2  
3 # --- your solution here --- #  
4  
5 # function call  
6 n_1 = 45
```

```
7 n_2 = 20
8 addition = add(n_1, n_2)
9 print(addition)
```

```
[6]: 65
```

Q7:

Define a function named **minimum_difference** which is taking 3 numeric parameters.
Function will calculate the difference between each pairs.
And return the minimum of these differences.

Hints:

- Use absolute value to calculate differences -> `abs()`
- To get minimum -> `min()`

```
[7]: 1 # Q 7:
2
3 # --- your solution here --- #
4
5 # function call
6 a = 8
7 b = 5
8 c = 10
9 min_diff = minimum_difference(a, b, c)
10 print(min_diff)
```

```
[7]: 2
```

Q8:

Define a function named **square_root** which is taking an integer parameter.

Function will return the square root of this parameter.

Hint: Square Root -> math.sqrt()

```
[  
8 1 # Q 8:  
]:  
2  
3 import math  
4  
5 # --- your solution here --- #  
6  
7 # function call  
8 num = 81  
9 sqr = square_root(num)  
10 print(sqr)
```

```
[  
8 9.0  
]:
```

Q9:

Define a function named **get_logarithm** which takes a numeric parameter.

Function will return the logarithm of this number.

Hint: logarithm -> math.log()

```
[  
9 1 # Q 9:  
]:  
2  
3 import math  
4  
5 # --- your solution here --- #  
6  
7 # function call  
8 n = 12
```

```
9     log = get_logarithm(n)
10    print(log)
11
12    # chained function call
13    print(get_logarithm(n))
```

```
[9]: 2.4849066497880004
]: 2.4849066497880004
```

Q10:

Define a function named **hypotenuse** which takes two numeric parameters.

Let's assume these numbers are two sides of a right triangle. The function will return the hypotenuse of this triangle.

Hint:

$$\text{hipo}^2 = a^2 + b^2$$

```
[1]: 1    # Q 10:
0: 2
3    import math
4
5    # --- your solution here --- #
6
7    # function call
8    # 3-4-5 triangle
9    print(hypotenuse(3, 4))
10
11   # 7-24-25 triangle
12   print(hypotenuse(7, 24))
13
14   # 8-15-17 triangle
15   print(hypotenuse(8, 15))
```

[1
0]:

5.0
25.0
17.0

OceanofPDF.com

SOLUTIONS - Functions I

Here are the solutions for the quiz for Chapter 5 - Functions I.

SOLUTIONS - Functions I:

S1:

```
[  
1 1 # S 1:  
]:  
2  
3 def first_function():  
4     print("Hello World!")  
5  
6 # call the function you defined  
7 first_function()
```

```
[  
1 Hello World!  
]:
```

S2:

```
[  
2 1 # S 2:  
]:  
2  
3 def say_hello(name_1, name_2):  
4     concat = "Hello " + name_1 + " and " +  
5     name_2  
6     print(concat)  
7  
8 # call the function you defined  
9 say_hello("Clark Kent", "Superman")  
10 say_hello("Bruce Wayne", "Batman")  
11 say_hello("Peter Parker", "Spiderman")
```

```
[  
2  
]:
```

```
Hello Clark Kent and Superman
```

```
Hello Bruce Wayne and Batman  
Hello Peter Parker and Spiderman
```

S3:

```
[  
3  
]:
```

```
1      # S 3:  
2  
3      def say_my_name():  
4  
5          # input() -> waits for enter  
6          name = input("Please enter your name: ")  
7  
8          # continues after pressing enter  
9          print("Hi {0}".format(name))  
10  
11     # call the function you defined  
12     say_my_name()
```

```
[  
3  
]:
```

```
Please enter your name: musa arda
```

```
Hi musa arda
```

S4:

```
[  
4  
]:
```

```
1      # S 4:  
2  
3      def get_max(num_1, num_2, num_3):  
4          """  
5              This function gets 3 numbers and selects the  
6              maximum.  
7              Parameters: int num_1, num_2, num_3
```

```
7     Returns: value from Python's max() function
8     """
9     # get the maximum -> max()
10    maximum = max(num_1, num_2, num_3)
11    # return the maximum
12    return maximum
13
14 # call function -> get the returned value and print
15 max_number = get_max(4, 82, 19)
16 print(max_number)
17
18 # docstring
19 help(get_max)
20 print(get_max.__doc__)
```

```
[4]:
```

```
82
```

```
Help on function get_max in module __main__:
```

```
get_max(num_1, num_2, num_3)
```

```
    This function gets 3 numbers and selects the
    maximum.
```

```
    Parameters: int num_1, num_2, num_3
```

```
    Returns: Value from Python's max function
```

```
    This function gets 3 numbers and selects the
    maximum.
```

```
    Parameters: int num_1, num_2, num_3
```

```
    Returns: Value from Python's max function
```

```
S5:
```

```
[5]:
```

```
1     # S 5:
2
3     def strip_and_lower(text):
```

```

4      # strip() -> removes spaces from beginning and
5      # end
6      removed = text.strip()
7      # lower() -> converts text into lower case
8      lower_case = removed.lower()
9      # return the final string
10     return lower_case
11
12     # function call
13     txt = "  PyThon MaCHine LeARning  "
14     final_txt = strip_and_lower(txt)
15     print(final_txt)

```

```

[5]: python machine learning
]:
```

S6:

```

[6]: 1      # S 6:
      :
2
3      def add(n_1, n_2):
4          """
5              It adds two numbers.
6              Parameters: int n_1, n_2
7              Returns: sum of numbers
8          """
9          addition = n_1 + n_2
10         return addition
11
12     # function call
13     n_1 = 45
14     n_2 = 20
15     addition = add(n_1, n_2)
16     print(addition)

```

```
[  
6      65  
]:
```

S7:

```
[  
7  1  # S 7:  
]:  
2  
3  def minimum_difference(a, b, c):  
4      """  
5          It calculates the differences of each pair.  
6          Parameters: int a, b, c  
7          Returns: minimum of pair differences  
8      """  
9      # pair differences  
10     diff_1 = abs(a - b)  
11     diff_2 = abs(a - c)  
12     diff_3 = abs(b - c)  
13     # return the minimum difference  
14     return min(diff_1, diff_2, diff_3)  
15  
16  # function call  
17  a = 8  
18  b = 5  
19  c = 10  
20  min_diff = minimum_difference(a, b, c)  
21  print(min_diff)
```

```
[  
7      2  
]:
```

S8:

```
[ 1  # S 8:  
8
```

]:

```
2
3 import math
4
5 def square_root(n):
6     """
7         Computes the square root.
8         Parameter: int n
9         Returns: square root
10    """
11    # compute square root
12    sqr = math.sqrt(n)
13    # return square root
14    return sqr
15
16 # function call
17 num = 81
18 sqr = square_root(num)
19 print(sqr)
```

[
8
]:

9.0

S9:

[
9
]:

```
1 # S 9:
2
3 import math
4
5 def get_logarithm(n):
6     return math.log(n)
7
8 # function call
```

```
9     n = 12
10    log = get_logarithm(n)
11    print(log)
12
13    # chained function call
14    print(get_logarithm(n))
```

```
[9]: 2.4849066497880004
]: 2.4849066497880004
```

S10:

```
[1
0]: 1    # S 10:
2
3    import math
4
5    def hypotenuse(a, b):
6        return math.sqrt(a**2 + b**2)
7
8    # function call
9    # 3-4-5 triangle
10   print(hypotenuse(3, 4))
11
12   # 7-24-25 triangle
13   print(hypotenuse(7, 24))
14
15   # 8-15-17 triangle
16   print(hypotenuse(8, 15))
```

```
[1
0]: 5.0
     25.0
     17.0
```

OceanofPDF.com

6. Project 1 – Functions

This is our first project in this book. It will cover the topics that we have seen so far. Mainly it's going to be on functions. We will see how we define and call the functions. We will be using The Turtle Module in Python.

You can find the source code of this project in the [Github repository](#) of the book. It is in the folder named *6_Project_I_Functions*.

The Turtle Module

The [Turtle](#) Module (`turtle` for short) is a Python library which is used to create graphics, pictures and games^[6]. It is mainly used for educational purposes. It was developed by Wally Feurzeig, Seymour Parpet and Cynthina Slolomon in 1967. Turtle is a pre-installed library in Python that is similar to the virtual canvas that we can draw pictures and attractive shapes. It provides the onscreen pen that we can use for drawing.

The first thing you should learn is how to make the `turtle` move in the direction you want it to go.

Moving the Turtle

There are four directions that a turtle can move in:

- Forward: **forward()** or **fd()**
- Backward: **backward()** or **bk()**
- Left: **left()** or **lt()**
- Right: **right()** or **rt()**

The **turtle** moves **.forward()** or **.backward()** in the direction that it's facing. You can change this direction by turning it **.left()** or **.right()** by a certain degree. For example to move the turtle forward we type **turtle.fd(distance)** where the distance parameter is a number (integer or float) in pixels.

To be able to use the turtle module we have to import it. Let's import it and see basic movements:

```
[  
1 1 # import the module -> turtle  
]:  
2 import turtle
```

When we import the module, we get access to any objects and functions in it. We are interested in a special function, a **constructor** for the turtle module. We can access it via: **turtle.Turtle()**. We will assign it to a local variable. This is an **instance** of the turtle object. When you launch the Turtle, you will see a screen like the one in *Figure 6-1*:

```
[  
2 1 # get the turtle object and assign it to a local  
]:  
2 t = turtle.Turtle()
```

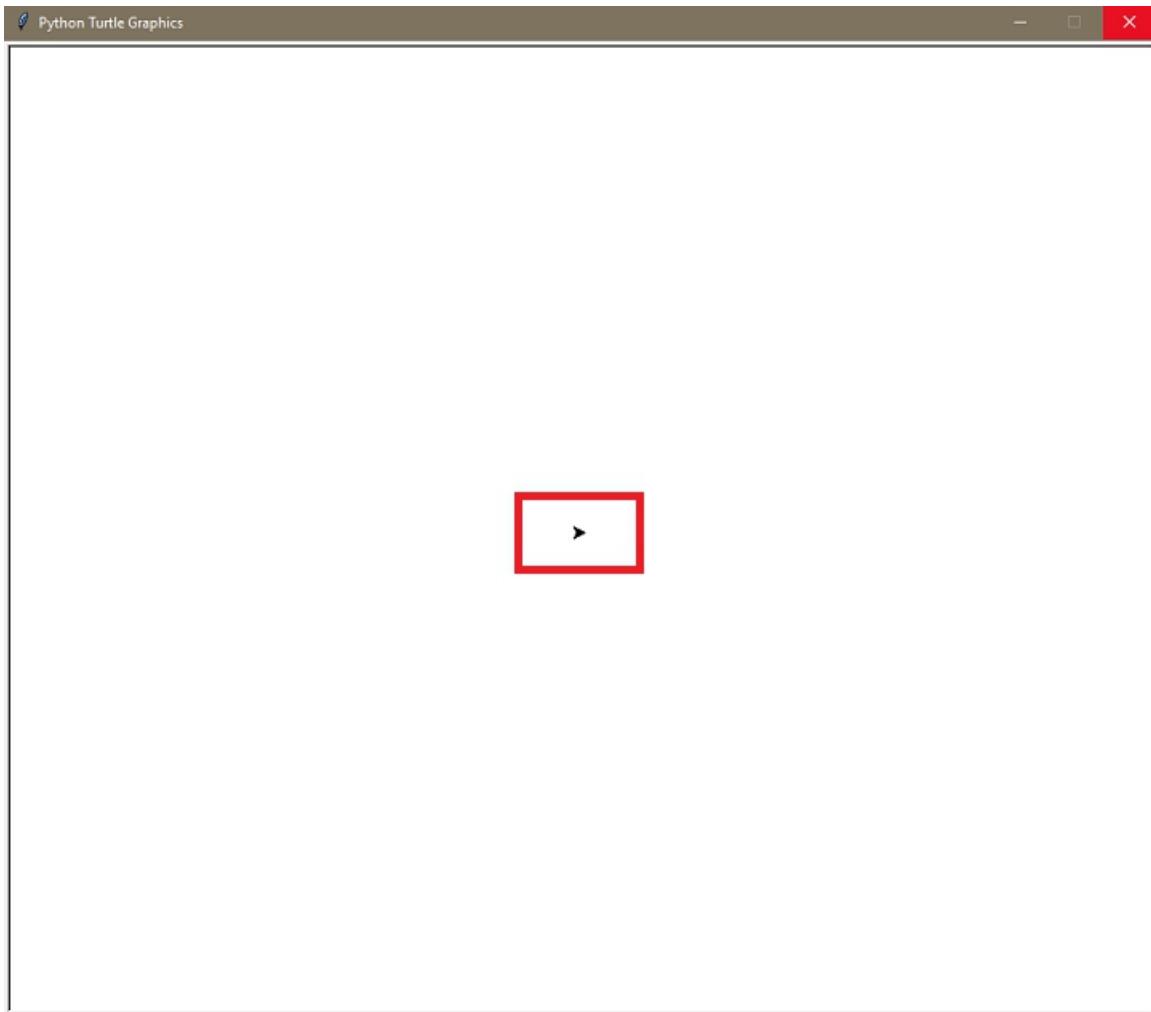


Figure 6-1: The turtle screen started

Important Note: Do not close the turtle window. If you close it, make sure you **restart the kernel** to start from the beginning. Otherwise you might get errors when you run the same code. If you need to close the turtle screen, first run the **.mainloop()** function, then close it. You should call the **.mainloop()** function as: **turtle.mainloop()**. The **.mainloop()** must be the last statement in a turtle graphics program.

Keep the turtle screen open (*Figure 6-1*) and run the code cell below. You will see the turtle moves 100 pixels to the right, the forward direction:

```
[  
3  1 # move turtle forward by 100 pixels  
]:  
2  t.fd(100)
```

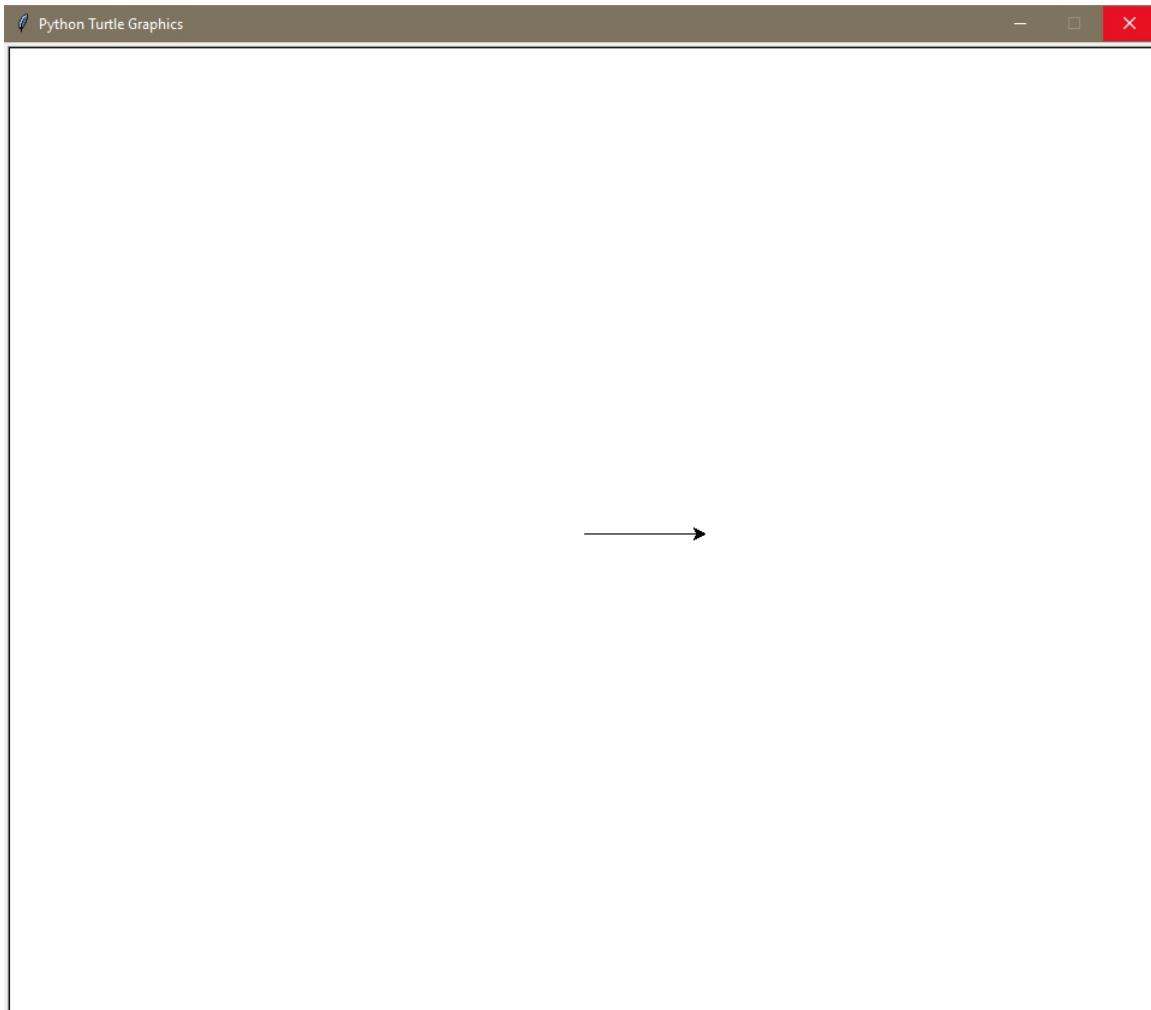


Figure 6-2: Turtle moves forward by 100 pixels

Now let's turn it to the left by 90 degrees. If you run the code in cell 4 below, you will see the turtle changes its direction

upwards:

```
[  
4  1 # turn the turtle to the left by 90 degrees  
]:  
2 t.lt(90)
```

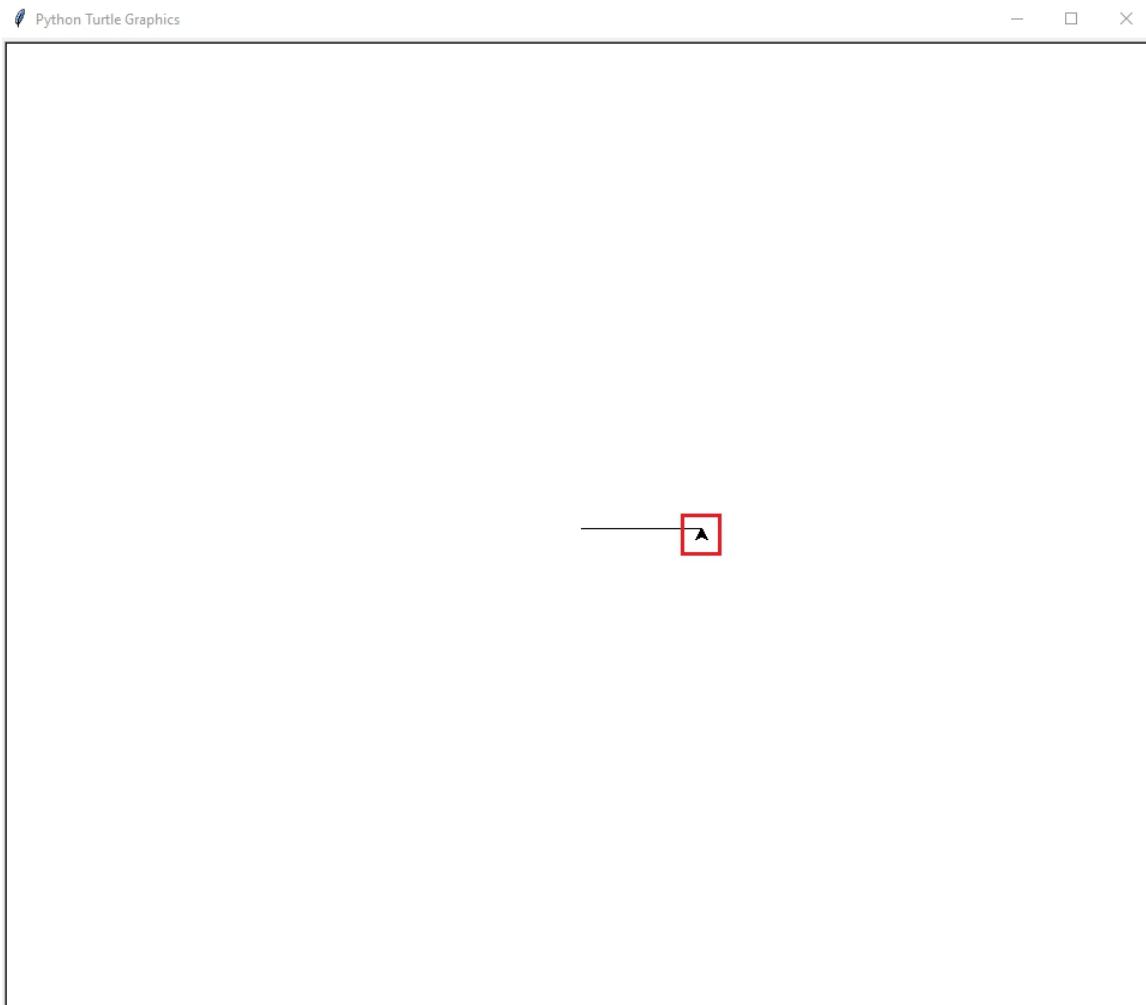


Figure 6-3: The turtle turns upwards

Now if you run the cells from 5 to 10, the turtle will draw a square, by moving forward (`fd`) and turning left (`lt`) 4 times each. It will end up at the same point and in the same direction where it started:

```
[5]: 1 # move turtle forward by 100 pixels
]: 2 t.fd(100)
```

```
[6]: 1 # turn the turtle to the left by 90 degrees
]: 2 t.lt(90)
```

```
[7]: 1 # move turtle forward by 100 pixels
]: 2 t.fd(100)
```

```
[8]: 1 # turn the turtle to the left by 90 degrees
]: 2 t.lt(90)
```

```
[9]: 1 # move turtle forward by 100 pixels
]: 2 t.fd(100)
```

```
[10]: 1 # turn the turtle to the left by 90 degrees
      2 t.lt(90)
```

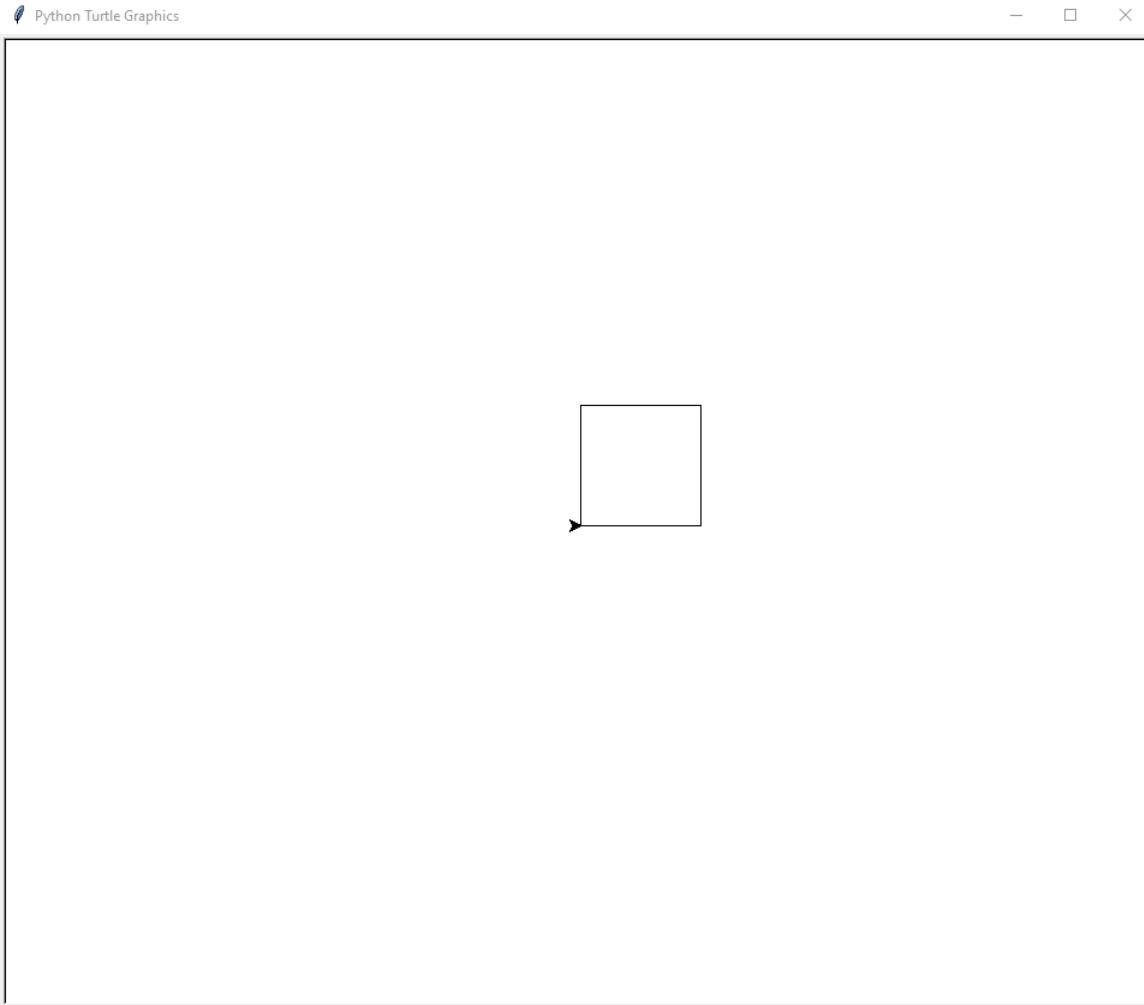


Figure 6-4: A full square with turtle

If we want to clear the screen and move the turtle to its initial position, all we have to do is to call the `clear()` function.

```
[1]: 1 # clear screen and reset the turtle to initial position
      2 t.clear()
```

We can also call the `resetscreen()` function to reset the screen. But this function is called directly with the module itself. Remember we called the `clear()` function with the instance `t`, we created in cell 2.

```
[1] 1 # reset the screen
[2]: 2 turtle.resetscreen()
```

Now that we know how basically the turtle moves, let's draw some interesting shapes with it.

OceanofPDF.com

Square

To create a square with the turtle, we have to follow the **fd-lt** pattern 4 times. Let's write all the steps one by one first:

```
[1
3]: 1      # We want to create a square
      2
      3      # Move 100 pixel forward
      4      # Turn left 90 deg
      5
      6      # Move 100 pixel forward
      7      # Turn left 90 deg
      8
      9      # Move 100 pixel forward
      10     # Turn left 90 deg
      11
      12     # Move 100 pixel forward
      13     # Turn left 90 deg
      14
      15     t.fd(100)
      16     t.lt(90)
      17
      18     t.fd(100)
      19     t.lt(90)
      20
      21     t.fd(100)
      22     t.lt(90)
      23
      24     t.fd(100)
      25     t.lt(90)
```

With the code in cell 13, we can draw a perfect square. But there is an issue here. We repeated exactly the same lines of code 4 times. Repetition is something you should always avoid in programming. Actually there is a **loop** here. We haven't covered the **Loops** yet, but here we can see a little bit of them. Chapter 10 is dedicated to Loops, but we can write a simple **for** loop here. The **for** loop will repeat how many times we want to, 4 in our case. And it will repeat the **fd-lt** pattern.

Before the **for** loop, let's reset the screen. Then we can run our loop.

```
[1] 1 turtle.resetscreen()
[4]: 1
      # with loop
      # it's going to repeat fd-lt patter 4 times
      3
      4 for i in range(4):
      5     t.fd(100)
      6     t.lt(90)
```

Now let's use this for loop inside a function. We will create a function named **square**. The parameters will be:

- a turtle object, **t**
- int **d** (distance), in pixels
- bool **is_screen_reset**, to check if the caller wants a screen reset

```
[1]: 1 # square function
      2
      3 def square(t, d, is_screen_reset):
      4     """
      5         It draws square with turtle.
      6         Parameters: turtle t, int d, bool
      7         is_screen_reset
      8         Returns: None
      9         """
     10
     11     # reset screen
     12     if is_screen_reset == True:
     13         reset()
     14
     15     # draw a square
     16     for i in range(4):
     17         t.fd(d)
                     t.lt(90)
```

In cell 16, line 11, you see an `if` statement. It is a conditional statement to assess if some conditions is true or not. If it's true, then it will run the code in the `if` block. Don't worry, you will learn all about conditionals in Chapter 8. We used it here, to check a parameter value. If the `is_screen_reset` parameter is `True`, then it will call the `reset()` function. If not, it will not call it.

Let's also define the `reset()` function. Instead of calling the `turtle.resetScreen()` anytime we need to reset screen, we create a function for this task. We will just call the function whenever we need the screen to be reset. The `reset()` function has no parameters:

```
[1 7]: 1 # reset function
2
3 def reset():
4     turtle.resetscreen()
```

Now that we defined it, let's call the `square()` function to first reset the screen and draw a square of size 200 pixels:

```
[1 8]: 1 square(t, 200, True)
```

And here we go. We call the `square` function with the parameters `t` (the turtle object we created before), `200` pixels for distance, and `True` to reset the screen: `square(t, 200, True)`. And it draws a perfect square.

Let's say we want to draw a bigger square. Just change the distance parameter (`d`) and call the function again:

```
[1 9]: 1 square(t, 300, True)
```

Now let's say we don't want to reset the screen. We want the old square to stay, but we will draw a new one on to it. The size will be 100 pixels for the new one:

```
[2 0]: 1 square(t, 100, False)
```

If we run the code in cell 20, we will see two squares on the turtle screen.

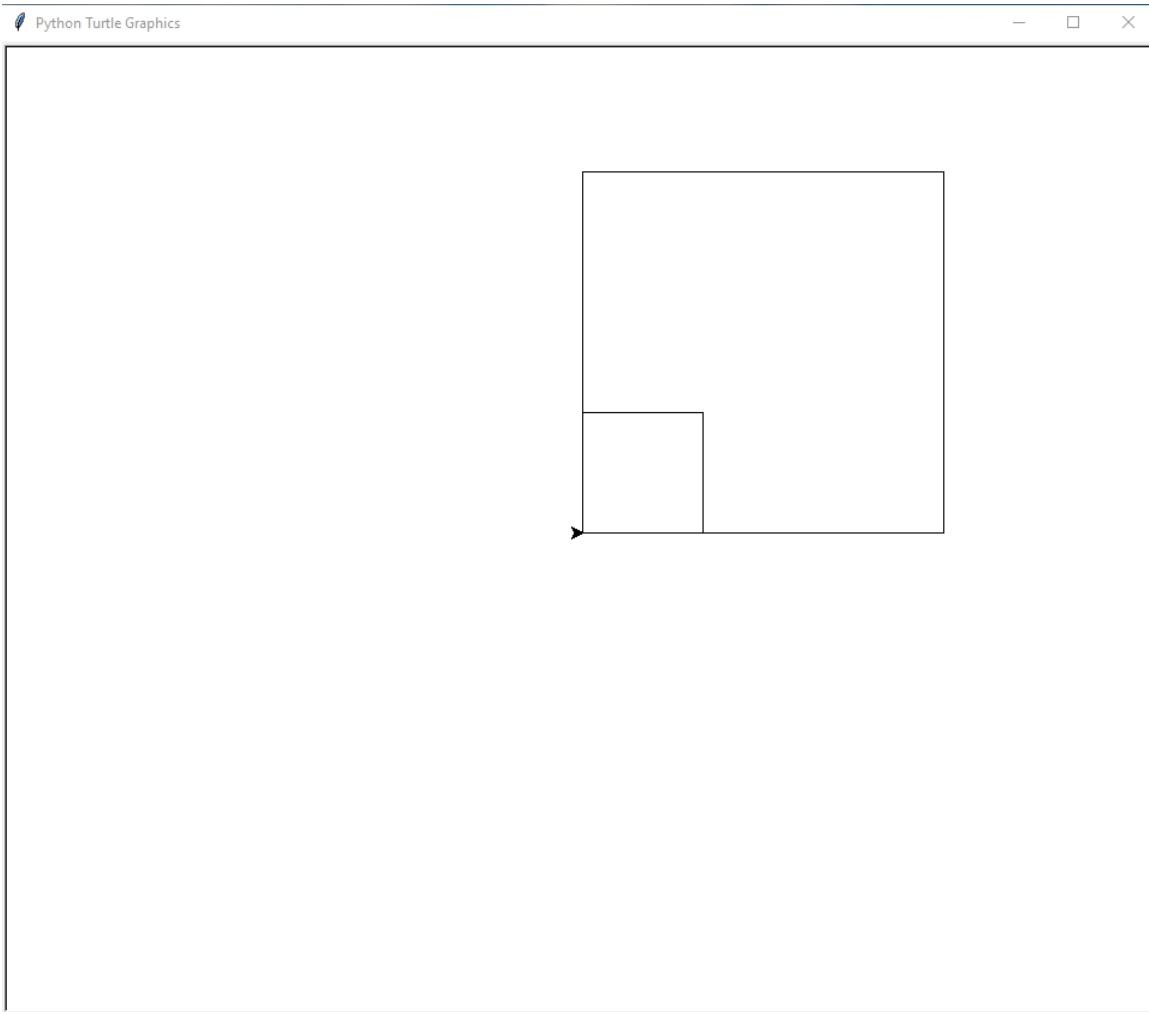


Figure 6-5: Two squares with sizes 300 and 100 pixels

Let's make it more interesting. Let's say, we want to draw 10 squares with varying sizes. The first one will be of 20 pixels and it will increase by 20 up to 200. The last one will be in 200 pixels size. As you may guess, we will use a **for** loop to call our **square()** function. Let's see the code first:

```
[2] 1 # draw 10 squares of size: 20 - 40 - 60 80 - ... - 200
 2
 3 # first reset the screen just once
```

```
4 reset()
5
6 # loop to draw the circles
7 for i in range(1, 11):
8     square(t, i * 20, False)
```

If we run cell 21, we will get a beautiful shape of 10 squares with different sizes. Remember we talked about the `range()` function in Chapter 4. It is used to create a range from start to end index. We used it as `range(1, 11)` to get a range of 10 numbers, 1 to 10. The end index, 11, is not included.

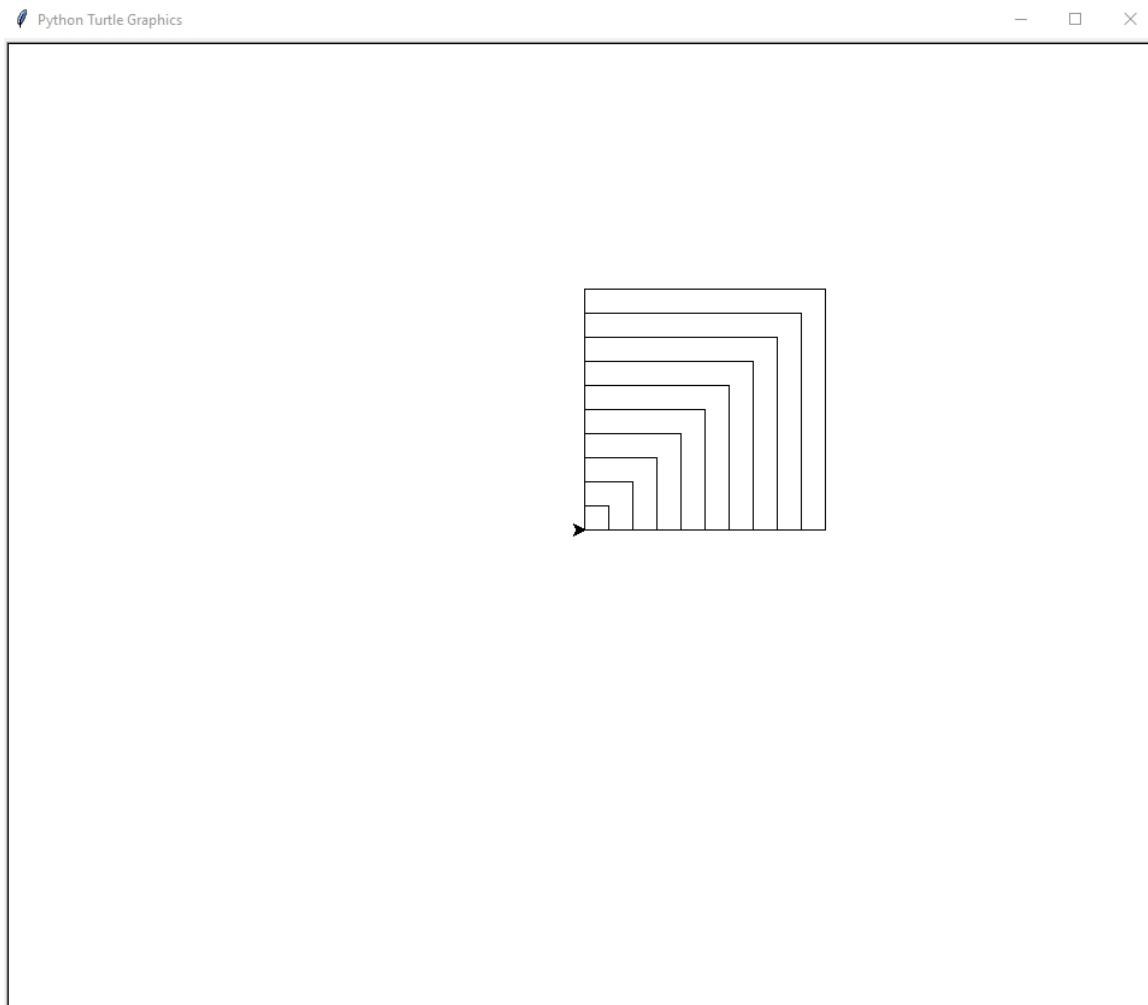


Figure 6-6: 10 squares with sizes 20 to 200.

OceanofPDF.com

Polygon

We will draw a polygon now. The difference between a square and a polygon is that the polygon may have varying number of sides. It doesn't have to be 4 as in the square. And we will take the number of sides as a parameter for our `polygon()` function. The parameters will be:

- a turtle object, `t`
- int `d` (distance), in pixels
- int `n`, number of sides
- bool `is_screen_reset`, to check if the caller wants a screen reset

[2]
2]:

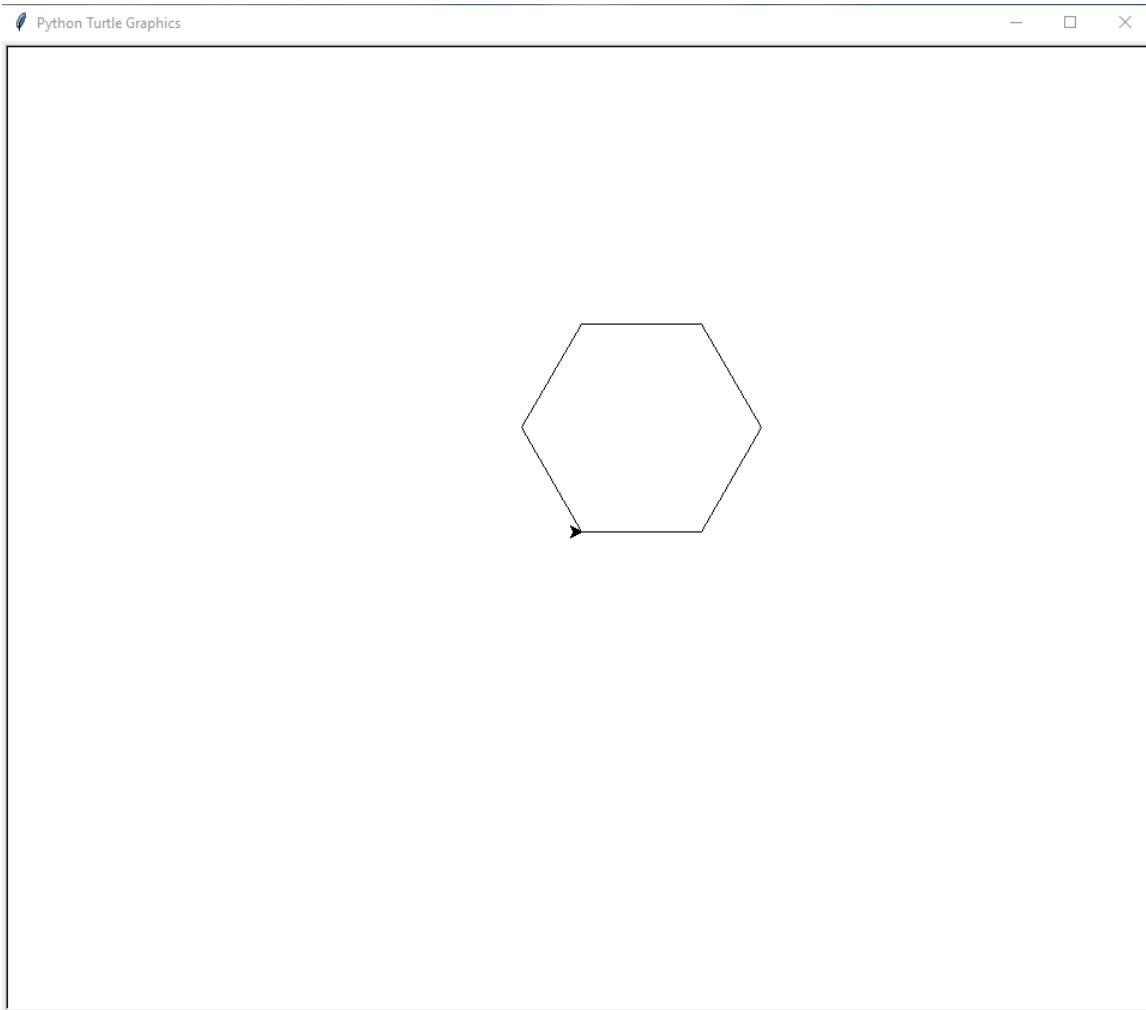
```
1  # polygon function
2
3  def polygon(t, d, n, is_screen_reset):
4      """
5          Draws a polygon.
6          Parameters: turtle t, int d (pixels), int n
7          (number of sides),
8          bool is_screen_reset
9          """
10
11     # resetting
12     if is_screen_reset:
13         reset()
```

```
15      # angle
16      angle = 360 / n
17
18      # polygon
19      for i in range(n):
20          t.fd(d)
21          t.lt(angle)
```

Since the number of sides, `n`, is a parameter now, we have to find a way to calculate the angle to turn left. We cannot use `lt(90)` this time. The `angle` should be `360 / n` degrees for a polygon. First we calculate it in line 16 of cell 22. The rest is almost the same as the square function. We loop over `n` sides and turn left by `angle` degrees.

Let's call our `polygon()` function to draw a six-sided shape, which is called hexagon.

```
[2
3]: 1 # draw a 6 sided hexagon
      2 polygon(t, 100, 6, True)
```



*Figure 6-7: A six-sided shape, hexagon, drawn by the polygon()
function*

Named Arguments:

When you call a function you may want to pass the arguments with their names. This is a good practice which improves code readability. Arguments which you pass with their names are called **named arguments (parameters)**. We can pass the named arguments like: **function_name(parameter_name) =**

parameter_value). Let's call the **polygon()** function with named arguments this time:

```
[2 4]: 1 # call the polygon fn with named arguments
2 polygon(t=t, d=120, n=8, is_screen_reset=False)
```

As you see in cell 24, we call the **polygon()** function, with the named arguments. Now we know which value goes to which parameter. And here is the resulting screen:

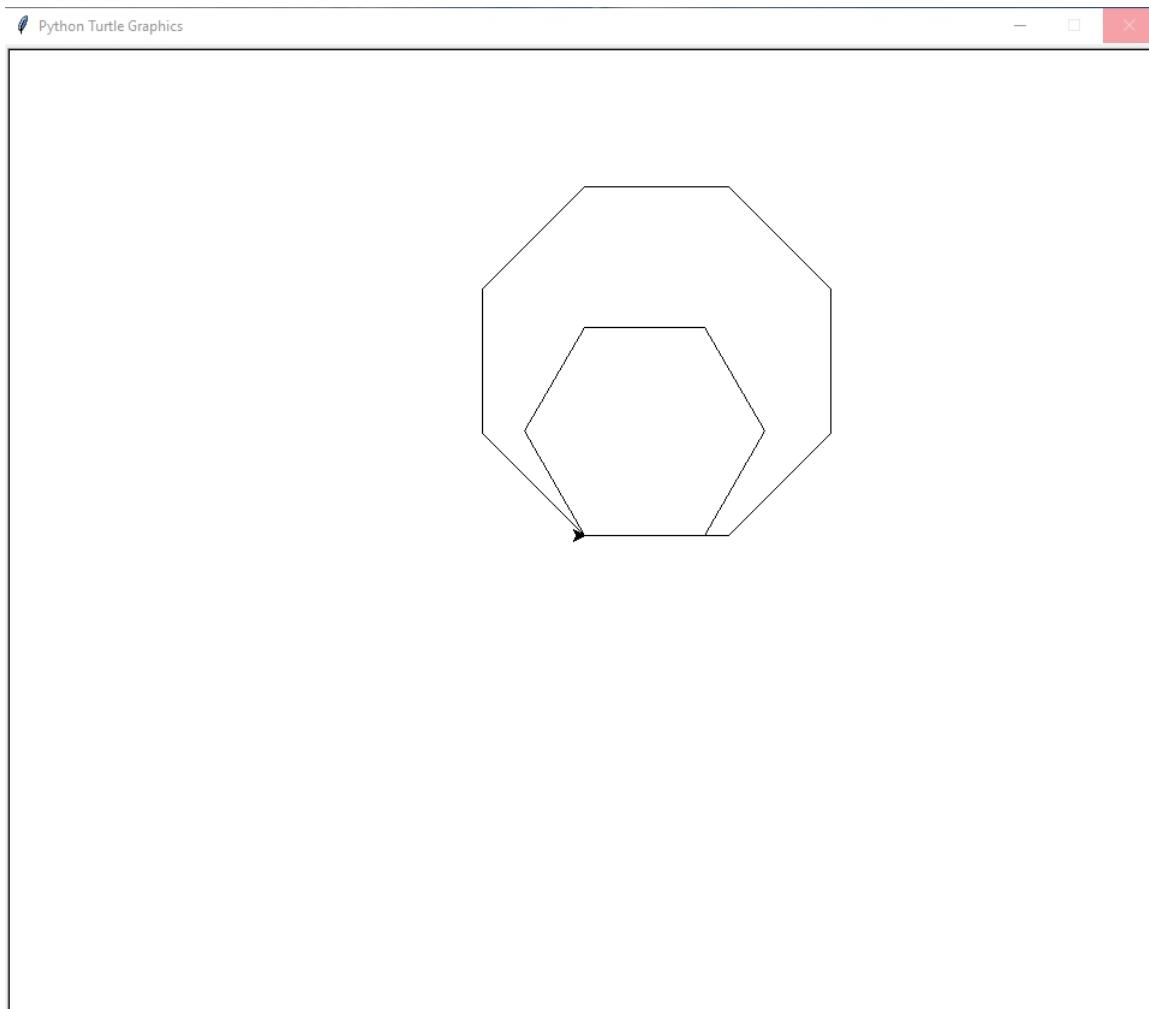


Figure 6-8: Calling the polygon() function with named arguments without resetting the screen

We could also call the square function with named arguments too, as in cell 25. And the result will be a screen with only a square of 250 pixels size.

```
[2 5]: 1 # call the square fn with named arguments
2 square(t, d=250, is_screen_reset=True)
```

OceanofPDF.com

Circle

Now let's draw a circle with the turtle. It might be a little tricky. Let's first define the `circle()` function then talk about the steps. One important point to note is, the `circle()` function will not do any drawing in it. It will call the `polygon()` function and pass the necessary arguments. All the drawing will take place in the `polygon()` function. A circle is nothing but a polygon with infinitely many sides.

The parameters of the `circle()` will be:

- a turtle object, `t`
- int `r`, the radius of the circle
- bool `is_screen_reset`, to check if the caller wants a screen reset

[2
6]:

```
1  import math
2
3  def circle(t, r, is_screen_reset):
4      """
5          Draws a circle.
6          Parameters: turtle t, int r, bool
7          is_screen_reset
8          Returns: None (void)
9
10         # perimeter -> total distance to draw
11         perimeter = 2 * math.pi * r
```

```

12
13      # angles to turn at each step
14      angles = 10
15
16      # steps -> 360 / angles
17      # angle of an arc
18      steps = int(360 / angles)
19
20      # distance -> distance to draw in each arc
21      distance = perimeter / steps
22
23      # call polygon
24      polygon(t, d=distance, n=steps,
           is_screen_reset=is_screen_reset)

```

In cell 26, we calculate the circumference (perimeter) of the circle in line 11. We import the **math** module to use the pi number as: **math.pi**. Then in line 14, we define a variable, **angles**, to keep the degrees the turtle will turn at each step. In line 18, we calculate the total number of steps that the turtle should take. In line 21, we calculate the distance which the turtle should move at each step. And since we have everything we need to call the **polygon()** function, in line 24, we call it with the named arguments.

If we call the **circle()** function, we will see a circle on the screen. It may not be a perfect circle, but it is quite close.

```

[2  1 # call the circle function with radius of 200 and
7]: 1 resetting the screen
      2 circle(t, 200, True)

```

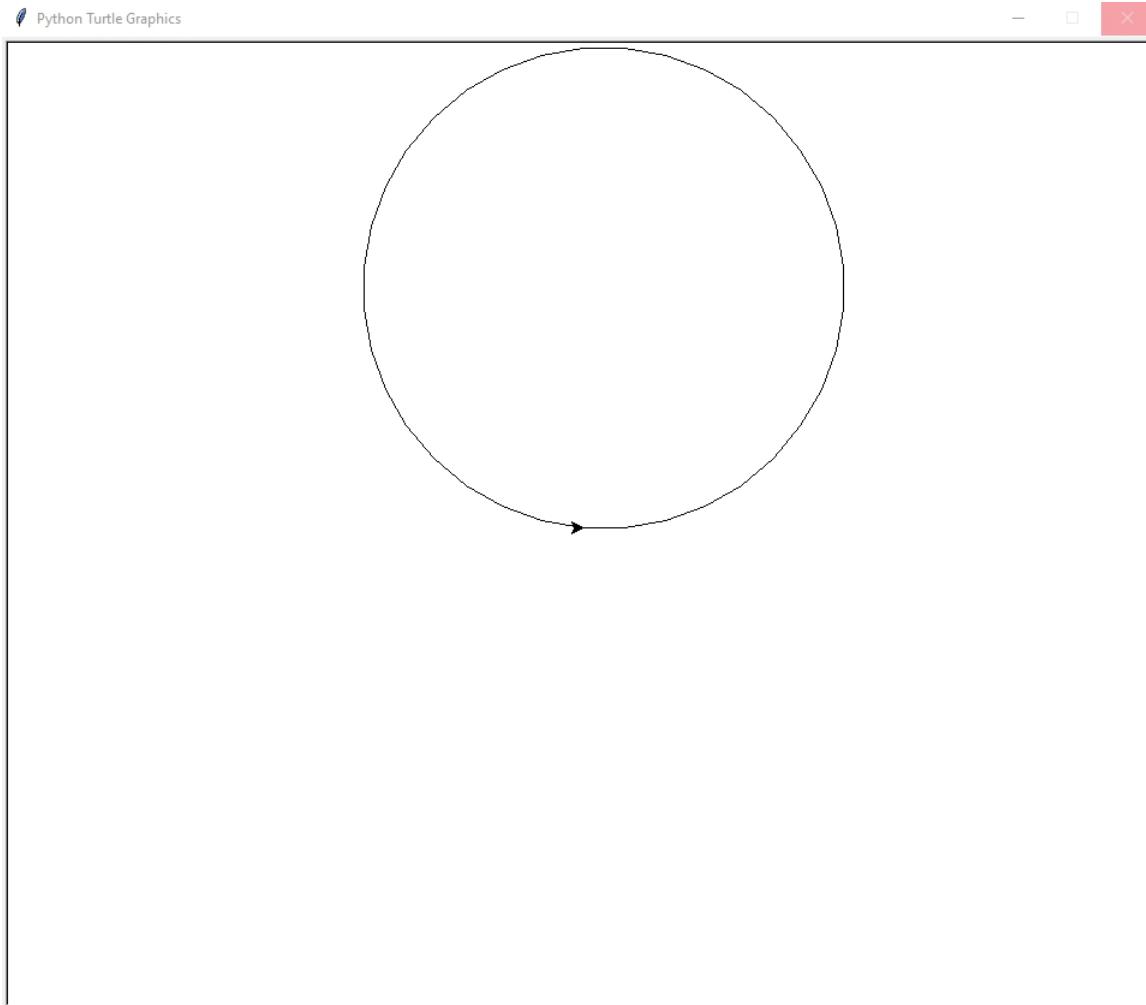


Figure 6-9: An approximate circle with the `circle()` function

And now you see the power of functions. The `circle()` function just passes the parameters into the `polygon()` function and we get a nice looking circle.

Let's call the circle function in a loop, to draw multiple circles at once:

```
[2 8]: 1 # reset the screen first
        2 reset()
        3
```

```
4 # call the circle in a loop
5 for i in range(1, 4):
6     # modify the radius at each iteration
7     circle(t, i * 50, False)
```

We set a **for** loop in the **range(1, 4)** which are numbers of 1, 2 and 3. At every iteration, the variable **i** will be one of the numbers of 1, 2 and 3 respectively. And we will use this number to increase the radius by 50 pixels, in line 7. Here is the resulting screen:

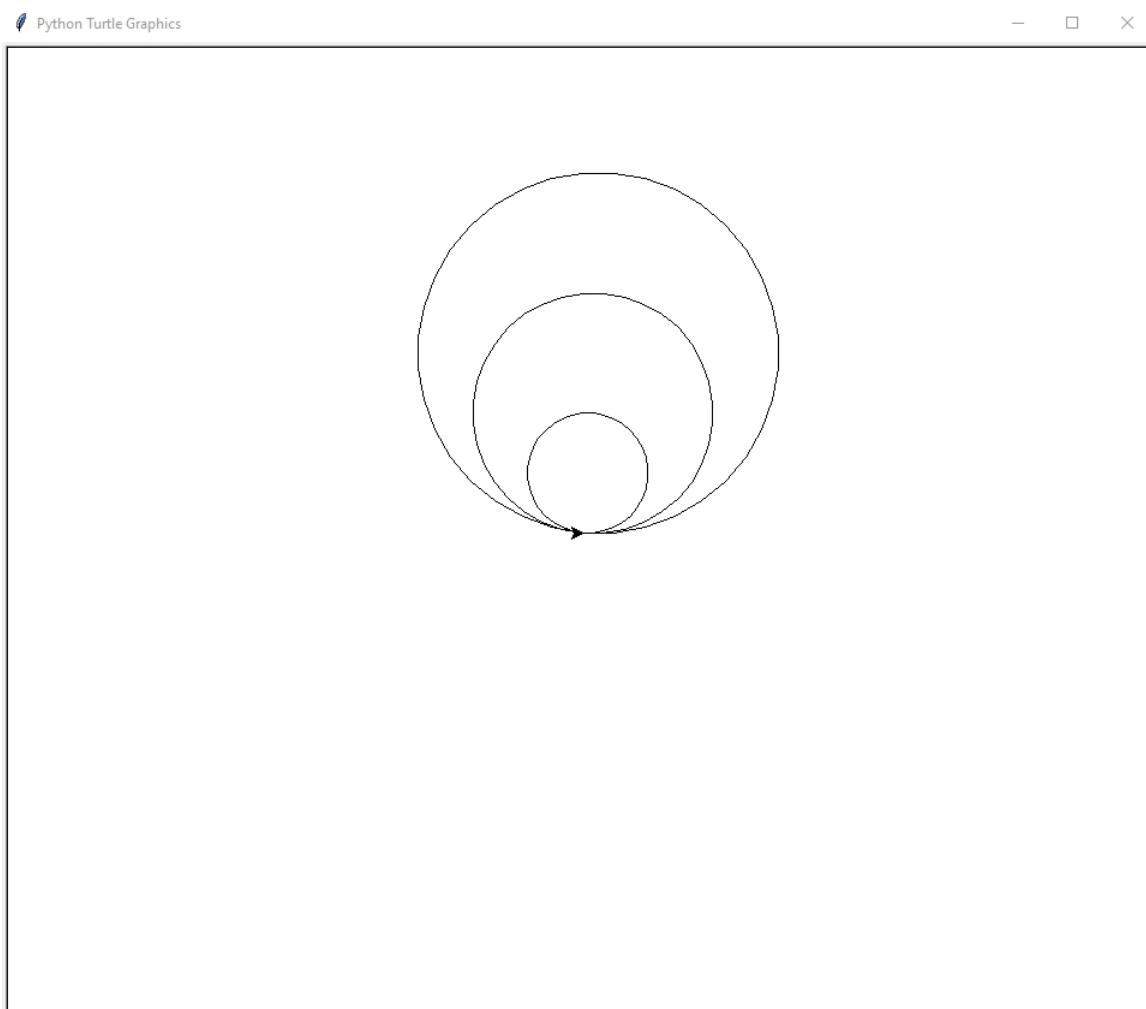


Figure 6-10: Three consecutive circles drawn by the for loop

OceanofPDF.com

Star

So far we have defined functions for the shapes: square, polygon and circle. Now let's draw a more interesting one, a star. And our star will be colorful.

To start with, we have to calculate the angle of a star. Don't worry if you don't know the math behind it, we only do it to assign a variable. The logic behind is not needed. You can just take it as 144 degrees.

```
[2  
9]: 1 # angle of a 5 sized star  
      2  
      3 angle = 180 / 5  
      4 angle = 180 - angle  
      5 print(angle)
```

```
[2  
9]: 144.0
```

Now let's define the `star()` function. The parameters will be:

- a turtle object, `t`
- int `d` (distance), in pixels
- bool `is_screen_reset`, to check if the caller wants a screen reset
- string `color`, the color name

```
[3  
0]: 1 def star(t, d, is_screen_reset, color):  
      2     """
```

```

3      Draws a star with 5 sizes.
4      Parameters: turtle t, int d, bool
5      is_screen_reset, str color
6
7      # resetting
8      if is_screen_reset:
9          reset()
10
11     # angle
12     angle = 180 / 5
13     angle = 180 - angle
14
15     # line color -> pen color
16     t.color(color)
17
18     # start the filling
19     t.fillcolor(color)
20     t.begin_fill()
21
22     for i in range(5):
23         t.fd(d)
24         t.lt(angle)
25
26     # end the filling
27     t.end_fill()

```

In cell 30, we define the `star()` function. We first assign a color to the turtle in line 16 as: `t.color(color)`. There are two color attributes, the pen color for the shape outline and the fill color. In line 19, we assign the fill color. Both colors are from the `color` parameter of the function. Then in line 20, we start the filling

via `t.begin_fill()`. In line 22, we set a `for` loop to move the turtle forward and turn left 5 times. At each time it turns by `angle` degrees. And in line 27, we end up the filling. Any shape will be filled in between `begin_fill()` and `end_fill()` functions.

Let's call the `star()` function with green color and see a beautiful green star on the screen:

```
[3]  
1 star(t, 300, is_screen_reset=True, color='green')
```

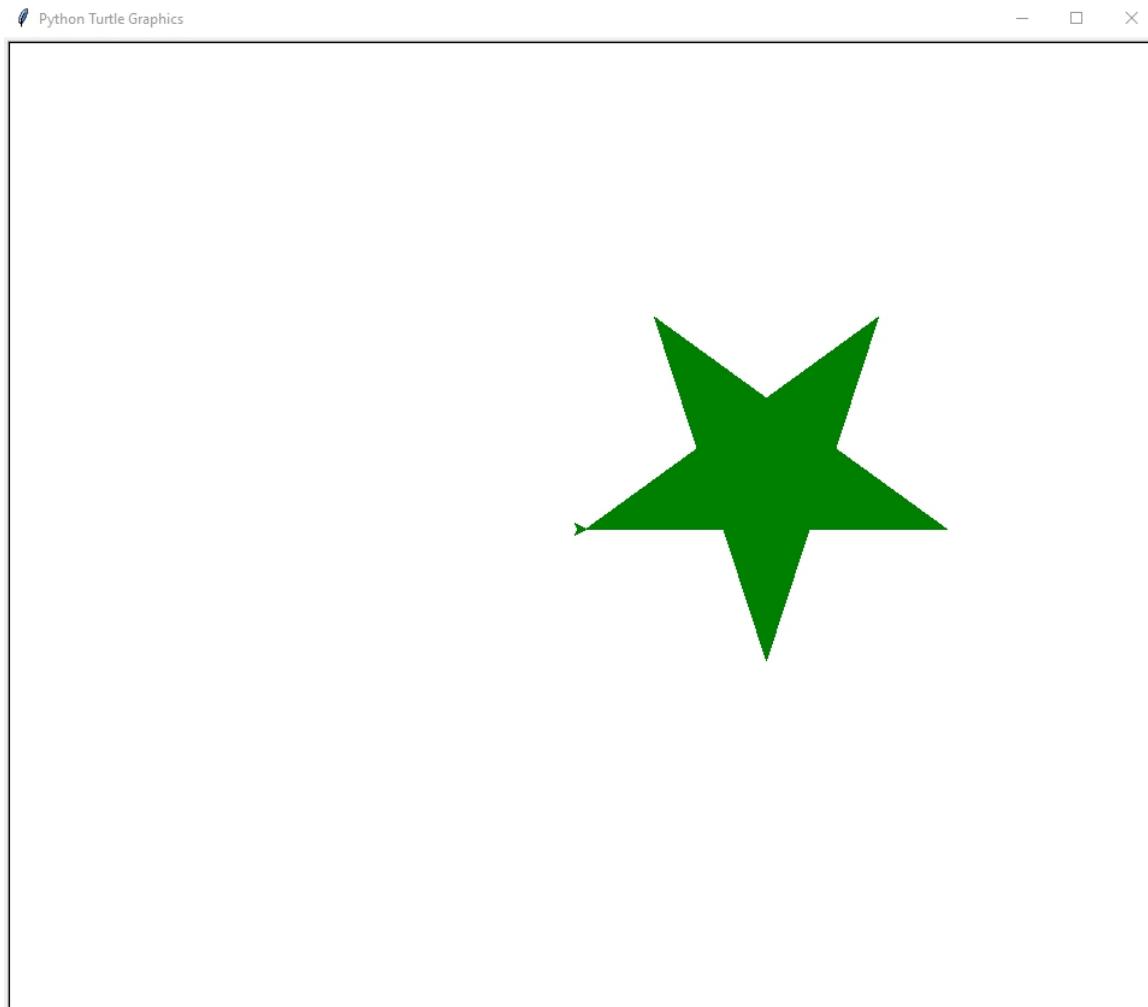


Figure 6-11: The start with 300 distance and green color

Let's draw another star, but do not reset the screen this time. We will have two stars when we run cell 32.

```
[3 2]: 1 star(t, 200, is_screen_reset=False, color='orange')
```

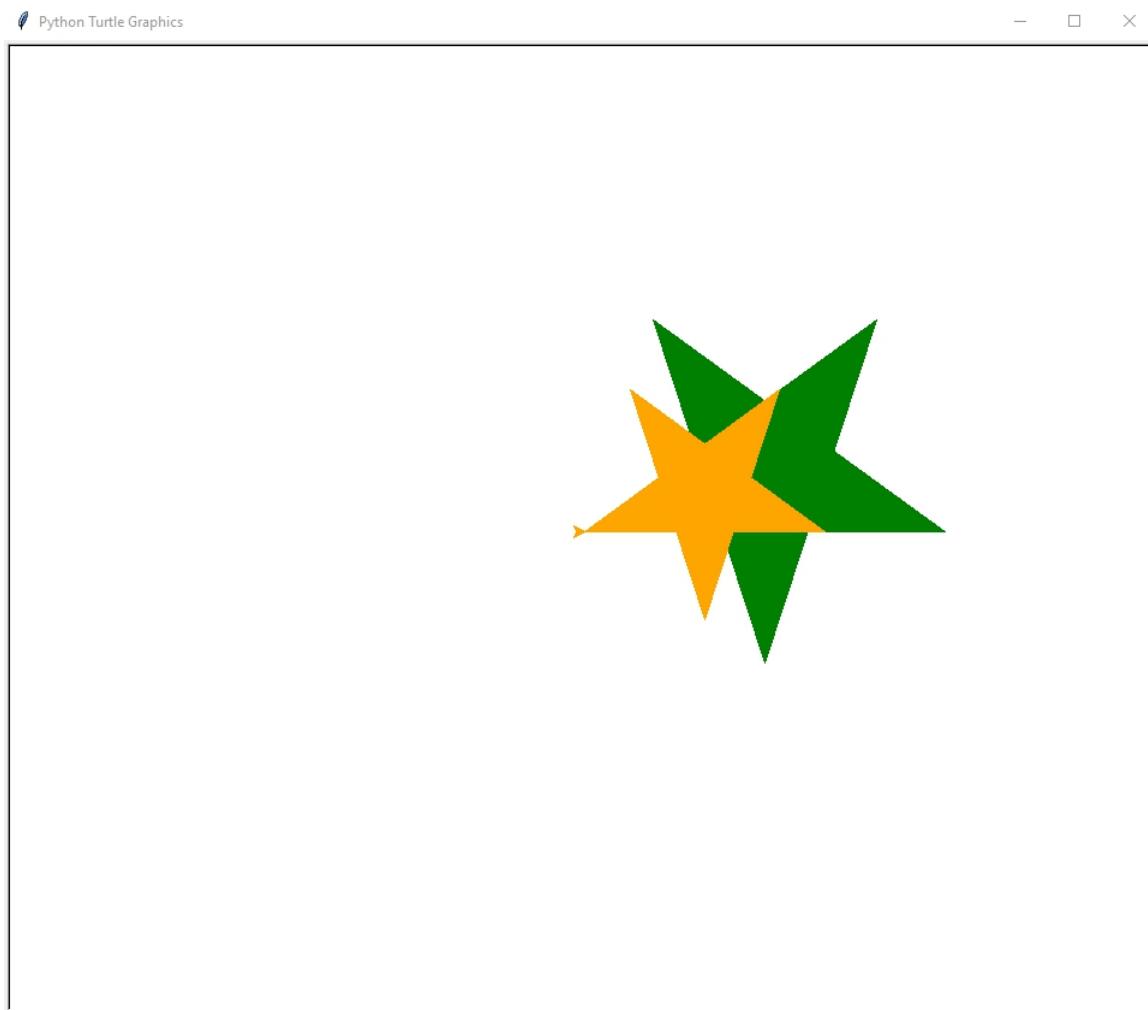


Figure 6-12: Green and orange stars

Let's draw a third star, a purple one.

```
[3 3]: 1 star(t, 100, is_screen_reset=False, color='purple')
```

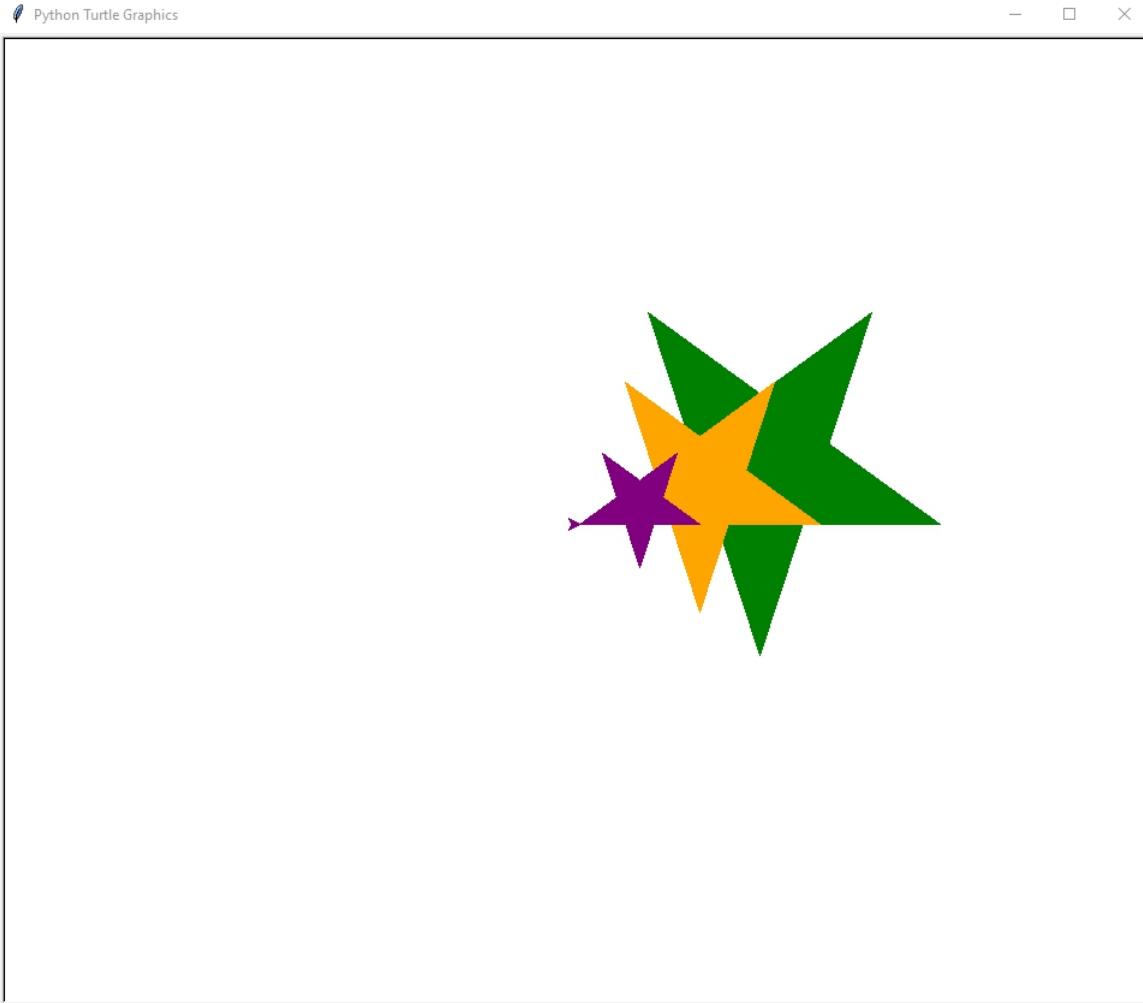


Figure 6-13: Three stars with varying side lengths

Our `star()` function looks good but we can improve it. For example it doesn't have to be 5 sided. Let's add another parameter, the number of sides, `n`, and draw stars with different shapes. The angle will be dependent on the number of sides, `n`.

```
[3] 1  # redefine the star function with number of sides
4]: 2  as a new parameter
     3  def star(t, d, n, is_screen_reset, color):
     4      """
     5          Draws a star with n sizes.
```

```

6  Parameters: turtle t, int d, int n, bool
7  is_screen_reset, str color
8  """
9
9  # resetting
10 if is_screen_reset:
11     reset()
12
13 # angle
14 angle = 180 / n
15 angle = 180 - angle
16
17 # line color -> pen color
18 t.color(color)
19
20 # start the filling
21 t.fillcolor(color)
22 t.begin_fill()
23
24 for i in range(n):
25     t.fd(d)
26     t.lt(angle)
27
28 # end the filling
29 t.end_fill()

```

The only difference in this new `star()` function is the way we calculate the `angle`. Instead of dividing 180 degrees by 5 (as we did in cell 30) we will divide 180 by the number of sides, `n`. You can see it in line 14. The rest is all the same.

Let's call it with 11 sides and yellow color, by resetting the screen:

```
[3] 1 star(t, 300, 11, is_screen_reset=True,  
5]: 2 color='yellow')
```

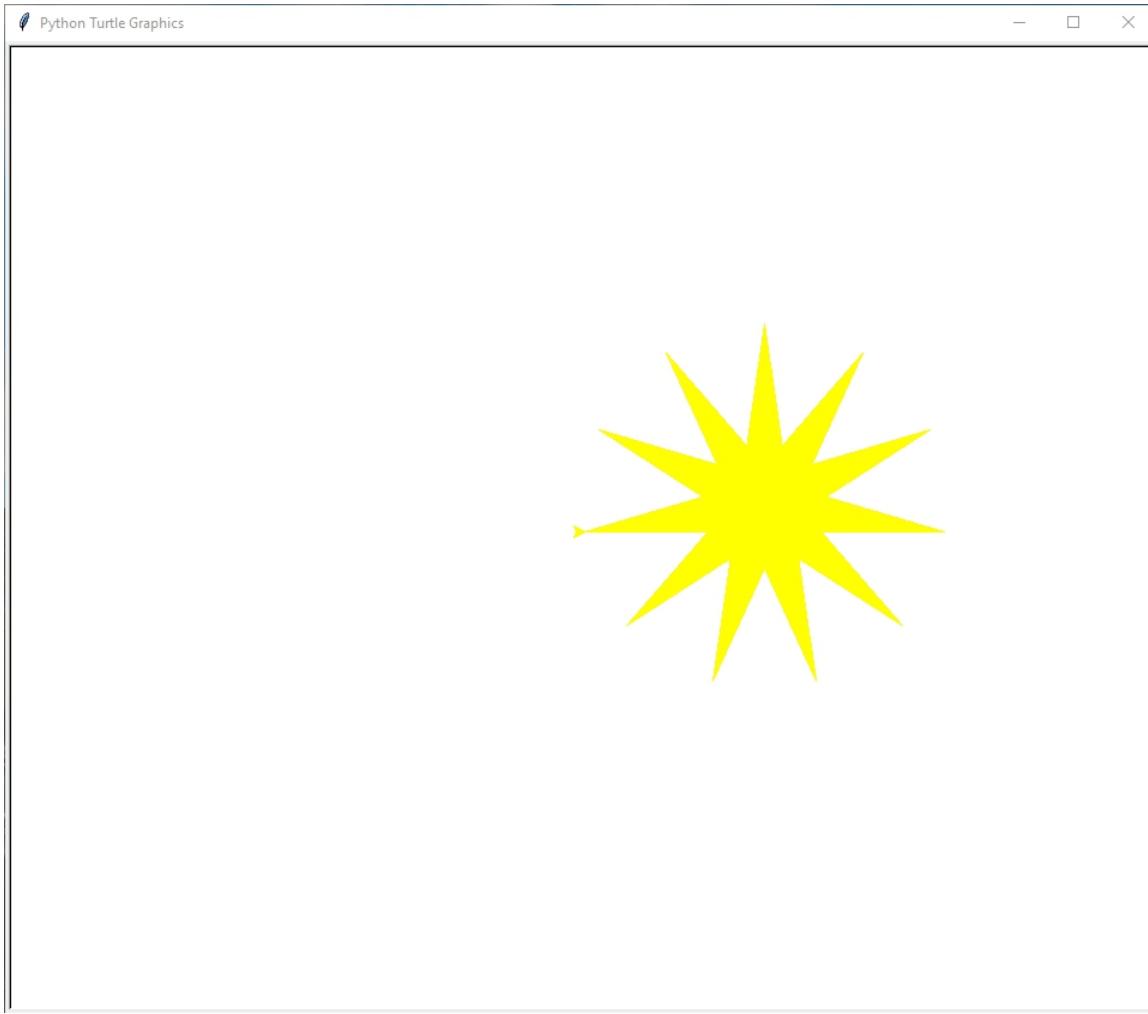


Figure 6-14: A yellow star with 11 sides

As a final exercise, let's improve our `star()` function a bit more. Let's make pen color and fill color parametric this time. We will remove the `color` parameter and add two new parameters, `pen_color` and `fill_color`.

```
[3] 1 # redefine the star function with pen_color and  
6]: 2 fill_color  
     2
```

```

3  def star(t, d, n, is_screen_reset, pen_color,
4      fill_color):
5          """
6              Draws a star with n sizes.
7              Parameters: turtle t, int d, int n, bool
8              is_screen_reset, str pen_color, fill_color
9          """
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

```

resetting

if is_screen_reset:
 reset()

angle

 angle = 180 / n
 angle = 180 - angle

line color -> pen color

 t.color(pen_color)

start the filling

 t.fillcolor(fill_color)
 t.begin_fill()

for i **in** range(n):
 t.fd(d)
 t.lt(angle)

end the filling

 t.end_fill()

In cell 36, line 18, we passed the **pen_color** parameter into the **color()** function as: **t.color(pen_color)**. And in line 21, we passed the **fill_color** parameter into the **fillcolor()** function as:

`t.fillcolor(fill_color)`. Let's call it with different pen and fill colors:

```
[3] 1 star(t, 300, 15, is_screen_reset=True,  
7]:   pen_color='red', fill_color='yellow')
```

And here is the result. A beautiful star with 15 sides, red outline and yellow fill color:



Figure 6-15: A star with 15 sides, different pen color and fill color

As the last cell in our turtle program, we must run the `mainloop()` function. Then we can close the turtle screen without

any errors.

```
[3  
8]: 1 # last statement -> before close  
     2 # restart the kernel after you close the turtle  
     3  
     4 turtle.mainloop()
```

And that's the end of the first project in this book. We defined functions, passed arguments, modify them and reuse them in this project. Now you should have a good understanding of basics of functions in Python. We will have the second part of functions in Chapter 9.

OceanofPDF.com

7. Assignment 1 – Functions

We finished our first project, Project 1 - Functions. Now it's your turn to recreate the same project. This chapter is the assignment on the Project 1 Functions. In this assignment, you will use Python's built-in Turtle module. You will define functions, and call them via parameters to draw shapes with turtle. So you will be able to practice functions by yourself.

To complete the assignment, follow the instructions where you see a **#TODO** statement. You can also download the assignment as a Jupyter file in the [Github repository](#) of the book. You can find the solutions in the previous chapter.

The Assignment

Complete the code with respect to **#TODO** statements.

```
[  
1  1 # import the module -> turtle  
]:  
2  
3 #TODO - import turtle module
```

```
[  
2  1 #TODO - create a turtle object and name it as t  
]:
```

```
[  
3  1 # move turtle forward by 100 pixels  
]:
```

```

2 t.fd(100)

[4]: 1 # turn the turtle to the left by 90 degrees
]: 2 t.lt(90)

[5]: 1 #TODO - move turtle t 100 pixels forward
]: 

[6]: 1 #TODO - turn turtle to the left by 90 degrees
]: 

[7]: 1 #TODO - move turtle t 100 pixels forward
]: 

[8]: 1 #TODO - turn turtle to the left by 90 degrees
]: 

[9]: 1 #TODO - move turtle t 100 pixels forward
]: 

[10]: 1 #TODO - turn turtle to the left by 90 degrees
]: 

```

Move functions:

- fd(px) -> forward
- bk(px) -> backward
- lt(angle) -> turns left
- rt(angles) -> turns right

```

[1]: 1 # clear screen and reset the turtle to initial position
]: 2 t.clear()

```

```
[1 2]: 1 # reset the screen
      2 turtle.resetscreen()

[1 3]: 1 # We want to create a square
      2
      3 # Move 100 pixel forward
      4 # Turn left 90 deg
      5
      6 # Move 100 pixel forward
      7 # Turn left 90 deg
      8
      9 # Move 100 pixel forward
     10 # Turn left 90 deg
     11
     12 # Move 100 pixel forward
     13 # Turn left 90 deg
     14
     15 t.fd(100)
     16 t.lt(90)
     17
     18 t.fd(100)
     19 t.lt(90)
     20
     21 t.fd(100)
     22 t.lt(90)
     23
     24 t.fd(100)
     25 t.lt(90)
```

```
[1 4]: 1 #TODO - reset turtle screen
```

```
[1 5]: 1 # with loop
```

```
2 # it's going to repeat fd-lt patter 4 times
3
4 for i in range(...#TODO...):
5     #TODO - move forward and turn left
```

Square:

Create a square with a function -> square

Parameters: turtle object, d in pixels and is_screen_reset

```
[1] 1 # square function
[6]: 2
3 def square(t, d, is_screen_reset):
4     """
5         It draws square with turtle.
6         Parameters: turtle t, int d, bool
7         is_screen_reset
8         Returns: None
9
10        """
11        # reset screen
12        if is_screen_reset == True:
13            reset()
14
15        # draw a square
16        #TODO - create a loop to draw a circle
```

```
[1] 1 # reset function
[7]: 2
3 def reset():
4     #TODO - call turtle modules resetscreen function
```

```
[1] 1 square(t, 200, True)
[8]:
```

```
[1 9]: 1 #TODO - call square function with: t, 300 distance  
       and resetting screen  
[2 0]: 1 #TODO - call square function with: t, 100 distance  
       and NOT resetting screen  
[2 1]: 1 # draw 10 squares of size: 20 - 40 - 60 80 - ... - 200  
      2  
      3 # first reset the screen just once  
      4 reset()  
      5  
      6 # loop to draw the circles  
      7 for i in range(1, 11):  
          #TODO - call square function with NOT resetting  
          screen and appropriate parameters
```

Polygon:

Draw a polygon.

```
[2 2]: 1 # polygon function  
      2  
      3 def polygon(t, d, n, is_screen_reset):  
          #TODO - write the function documentation  
          (docstring)  
      4  
      5 # resetting  
      6 #TODO - check if screen will be reset and do  
      7 it if needed  
      8  
      9 # angle  
     10 angle = 360 / n  
     11  
     12 # polygon  
     13 for i in range(...#TODO...):
```

```
14      #TODO - move and turn the turtle to draw
      polygon
```

```
[2
3]: 1 # draw a 6 sided hexagon
2 polygon(t, 100, 6, True)
```

Named Arguments:

When you call a function and passing parameters:

```
function_name(parameter_name = parameter_value)
```

```
[2
4]: 1 # call the polygon fn with named arguments
2 polygon(t=t, d=120, n=8, is_screen_reset=False)
```

```
[2
5]: 1 # call the square fn with named arguments
2
3 #TODO - call square function with named
parameters
4 # -> t for turtle, 250 for d and True for
is_screen_reset
```

Circle:

Draw a Circle ->

- t
- r
- is_screen_reset

```
[2
6]: 1 import math
2
3 def circle(t, r, is_screen_reset):
4     """
5         Draws a circle.
6         Parameters: turtle t, int r, bool
7         is_screen_reset
8         Returns: None (void)
```

```

8      """
9
10     # perimeter -> total distance to draw
11     # TODO - Calculate the perimeter of circle ->
12     2 * math.pi * r
13
14     # angles to turn at each step
15     angles = 10
16
17     # steps -> 360 / angles
18     # angle of an arc
19     # TODO - Calculate number of steps for arc
20     # of 10 degrees, then convert into int
21
22     # distance -> distance to draw in each arc
23     # TODO - Calculate the distance which is
24     # perimeter / steps
25
26     # call polygon
27     # TODO - Call polygon function with
28     # appropriate parameters

```

[2]
7]:

```

1 # call the circle function with radius of 200 and
2     resetting the screen
3 circle(t, 200, True)

```

[2]
8]:

```

1 # reset the screen first
2 # TODO - call reset() function
3
4 # call the circle in a loop
5 for i in # TODO - Range from 1 to 4:
6
7     # TODO - Call circle() function with i * 50
8     radius and do not reset screen

```

Star:

Draw a star.

```
[2
9]: 1 # angle of a 5 sized star
      2
      3 angle = 180 / 5
      4 angle = 180 - angle
      5 print(angle)
```

```
[2
9]: 144.0
```

```
[3
0]: 1     def star(t, d, is_screen_reset, color):
      2
      3     """
      4     Draws a star with 5 sizes.
      5     Parameters: turtle t, int d, bool
      6     is_screen_reset, str color
      7     """
      8
      9     # resetting
      10    # TODO - call reset screen based on
      11    is_screen_reset
      12
      13    # angle
      14    # TODO - Calculate the angle for 5 sizes
      15
      16    # line color -> pen color
      17    t.color(color)
      18
      19    # start the filling
      20    t.fillcolor(color)
      21    t.begin_fill()
      22
      23    # 5 sized star
      24    for i in range(...#TODO...):
```

```
22      #TODO - move and turn the turtle to draw
23      a star
24
25      # end the filling
26      t.end_fill()
```

```
[3
1]: 1 star(t, 300, is_screen_reset=True, color='green')
```

```
[3
2]: 1 star(t, 200, is_screen_reset=False, color='orange')
```

```
[3
3]: 1 star(t, 100, is_screen_reset=False, color='purple')
```

Variable sized star.

```
[3
4]: 1      # redefine the star function with number of sides
2      as a new parameter
3
4      def star(t, d, n, is_screen_reset, color):
5          """
6              Draws a star with n sizes.
7              Parameters: turtle t, int d, int n, bool
8              is_screen_reset, str color
9          """
10
11
12      # resetting
13      # TODO - call reset screen based on
14      is_screen_reset
15
16      # angle
17      # TODO - Calculate the angle for n sizes
18
19      # line color -> pen color
20      # TODO - set the line color
21
22      # start the filling
```

```
19      # TODO - set the fill color
20      # TODO - begin fill
21
22      # n sized star
23      for i in range(...#TODO...):
24          #TODO - move and turn the turtle to draw
25          a star
26
27      # end the filling
28      # TODO - end fill
```

```
[3] 1 star(t, 300, 11, is_screen_reset=True,
5]: 1 color='yellow')
```

Different fill and pen colors.

```
[3] 1 # redefine the star function with pen_color and
6]: 1 fill_color
2
3 def star(t, d, n, is_screen_reset, pen_color,
3 fill_color):
4     """
5     Draws a star with n sizes.
6     Parameters: turtle t, int d, int n, bool
6     is_screen_reset, str pen_color, fill_color
7     """
8
9     # resetting
10    # TODO - call reset screen based on
10    is_screen_reset
11
12    # angle
13    # TODO - Calculate the angle for n sizes
14
15    # line color -> pen color
16    # TODO - set the line color
```

```
17
18      # start the filling
19      # TODO - set the fill color
20      # TODO - begin fill
21
22      # n sized star
23      for i in range(...#TODO...):
24          #TODO - move and turn the turtle to draw
25          # a star
26
27      # end the filling
28      # TODO - end fill
```

```
[3
7]: 1 star(t, 300, 15, is_screen_reset=True,
      pen_color='red', fill_color='yellow')
```

```
[3
8]: 1 # last statement -> before close
2 # restart the kernel after you close the turtle
3
4 turtle.mainloop()
```

OceanofPDF.com

8. Conditional Statements

What is a Conditional Statement?

Python runs the code from top to down. We call it as the Execution Flow. There will be places in the code that you may want to control this flow. For example, let's say you want to print the number if and only if it is positive. This decision making is called **conditional execution** and in Python it is done via **if... elif... else** statements. In this chapter we will learn the **conditional statements**.

A **conditional statement** is an expression which evaluates either **True** or **False**. Let's see some examples:

```
[  
1  6 == 6  
]:
```

```
[  
1  True  
]:
```

As you see in cell 1, we check if 6 is equal to 6 via a double equal sign (`==`). And the result is `True`, as we expect. This is a simple conditional statement.

Important Note: In Python single equal sign (`=`) is used for **assignment**, but the double equal signs (`==`) is used for **equality check**.

```
[2]: 1 # single equal sign -> = , assignment
]: 2 a = 6
3 a
```

```
[2]: 6
]:
```

In cell 2, we define a variable named `a`. We used the assignment operator which is the single equal sign (`=`).

```
[3]: 1 # double equal sign -> == , equality check
]: 2 a == 6
```

```
[3]: True
]:
```

In cell 3, we check if the value of the variable `a` is equal to `6` or not via double equal signs (`==`). Here it is not an assignment, it is an equality check. Let's check another value for `a`:

```
[4]: 1 a == 8
]:
```

```
[4]: False
]:
```

In cell 4, we ask Python, if the value of variable `a` is equal to `8`, and it returns `False`. Because the value of `a` is `6`.

True and **False**: In Python, **Boolean** variables are defined by **True** and **False** keywords.

```
[  
5  1 type(True)  
]:
```

```
[  
5  1 bool  
]:
```

Boolean Expressions:

A boolean expression evaluates to one of two states **True** or **False**. Let's see what they are in Python. In the list below, every boolean expression checks for a specific condition:

- `x == y` : x is equal to y
- `x != y` : x is not equal to y
- `x < y` : x is less than y
- `x > y` : x is greater than y
- `x <= y` : x is less than or equal to y
- `x >= y` : x is greater than or equal to y

Let's define two variables, `x` and `y`, and check all of these boolean expressions.

```
[  
6  1 x = 12  
]:  
2  y = 8  
3  
4  print("x == y : ", x == y)
```

```
5 print("x != y :", x != y)
6 print("x < y :", x < y)
7 print("x > y :", x > y)
8 print("x <= y :", x <= y)
9 print("x >= y :", x >= y)
```

```
[6]: x == y : False
      x != y : True
      x < y : False
      x > y : True
      x <= y : False
      x >= y : True
```

Logical Operators:

In Python, we have 3 logical operators: **and**, **or**, and **not**.

- **and**: True if both of the conditions are True, False otherwise
- **or**: True if any of the conditions is True, False if all conditions are False
- **not**: negates a boolean expression, “not True” is False

Now let's use these logical operators in the code:

and:

```
[7]: 1 True and True
      : 1
```

```
[7]: True
```

```
]: [8]: 1 True and False
]: [8]: False
]: [9]: 1 False and True
]: [9]: False
[1 0]: 1 False and False
[1 0]: False
```

or:

```
[1 1]: 1 True or True
[1 1]: True
[1 2]: 1 True or False
[1 2]: True
[1 3]: 1 False or True
```

```
[1  
3]: True
```

```
[1  
4]: 1 False or False
```

```
[1  
4]: False
```

Now let's do some examples with variables:

```
[1  
5]: 1 # a -> True  
2 a = 5 == 5  
3 a
```

```
[1  
5]: True
```

In cell 15, we first evaluate if 5 is equal to 5 as `5 == 5`, which returns `True`. Then we assign this `True` value to the variable `a`. That's why the value of `a` prints as `True`.

```
[1  
6]: 1 # b -> False  
2 b = 'a' == 'A'  
3 b
```

```
[1  
6]: False
```

In cell 16, we first evaluate if the string '`a`' is equal to string '`A`' as `'a' == 'A'`, which returns `False`. Then we assign this `False` value to the variable `b`. That's why the value of `b` prints as `False`.

```
[1 7]: 1 # and
2 print("---- and ----")
3 "{0} and {1}: {2}".format(a, b, a and b)
```

```
[1 7]: ---- and ----
'True and False: False'
```

In cell 17, we used a built-in string method (function) which is **string.format()**. Our string is "**{0} and {1}: {2}**" and we put **.format()** method at the end of this string as:

```
"{0} and {1}: {2}".format(a, b, a and b)
```

There are 3 arguments in the **format()** function call:

- 1) **a**
- 2) **b**
- 3) **a and b**

And these 3 arguments will be placed at the placeholders inside the string. **{0}**, **{1}** and **{2}** are the placeholders. So the final string will print as: **True and False: False**. Be careful that, the argument **a and b** evaluates as **False**. Because the variable **a** is **True** and **b** is **False**. **True and False is False**.

```
[1 8]: 1 # or
2 print("---- or ----")
3 "{0} or {1}: {2}".format(a, b, a or b)
```

```
[1 8]: ---- or ----
'True or False: True'
```

In cell 18, this time we put **a or b** as the third argument in the **format()** function call. And it evaluates as **True**, because the variable **a** is **True** and **b** is **False**. **True or False is True**.

OceanofPDF.com

True / False - What is a Conditional Statement?

For each questions below, decide whether it's true or false.

Q1:

In Python = and == are the same. They both check for equality.

- True
- False

Q2:

(True and False) or (False or True) expression will evaluate as **True**.

- True
- False

SOLUTIONS - True / False - What is a Conditional Statement?

Here are the solutions for True / False questions.

S1:

In Python = and == are the same. They both check for equality.

- True
- False

It's **false**. "=" is assignment, whereas "==" is equality check. They are different operators.

S2:

(True and False) or (False or True) expression will evaluate as **True**.

- True
- False

It's **true**. First parenthesis is **False**, and the second one is **True**. Since there is "**or**" between parentheses, the final result is **True**.

if

The **if statement** in Python, is used for decision-making operations. It contains a block of code which only runs when the condition given in the **if** statement is **True**. Let's see how it works in the code:

```
[1
9]: 1 # Print x, if it is greater than zero (positive)
      2
      3 x = 7
      4
      5 if x > 0:
      6     print("{} is greater than 0.".format(x))
```

```
[1
9]: 7 is greater than 0.
```

In cell 19 line5, we have an **if** statement. The condition check is “if **x** is greater than zero or not”: **x > 0**. If that expression evaluates as **True** then Python will run the code in the **if** block, which is the line 6. And it will print a text with **string.format()** function. Since we already assigned **x** as 7, it is greater than 0. So the if condition is **True** and it will print the text of “7 is greater than 0.” in the output.

```
[2
0]: 1 x = -10
      2
      3 if x > 0:
      4     print("{} is greater than 0.".format(x))
      5
```

```
6 print("end of if statement")
```

[2
0]: end of if statement

In cell 20, we assign the value **-10** to the variable **x**. Since it is not greater than 0, the condition check in the **if** statement returns **False**: **x > 0**. And since the condition check is **False**, Python will not run the code in the **if** block, which is the line 4. That's why we only see the text “end of if statement” in the output.

```
[2  
1]: 1      # If x is greater than 0 and less than or equal to  
2          10, print it  
3  
4          x = 10  
5  
6          print("-----")  
7          if x > 0 and x <= 10:  
8              print("{} is greater than 0 and less than or  
9                  equal to 10.".format(x))  
10         print("-----")
```

[2
1]: -----
10 is greater than 0 and less than or equal to 10.

In cell 21, we check two conditions combined with the **and** operator. The value of **x** is 10. So **x > 0** is **True**, and **x <= 10** is also **True**. Finally the combined expression **x > 0 and x <= 10**

is **True**. Which means the condition check in the if statement returns **True** and Python will run the code in line 8.

```
[2
2]: 1 x = 17
      2
      3 print("-----")
      4
      5 if x > 0 and x <= 10:
          print("{} is greater than 0 and less than or equal
to 10.".format(x))
      6
      7
      8 print("-----")
```

```
[2
2]: -----
      -----
      -----
```

The condition check in cell 22 returns **False**. Since **x** is 17, **x <= 10** will return **False**. **And x > 0 and x <= 10** will also return **False**. That means the code in line 6 is not executed.

```
[2
3]: 1 x = 6
      2
      3 print("-----")
      4
      5 if x > 0 and x <= 10:
          print("{} is greater than 0 and less than or equal
to 10.".format(x))
      6
      7
      8 print("-----")
```

```
[2
3]: -----
      -----
```

6 is greater than 0 and less than or equal to 10.

The condition check in cell 23 is **True**, because **x > 0 and x <= 10** is **True** when **x** is 6. So the code line 6 is executed.

```
[2  
4]: 1 # if x is greater than or equal to 0 and less than  
2  
3 x = 19  
4  
5 print("-----")  
6  
7 if x >= 0 and x < 20:  
8     print("{} is greater than or equal 0 and less  
9         than 20.".format(x))  
10    print("-----")
```

```
[2  
4]: -----  
19 is greater than or equal 0 and less than 20.  
-----
```

In cell 24, the condition check for the if statement is **True**. That's because **x >= 0 and x < 20** evaluates as **True** when **x = 19**. The code line 8 is executed by the Python Interpreter.

```
[2  
5]: 1 x = 49  
2  
3 print("-----")  
4  
5 if x >= 0 and x < 20:
```

```
6     print("{} is greater than or equal 0 and less than  
7         20.".format(x))  
8     print("-----")
```

```
[2  
5]:
```

```
-----  
-----
```

In cell 25, the result of the expression `x >= 0 and x < 20` will return `False` when `x = 49`. The code line 6 is not executed.

```
1 x = -6  
2  
3 print("-----")  
4  
5 if x >= 0 and x < 20:  
6     print("{} is greater than or equal 0 and less than  
7         20.".format(x))  
8     print("-----")
```

```
[2  
6]:
```

```
-----  
-----
```

In cell 26, the result of the expression `x >= 0 and x < 20` will return `False` when `x = -6`. The code line 6 is not executed.

```
1     # or  
2  
3     # if x is greater than 100 or less than 0, print it  
4  
5     x = 80
```

```
6  
7     print("-----")  
8  
9     if x > 100 or x < 0:  
10        print("{} is greater than 100 or less than  
11        0.".format(x))  
12    print("-----")
```

```
[2  
7]:
```

```
-----  
-----
```

In cell 27, the result of the expression `x > 100 or x < 0` will return **False** when `x = 80`. The code line 10 is not executed.

```
[2  
8]:
```

```
1 x = 280  
2  
3     print("-----")  
4  
5     if x > 100 or x < 0:  
6        print("{} is greater than 100 or less than  
7        0.".format(x))  
8     print("-----")
```

```
[2  
8]:
```

```
-----  
280 is greater than 100 or less than 0.  
-----
```

In cell 28, the result of the expression `x > 100 or x < 0` will return **True** when `x = 280`. So the code line 6 is executed and it will print the text “280 is greater than 100 or less than 0.”.

```
[2 9]: 1 x = -300
2
3 print("-----")
4
5 if x > 100 or x < 0:
6     print("{} is greater than 100 or less than
7 0.".format(x))
8 print("-----")
```

```
[2 9]: -----  
-300 is greater than 100 or less than 0.  
-----
```

In cell 29, the result of the expression `x > 100 or x < 0` will return **True** when `x = -300`. So the code line 6 is executed and it will print the text “-300 is greater than 100 or less than 0.”.

OceanofPDF.com

True / False - if

For each questions below, decide whether it's true or false.

Q1:

The code below will print as:

60 is greater than or equal to 0 or less than 20

```
x = 60
```

```
if x >= 0 or x < 20:
```

```
    print("{} is greater than or equal to 0 or less than  
20".format(x))
```

- True
- False

Q2:

The code below will print as:

49 is greater than or equal to 0 and less than 20

```
x = 49
```

```
if x >= 0 and x < 20:
```

```
    print("{} is greater than or equal to 0 and less than  
20".format(x))
```

- True
- False

SOLUTIONS - True / False - if

Here are the solutions for True / False questions.

S1:

The code below will print as:

60 is greater than or equal to 0 or less than 20

```
x = 60
```

```
if x >= 0 or x < 20:
```

```
    print("{} is greater than or equal to 0 or less than  
20".format(x))
```



True



False

It's **true**, because **x >= 0 or x < 20** evaluates **True**.

S2:

The code below will print as:

49 is greater than or equal to 0 and less than 20

```
x = 49
```

```
if x >= 0 and x < 20:
```

```
    print("{} is greater than or equal to 0 and less than  
20".format(x))
```



True



False

It's **false**. **x < 20** will evaluates **False** when **x = 49**. Since "**and**" wants both sides to be **True**, the code in if statement will not run.

else

Most of the time we need more than one condition to check. For multiple conditions we use `else` statement. The `else` statement is used when you have to judge one condition on the basis of the other. If one condition turns out to be `False`, then there should be another condition that should justify the statement.

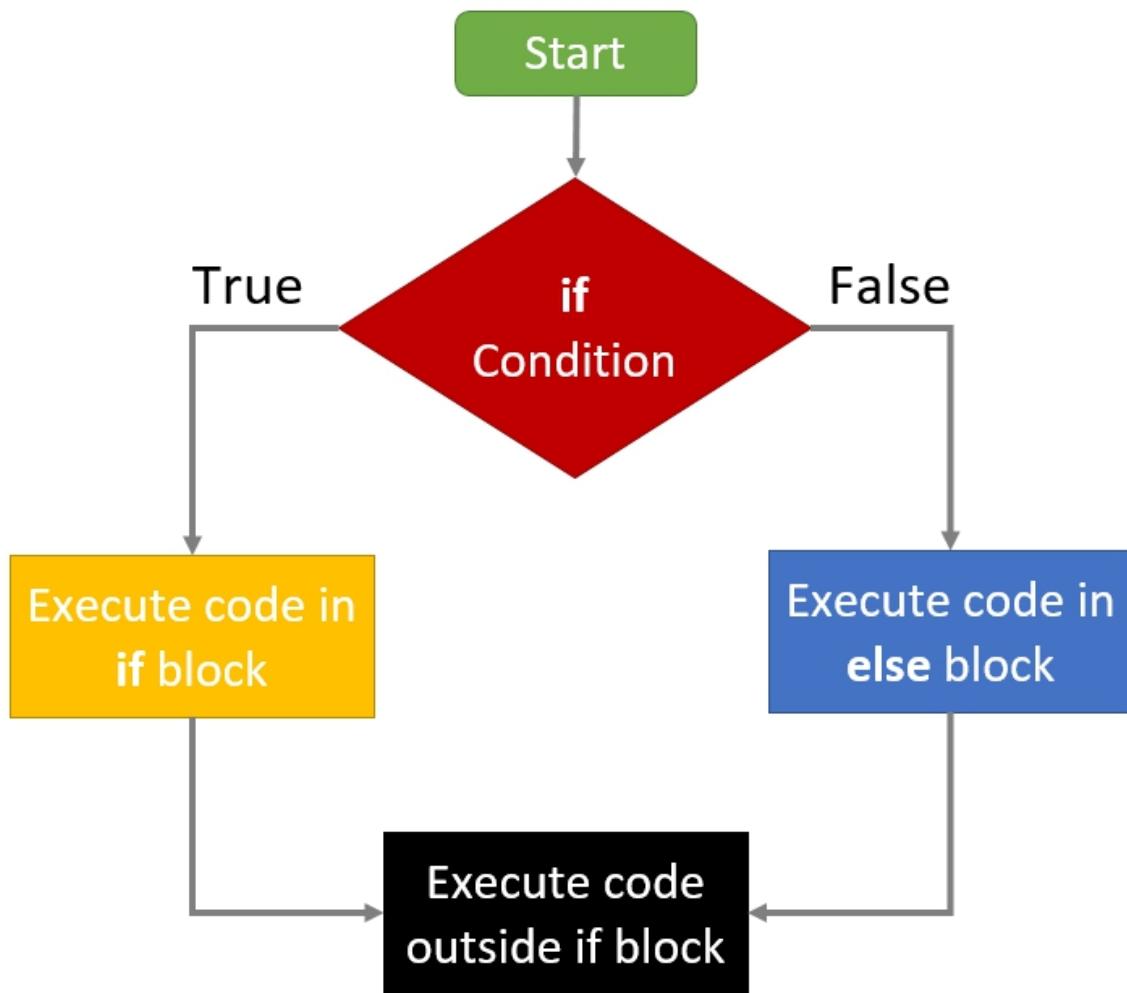


Figure 8-1: The if-else code flow

Let's see the `if-else` structure with examples:

```
[3  
0]: 1 # if x is an even number, print "EVEN"  
2 # if not (else) print "ODD"  
3  
4 # condition for being even: divisible by 2 (%  
modulus operator)  
5  
6 x = 14  
7  
8 print("---- before if block ----")  
9  
10 if x % 2 == 0:  
11     print("EVEN")  
12 else:  
13     print("ODD")  
14  
15 print("---- after if block ----")
```

```
[3  
0]: ---- before if block ----  
      EVEN  
      ---- after if block ----
```

In cell 30, we want to check if variable, `x` is even or odd. In line 10, the `if` condition is `x % 2 == 0`. Which checks if it's even. The remainder will be 0 when we divide an even number by 2. If the number is even, Python will execute line 11 and print as EVEN. If it's not, then the `else` block is executed and it will print ODD. In this example, since `x` is 14 and its even, the `if` block is executed.

```
[3  
1]: 1 x = 9
```

```
2
3     print("---- before if block ----")
4
5     if x % 2 == 0:
6         print("EVEN")
7     else:
8         print("ODD")
9
10    print("---- after if block ----")
```

```
[3
1]:      ---- before if block ----
          ODD
      ---- after if block ----
```

In cell 31, since `x = 9` is an odd number, the `if` check fails and `else` block is executed. It prints `ODD`.

```
1     # if x is integer, print "INTEGER"
2     # else print "NOT INTEGER"
3
4     # for integer check -> type()
5
6     x = 9
7
8     if type(x) == int:
9         print("INTEGER")
10    else:
11        print("NOT INTEGER")
```

```
[3
2]:      INTEGER
```

In cell 32, we check if the type of variable `x` is integer (`int`) or not. Since its value is 9, which is an integer, the `if` check returns `True` and the code line 9 is executed. And it prints `INTEGER`.

```
[3] 1 # lets define a string (str) variable
     2 # and check its type
     3 x = "xyz"
     4
     5 if type(x) == int:
     6     print("INTEGER")
     7 else:
     8     print("NOT INTEGER")
```

```
[3] NOT INTEGER
```

In cell 33, `x` is defined as a string variable, “xyz”. And since its type is `str` now, the conditional check in the `if` block fails: `type(x) == int`. So the `else` block is executed. And the line 8 prints `NOT INTEGER`.

True / False - else

For each questions below, decide whether it's true or false.

Q1:

The code below will print **ODD**:

```
x = 17
if x % 2 == 0:
    print("ODD")
else:
    print("EVEN")
```

- True
- False

Q2:

Value of **x** will be **6** after executing the code below:

```
x = 5
if False:
    x -= 1
else:
    x += 1
```

- True
- False

SOLUTIONS - True / False - else

Here are the solutions for True / False questions.

S1:

The code below will print **ODD**:

```
x = 17
if x % 2 == 0:
    print("ODD")
else:
    print("EVEN")
```

- True
- False

It's **false**. It writes "ODD" when the number is even, and "EVEN" when it is odd :)

S2:

Value of **x** will be **6** after executing the code below:

```
x = 5
if False:
    x -= 1
else:
    x += 1
```

- True
- False

It's **true**. **if False** statement will never run, so the **else** block will be executed and **x** will be incremented by 1.

elif

So far, we only checked for two conditions:

- if
- else

What if we need more than two conditions to check? That's where the **elif** statement comes in. The **elif** statement is used to check multiple conditions one by one. We can place as many **elif** conditions as necessary between the **if** condition and the **else** condition. So basically the structure is **if-elif-elif-...-else**.

The best way to understand the **if-elif-else** structure is by doing exercises. As the first example, we will ask the user to enter a number. Then we will check if this number is either a positive number, zero or a negative number.

```
[3
4]: 1 # ask the user to enter a number
      2 # and check if the input is 'POSITIVE', 'ZERO',
      'NEGATIVE'
      3
      4 def is_positive():
      5
      6     # input() returns str
      7     user_input = input("Please enter a number: ")
      8
      9     # cast input into int -> int()
     10     n = int(user_input)
     11
```

```
12     # check conditions
13     if n > 0:
14         print("POSITIVE")
15     elif n == 0:
16         print("ZERO")
17     else:
18         print("NEGATIVE")
```

We define a function named `is_positive` in cell 34. It has 3 conditions to check. The `if` statement will check whether it is positive or not. The `elif` will check whether it is 0 or not. And finally, the `else` will accept any other possibilities. When none of the `if` and `elif` statements are `True`, the `else` block runs.

Now let's call the function and enter 3 different numbers:

```
[3
5]: 1 is_positive()
```

```
[3
5]: Please enter a number: -8
      NEGATIVE
```

```
[3
6]: 1 is_positive()
```

```
[3
6]: Please enter a number: 25
      POSITIVE
```

```
[3
7]: 1 is_positive()
```

```
[3
7]: Please enter a number: 0
      ZERO
```

Important Note: You cannot use `elif` without `if`. You will get `SyntaxError` if you try to use an `elif` without the corresponding `if` statement:

```
[3
8]: 1 a = 1
      2
      3 # write elif without if
      4 elif a == 1:
      5     print("a is 1")
      6 else:
      7     print("not 1")
```

```
[3
8]: File "<ipython-input-38-5d7d570f4f1f>", line 4
      elif a == 1:
      ^
SyntaxError: invalid syntax
```

Exercise:

Let's say we want to find which day of the week, based on the user input. We will define a function named `day_of_week`. It will ask for a number between 1 and 7. And it will decide the day of the week by using this number.

```
[3
9]: 1     def day_of_week():
      2
      3         # ask for number of the day
      4         number_of_day = int(input("Please enter a
      5         number (1-7): "))
      6         # define a variable for name of the day
```

```
7     name_of_day = ""
8
9     # conditions
10    if number_of_day == 1:
11        name_of_day = 'MONDAY'
12    elif number_of_day == 2:
13        name_of_day = 'TUESDAY'
14    elif number_of_day == 3:
15        name_of_day = 'WEDNESDAY'
16    elif number_of_day == 4:
17        name_of_day = 'THURSDAY'
18    elif number_of_day == 5:
19        name_of_day = 'FRIDAY'
20    elif number_of_day == 6:
21        name_of_day = 'SATURDAY'
22    else:
23        name_of_day = 'SUNDAY'
24
25    # return name of day
26    return name_of_day
```

The function definition is in cell 39. We ask for the user input with the `input()` function in line 4. And we convert the input to an integer via the `int()` function. We define a variable named `name_of_day` to keep the name. You can see the `if-elif` statements for all the possibilities. If none of the conditions are `True`, then the `else` statement in line 22 will execute.

Now let's call the function:

```
[4
0]: 1 day_of_week()
```

[4
0]:

Please enter a number (1-7): 4

'THURSDAY'

[4
1]:

1 day_of_week()

[4
1]:

Please enter a number (1-7): 7

'SUNDAY'

[4
2]:

1 day_of_week()

[4
2]:

Please enter a number (1-7): 0

'SUNDAY'

The **day_of _week** functions seems working fine, but it has bugs. You may see a bug in cell 42. The user input is 0, but the function prints as SUNDAY, which definitely not true. Here are some of the bugs we should fix.

What if the user:

- enters a letter instead of a number
- enters a number greater than 7
- enters a number less than 1

[4
3]:

1 *# a letter instead of a number*

2 day_of_week()

[4
3]:

Please enter a number (1-7): A

```
ValueError: invalid literal for int() with base 10:  
'A'
```

As you see in cell 43, we get **ValueError** when the user input is not a number. The error occurs in line 4, where we try to convert the input into an integer with **int()** function.

```
[4  
4]: 1 # a number greater than 7  
2 day_of_week()
```

```
[4  
4]: Please enter a number (1-7): 99  
'SUNDAY'
```

In cell 44, the user input is 99 but the function prints SUNDAY, which is not true.

```
[4  
5]: 1 # a number less than 1  
2 day_of_week()
```

```
[4  
5]: Please enter a number (1-7): -6  
'SUNDAY'
```

Another bug is in cell 45. The user input is a negative number, -6, but the function prints SUNDAY. Actually any number out of the range 1 to 6 will print SUNDAY, because it's the **else** statement.

Let's refactor the code and fix the bugs in our function.

```
[4  
6]: 1 def day_of_week():  
2     # ask for number of the day
```

```
3      # check if its numeric -> cast into int
4      # input() -> returns str
5      user_input = input("Please enter a number (1-
6      7): ")
7
8      # CONTROL 1 - BEING INTEGER
9      if user_input.isdigit():
10         number_of_day = int(user_input)
11     else:
12         print("Please enter an integer.")
13         # end the program -> return
14         return
15
16
17      # CONTROL 2 - BEING BETWEEN 1-7
18      if number_of_day < 1 or number_of_day > 7:
19          print("Please enter a number between 1-
20          7.")
21          # end the program -> return
22          return
23
24
25      # define a variable for name of the day
26      name_of_day = ""
27
28      # conditions
29      if number_of_day == 1:
30          name_of_day = 'MONDAY'
31      elif number_of_day == 2:
32          name_of_day = 'TUESDAY'
```

```

33     elif number_of_day == 3:
34         name_of_day = 'WEDNESDAY'
35     elif number_of_day == 4:
36         name_of_day = 'THURSDAY'
37     elif number_of_day == 5:
38         name_of_day = 'FRIDAY'
39     elif number_of_day == 6:
40         name_of_day = 'SATURDAY'
41     else:
42         name_of_day = 'SUNDAY'
43
44     # return name of day
45     return name_of_day

```

In cell 46, we redefine the `day_of_week` function. We first check if the input is a number. The user input is assigned to a variable named `user_input`. Then in line 8, we use a string method, `isdigit()`. The `isdigit()` function returns `True` if the `user_input` can be converted into a number. If it's a numeric value, then we can cast it into an integer in line 9: `number_of_day = int(user_input)`. If `user_input.isdigit()` returns `False`, which means the input is not a number, then the `else` statement will execute and it will print as "Please enter an integer.". And it will exit the function, because we have the `return` statement in line 13.

If the `user_input` is a number, we will have an `int` variable named `number_of_day`, the code will continue to run and it will print "----- CONTROL #1 is SUCCESSFUL -----" (line 15). Then we will have the second check, to see if the

`number_of_day` variable is out of the range of 1-7. The `if` statement in line 18 will check this: `number_of_day < 1` or `number_of_day > 7`. If it's out of this range then Python will print a text saying "Please enter a number between 1-7." and it will exit the function via the `return` statement (line 21).

If it all goes well, which means the user input is a number and it's in range 1-7, then Python will print the text saying "----- CONTROL #2 is SUCCESSFUL -----" in line 23.

In line 26, we define a string variable `name_of_day` to keep the name. The rest is a straightforward `if-elif-else` structure to check any possible value. And in line 45, we return the `name_of_day` variable.

Now that our function is bug free, at least for the moment, let's test it against the cases we stated before:

```
[4 7]: 1 # test for being a letter instead of a number
2 day_of_week()
```

```
[4 7]: Please enter a number (1-7): A
Please enter an integer.
```

We passed the first test. Let's try the second case:

```
[4 8]: 1 # test for a number greater than 7
2 day_of_week()
```

```
[4 8]: Please enter a number (1-7): 88
```

```
----- CONTROL #1 is SUCCESSFUL -----  
Please enter a number between 1-7.
```

As you see in cell 48, it passes the first check but fails at the second one, because the input is not in the expected range. So our second test is successful. Now let's do a test for the third case:

```
[4  
9]: 1 # test for a number less than 1  
2 day_of_week()
```

```
[4  
9]: Please enter a number (1-7): -4  
Please enter an integer.
```

And we see that we can handle the last case too. For now, our function seems working correctly. Let's also call with a valid day number.

A note on `isdigit()` function: The `isdigit()` function doesn't take negative numbers into account. It assumes negative numbers as strings, not numbers. So if you run it with a negative number in quotes like: `“-4”.isdigit()`, you will get **False**.

```
[5  
0]: 1 # call with a valid day number  
2 day_of_week()
```

```
[5  
0]: Please enter a number (1-7): 6  
----- CONTROL #1 is SUCCESSFUL -----  
----- CONTROL #2 is SUCCESSFUL -----  
'SATURDAY'
```

In cell 50, we call the function with number 6. It passes both checks and returns SATURDAY. Which means our function works as expected and it's bug-free.

OceanofPDF.com

True / False - elif

For each questions below, decide whether it's true or false.

Q1:

When the code below runs and the user enters **0** (zero), it will print "ZERO" on the screen.

```
def is_negative():
    user_input = input("Please enter a number: ")

    if not user_input.isdigit():
        return
    else:
        n = int(user_input)

        if n <= 0:
            print("NEGATIVE")
        elif n == 0:
            print("ZERO")
        else:
            print("POZITIVE")
```

- True
- False

Q2:

The code below will print TRUE:

```
name = "Python"

elif name == "Pyrhon":
    print("FALSE")
elif name == "Python":
    print("TRUE")
else:
    print("NOT SURE")
```

- True
- False

SOLUTIONS - True / False - elif

Here are the solutions for True / False questions.

S1:

When the code below runs and the user enters **0** (zero), it will print "ZERO" on the screen.

```
def is_negative():
    user_input = input("Please enter a number: ")

    if not user_input.isdigit():
        return
    else:
        n = int(user_input)

        if n <= 0:
            print("NEGATIVE")
        elif n == 0:
            print("ZERO")
        else:
            print("POZITIVE")
```

- True
- False

It's **false**. The first condition is "**if n <= 0**". It will evaluate **True** if the user enters **0**. So it will print "NEGATIVE".

S2:

The code below will print TRUE:

```
name = "Python"

elif name == "Pyrhon":
    print("FALSE")
elif name == "Python":
    print("TRUE")
else:
    print("NOT SURE")
```

- True

- False

It's **false**. You cannot use **elif** without **if**. So it will throw a **SyntaxError**.

OceanofPDF.com

Nested Conditionals

In Python we can nest one conditional within another. Nesting is simply putting one `if` block inside of another `if` block. Nested conditionals enables us to make complex decisions based on different scenarios.

Before dive into the examples on nested conditionals, there is a logical operator which we skipped so far. It's the `not` operator, and now it's time to see how it works.

`not`:

`not` operator turns a boolean (`True` or `False`) into its opposite. Let's see with examples.

```
[5  
1]: 1 not True
```

```
[5  
1]: False
```

```
[5  
2]: 1 not False
```

```
[5  
2]: True
```

As you see in cell 51, `not True` is equal to `False`. And in cell 52, `not False` is equal to `True`.

```
[5  
3]: 1 x = 5  
2 x == 5
```

[5
3]: True

In cell 53, we define a integer variable, `x`, and assign the value 5 to it. Then in line 2, we check its equality with number 5. The result `True` because its value is 5. Now let's use the `not` operator with this equality check. It will negate the result of the equality check.

[5
4]: 1 `not` `x == 5`

[5
4]: False

Let's see how we use the `not` operator with `if` statements.

[5
5]: 1 *# if statement in the usual way*
2
3 `if` `x == 5:`
4 `print('x is 5')`
5 `else:`
6 `print('x is not 5')`

[5
5]: x is 5

In cell 55, we use the `if` statement to check if the value of variable `x` is 5 or not. And since its value is 5, the code line 4 is executed.

[5
6]: 1 *# not in if statement*
2

```
3 if not x == 5:  
4     print("x is not 5")  
5 else:  
6     print('x is 5')
```

```
[5]  
[6]: x is 5
```

In cell 56 we set the `if` logic in the opposite way, with the `not` operator. We first checked if the value of `x` is not 5 as: `if not x == 5`. Since this check fails, the code in the `else` block is executed. In both cells, 55 and 56, we get the same result but in the opposite ways.

Now we are ready for the examples of nested conditionals. In the first example we will have two numbers and we will check if they are equal or not. If they are not equal then we want to see which one is bigger.

```
[5]  
[7]: 1 # Example:  
2  
3 x = 300  
4 y = 40  
5  
6 if x == y:  
7     print("x is equal to y")  
8 else:  
9     # x is not equal to y  
10    if x > y:  
11        print("x is greater than y")  
12    elif x == y:  
13        print("x is equal to y")
```

```
14     else:  
15         print("x is less than y")
```

```
[5  
7]:      x is greater than y
```

In cell 57, we define two variables `x` and `y`, with values 300 and 40 respectively. In line 6, we first check if they are equal with the `if` statement. Since they are not, Python runs the code in the `else` block. Inside the `else` block we have a nested `if` statement. This time it checks if `x` is greater than `y` as: `if x > y`. If this is `False`, then it moves to line 12, to the `elif` block. The `elif` check is whether they are equal or not. And since this is `False`, Python runs the code in the `else` block.

You might have noticed an issue in cell 57. We repeat a code segment two times which is the equality check (`x == y`). Since we check their equality in line 6, we do not need the check in line 12, which is the `elif` block. We can remove the `elif` block, because it's redundant. Let's do it as a new example in cell 58:

```
[5  
8]:  
1     # Example:  
2  
3     x = 10  
4     y = 50  
5  
6     if x == y:  
7         print("x is equal to y")  
8     else:  
9         # x is not equal to y  
10        if x > y:
```

```
11     print("x is greater than y")
12     # redundant
13     # elif x == y:
14         # print("x is equal to y")
15     else:
16         print("x is less than y")
```

[5
8]: x is less than y

In the next example, we will ask for a number from the user. Then we will make two decisions. If the number is less than 10 and if it's an odd or even number. We will define a function, **odd_or_even**, for this task. Let's start:

```
[5
9]: 1     # Example:
2     # ask a number from user
3
4     # if the number is less than 10:
5         # if odd -> ODD LESS THAN 10
6         # if even -> EVEN LESS THAN 10
7
8     # if it is greater than 10
9         # if odd -> ODD
10        # if even -> EVEN
11
12    def odd_or_even():
13        # ask for input
14        user_input = input("Please enter a number: ")
15
16        # input() -> str -> cast to int()
17        num = int(user_input)
18
```

```
19      # conditions
20      if num < 10:
21          # odd or even
22          if num % 2 == 1:
23              print("ODD LESS THAN 10")
24          else:
25              print("EVEN LESS THAN 10")
26      else:
27          # odd or even
28          if num % 2 == 1:
29              print("ODD")
30          else:
31              print("EVEN")
```

In the outer `if` block we check if the number is less than 10. If it is, then we check either it's odd or even in the nested `if` block. We have another odd-even check inside the `else` block. Now let's call the function and pass different numbers to see the result:

```
[6
0]: 1 odd_or_even()
```

```
[6
0]: Please enter a number: 7
      ODD LESS THAN 10
```

```
[6
1]: 1 odd_or_even()
```

```
[6
1]: Please enter a number: 4
      EVEN LESS THAN 10
```

```
[6
2]: 1 odd_or_even()
```

[6
2]: Please enter a number: 10
EVEN

[6
3]: 1 odd_or_even()

[6
3]: Please enter a number: 87
ODD

OceanofPDF.com

True / False - Nested Conditionals

For each questions below, decide whether it's true or false.

Q1:

The code below will print **K2** :

```
y = 5
if not y != 5:
    print("K1")
else:
    print("K2")
```

- True
- False

Q2:

The code below will print **A != B** :

```
a = 3
b = 5
if a != b:
    print("A != B")
else:
    if a > b:
        print("A > B")
    else:
        print("A < B")
```

- True
- False

SOLUTIONS - True / False - Nested Conditionals

Here are the solutions for True / False questions.

S1:

The code below will print **K2** :

```
y = 5
if not y != 5:
    print("K1")
else:
    print("K2")
```

- True
- False

It's **false**. Since "**not y != 5**" evaluates to **True**, it will print **K1**.

S2:

The code below will print **A != B** :

```
a = 3
b = 5

if a != b:
    print("A != B")
else:
    if a > b:
        print("A > B")
    else:
        print("A < B")
```

- True
- False

It's **true**. "**a != b**" will evaluate to **True** so, the first **if** statement will execute.

Recursion

From the previous chapters, we know that a function can call other functions. It can also call itself, which we call Recursion. For example, consider a function that prints numbers from a given number up to 0 in reverse order. This function can achieve this task, by calling itself recursively.

Important Note: In Recursion, you have to set an **exit condition**. In other words, the function should know where to stop. Otherwise, it will call itself up to infinity, which we call infinite recursion.

The best way to understand Recursion is to see different examples. Let's start with a simple one. We will define a function with name **count_down** and it will print the numbers backwards.

```
[6
4]: 1 # Example:
2
3 def count_down(n):
4     """
5     The countdown function to print numbers.
6     Parameters: int n, the number to print
7     The function calls itself with the next number.
8     """
9
10    # exit condition
11    if n <= 0:
12        print("----- End of Program -----")
13        return
```

```
14     else:  
15         print(n)  
16         # call itself  
17         count_down(n-1)
```

The **count_down** function first checks if the number **n** is 0 in line 11. This is the exit condition. If it's 0, then it will print a text of "----- End of Program -----" and it will exit via the **return** statement in line 13. If the number **n** is bigger than 0, the **else** block will execute. First it will print the number, then it will call itself with **n-1** in line 17. And the next **count_down** function will do exactly the same thing with **n-1**. It will continue in that way, until **n** becomes 0.

Let's call our recursive function with a value of 5.

```
[6  
5]: 1 count_down(5)  
  
[6  
5]:  
5  
4  
3  
2  
1  
----- End of Program -----
```

Let's call it with a bigger number, with number 10:

```
[6  
6]: 1 count_down(10)  
  
[6  
6]:  
10  
9
```

```
8
7
6
5
4
3
2
1
----- End of Program -----
```

As you see in cells 65 and 66, it starts printing the first number, 10, which is the parameter value you call the function with for the first time. Then it decreases one by one and prints them all. Until it hits the number 0.

Let's do another example. This time we will define a function with name `write_text`. It will have two parameters, a `text` and a number `n`. And it will print the text `n` times recursively. At each call to itself, it will decrease the number `n` by 1.

```
[6
7]: 1      # Example:
      2      # create a function which will print the text
              # parameter given to it.
      3      # The function will write this text number n
              # times.
      4      # n is second parameter
      5
      6      def write_text(text, n):
              # exit condition
      7      if n <= 0:
              return
      8      else:
```

```
11     print(text)
12     write_text(text, n-1)
```

The recursion is in line 12, where it calls itself with the same **text** parameter and **n-1**. Let's call it:

```
[6
8]: 1 write_text('Hands-On Python is great :)', 5)
```

```
[6
8]: Hands-On Python is great :)
      Hands-On Python is great :)
```

Let's modify our **write_text** function a bit. We want to print the numbers too.

```
[6
9]: 1 # let's print numbers
2
3 def write_text(text, n):
4     # exit condition
5     if n <= 0:
6         return
7     else:
8         print("{} : {}".format(n, text))
9         write_text(text, n-1)
```

We add the number **n** in the print statement in line 8. We use **string.format()** function with empty placeholders ({}). The first placeholder is for the number **n** and the second one is for the **text**.

Finally the function calls itself in line 9 with `text` and `n-1`. And here is the result:

```
[7 0]: 1 write_text('Hands-On Python is great :)', 5)
```

```
[7 0]: 5 : Hands-On Python is great :)
4 : Hands-On Python is great :)
3 : Hands-On Python is great :)
2 : Hands-On Python is great :)
1 : Hands-On Python is great :)
```

Let's assume we do not set a proper exit condition. As we stated earlier the function will call itself infinitely many times and Python will terminate the code. You may also get **RecursionError** depending your development environment.

Let's define a function with no exit condition:

```
[7 1]: 1 def up_to_hundred(n):
2
3     # no exit condition
4
5     print(n)
6     up_to_hundred(n+1)
```

If we call the `up_to_hundred` function with the definition in cell 71, JupyterLab will terminate it after some time. Otherwise, it will never stop. Let's redefine it and add an exit condition:

```
[7 2]: 1 def up_to_hundred(n):
2
```

```
3  # exit condition
4  if n > 100:
5      return
6  else:
7      print(n)
8      up_to_hundred(n+1)
```

Now if you call it, it will print the numbers from 80 to 100 successfully.

```
[7
3]: 1 up_to_hundred(80)
```

```
[7
3]: 80
81
82
...
...
99
100
```

True / False - Recursion

For each questions below, decide whether it's true or false.

Q1:

When a function calls other functions (not itself), this is called **Recursion**.

- True
- False

Q2:

If you do not set a proper **exit condition** the function will call itself up to infinity.

- True
- False

SOLUTIONS - True / False - Recursion

Here are the solutions for True / False questions.

S1:

When a function calls other functions (not itself), this is called **Recursion**.

- True
- False

It's **false**. **Recursion** is term used when the function calls itself.

S2:

If you do not set a proper **exit condition** the function will call itself up to infinity.

- True
- False

It's **true**. In Recursion, function will call itself up to infinity if there is no proper exit condition.

QUIZ - Conditional Statements

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_Conditional_Statements.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *8_Conditional_Statements*. Here are the questions for this chapter:

QUIZ - Conditional Statements:

Q1:

Ask the user to enter a number.

Find out if this number is Odd or Even.

```
[  
1 1 # Q 1:  
]:  
2  
3 # --- your solution here --- #
```

```
[  
1 Please enter a number: 8  
]:  
8 is EVEN
```

Q2:

Ask for a letter from the user.

Find out if it is a vowel or consonant.

```
[  
2 1 # Q 2:  
]:  
2  
3 # --- your solution here --- #
```

```
[  
2 Please enter a letter: u  
]:
```

VOWEL: u

Q3:

Define a function with name **which_shape**.

The function will ask a number from the user.

It will decide the name of the shape according to that number, and return that name.

Sides & Shapes:

- 3 - Triangle
- 4 - Rectangle
- 5 - Pentagon
- 6 - Hexagon
- 7 and more - Polygon

```
[  
3 1 # Q 3:  
]:  
2  
3 # --- your solution here --- #  
4  
5 # call the function you defined  
6 shape = which_shape()  
7 print(shape)
```

```
[  
3 Please enter the number of sides: 4  
]:  
 Rectangle
```

Q4:

Define a function to print the number of days in the given month.

Function name will be **number_of_days_in_month**.

It will ask month name from the user.

Number of Days in Gregorian calendar:

- January - 31

- February - 28 or 29
- March - 31
- April - 30
- May - 31
- June - 30
- July - 31
- August - 31
- September - 30
- October - 31
- November - 30
- December - 31

Expected Output:

Please enter the month name: October

'There are 31 days in October'

```
[  
4 1 # Q 4:  
]:  
2  
3 # --- your solution here --- #  
4  
5 # call the function you defined  
6 number_of_days_in_month()
```

```
[  
4 Please enter the month name: February  
]:  
'There are 28 or 29 days in February'
```

Q5:

Define a function to ask the length of sides of a triangle.

Function name is **name_the_triangle**.

The function will decide the type of triangle, according to the length of the sides.

And will return its name.

Types of Triangles:

- Equilateral: three equal sides
- Isosceles: two equal sides
- Scalene: no equal sides

```
[5]: 1 # Q 5:
      2
      3 # --- your solution here --- #
      4
      5 # call the function you defined
      6 print(name_the_triangle())
```

```
[5]: Length for the first side: 10
      Length for the second side: 45
      Length for the third side: 20
      Scalene
```

Q6:

Define a function that decides what to wear, according to the month and number of the day.

Function name will be **what_to_wear** and it will ask for the name of the month and day number.

According to month and day number, it will check for the season and with season, it will return what to wear.

There are 4 months which are in between the seasons; March, June, September and December. For these months, the first 15 days will be in the previous season. For example for March, it will be Winter until 15th. And it will be Spring after the 15th.

Seasons and Garments:

- Spring: Shirt
- Summer: T-shirt
- Autumn: Sweater

- Winter: Coat

Expected Output:

Enter month name: March

Enter day number: 24

In March, day 24 we wear Shirt

```
[  
6 1 # Q 6:  
]:  
2  
3 # --- your solution here --- #  
4  
5 # call the function you defined  
6 what_to_wear()
```

```
[  
6 Enter month name: September  
]:  
Enter day number: 8  
In September, day 8 we wear T-shirt
```

Q7:

Define a function named **which_day**.

It will ask for any of the capital letters: F, M, S, T, W.

These are the first letters of the day names in a week.

The function will decide which day by the first letter.

It may ask additional questions if needed?

Days: Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday

Additional questions by letter:

S:

- "first day of weekend" or
- "last day of weekend"

T:

- "at the beginning of the week" or
- "at the middle of the week"

Expected Output:

Please enter any of the capital letters: F, M, S, T, W : T

Is it at the beginning of the week? Yes - No? No

Your day: THURSDAY

```
[  
7 1 # Q 7:  
]:  
2  
3 # --- your solution here --- #  
4  
5 # call the function you defined  
6 which_day()
```

```
[  
7 Enter one of: F, M, S, T, W : T  
]:  
Is it at the beginning of the week? Yes - No : No  
Your day: THURSDAY
```

Q8:

Define a function named **odds_evens**.

It will ask for a positive number from the user.

Check if the input is an integer or not.

Also check if the number is a positive number.

Decide if the number is ODD or EVEN and print:

- If the number is odd -> ODD
- If the number is even and between 1 and 10 (both included) -> EVEN of TEN
- If the number is even and between 11 and 20 (both included) -> EVEN of TWENTY
- If the number is even and greater than 20 -> EVEN

Expected Output:

```
Please enter a number: 18
EVEN of TWENTY
```

```
[ 8 1 # Q 8:
]: 2
3 # --- your solution here --- #
4
5 # call the function you defined
6 odds_evens()
```

```
[ 8 Please enter a number: 18
]: EVEN OF TVENTY
```

Q9:

Sum of Odds and Evens:

If the sum of two numbers is Odd then only one of these is Odd.

If the sum of two numbers is Even then both are Odd or both are Even.

Define a function named **sum_of_odds_and_evens**.

It will ask for two integers from the user.

Then it will decide if the sum is Odd or Even.

Important Note: Do not calculate the actual summation as $x + y$.

Expected Output:

First Number: 8

Second Number: 3

Sum is ODD

```
[ 9 1 # Q 9:
]: 2
3 # --- your solution here --- #
```

```
4
5 # call the function you defined
6 sum_of_odds_and_evens()
```

```
[9]: Number One: 8
]: Number Two: 3
      Sum is ODD
```

Q10:

Define a function named **count_down**.

It will ask for two integers from the user.

First it will decide which one of them is small and which one is large.

Then it will print the numbers starting from the large one up to the small one (backwards).

Hints:

- Solve with Recursion.
- There may be a second function as **print_down** for Recursion.

```
[10]: 1 # Q 10:
2
3 # --- your solution here --- #
4
5 # call the function you defined
6 count_down()
```

```
[10]: Number One: 26
      Number Two: 12
      small 12
      large 26
      26
```

25

24

...

...

13

12

OceanofPDF.com

SOLUTIONS - Conditional Statements

Here are the solutions for the quiz for Chapter 8 - Conditional Statements.

SOLUTIONS - Conditional Statements:

S1:

```
[  
1 1      # S 1:  
]:  
2  
3      # ask for number  
4 num = int(input("Please enter a number: "))  
5  
6      # decide  
7 if num % 2 == 0:  
8     print(num, "is EVEN")  
9 else:  
10    print(num, "is ODD")
```

```
[  
1      Please enter a number: 8  
]:  
8 is EVEN
```

S2:

```
[  
2 1  # S 2:  
]:  
2  
3      # ask for a letter  
4 letter = input("Please enter a letter: ")  
5
```

```
6 if letter == 'a' or letter == 'e' or letter == 'i' or letter
7 == 'o' or letter == 'u':
8     print("VOWEL:", letter)
9 else:
10    print("CONSONANT:", letter)
```

[
2
]: Please enter a letter: u

S3:

```
[  
3  
]:  
1      # S 3:  
2  
3      def which_shape():  
4          """  
5              Asks for a number and returns the name of the  
6              shape.  
7          Parameters: None  
8          Returns: str shape_name  
9          """  
10         # get the number  
11         number_of_sides = int(input("Please enter the  
12         number of sides: "))  
13         # define a variable to keep the shape name  
14         shape_name = ""  
15  
16         if number_of_sides == 3:  
17             shape_name = 'Triangle'  
18         elif number_of_sides == 4:  
19             shape_name = 'Rectangle'  
20         elif number_of_sides == 5:
```

```
21         shape_name = 'Pentagon'  
22     elif number_of_sides == 6:  
23         shape_name = 'Hexagon'  
24     else:  
25         shape_name = 'Polygon'  
26  
27     # return the shape name  
28     return shape_name  
29  
30 # call the function you defined  
31 shape = which_shape()  
32 print(shape)
```

[
3
]:

Please enter the number of sides: 4
Rectangle

S4:

```
[  
4 1     # S 4:  
]:  
2  
3 def number_of_days_in_month():  
4     # variable for number of days  
5     days = 31  
6  
7     # ask for the month  
8     month = input("Please enter the month name:  
9     ")  
10  
11     # Now check months with number of days 28-  
12 29, 30 and 31  
11     if month == 'April' or month == 'June' or \  
12         month == 'September' or month ==  
'November':
```

```
13         days = 30
14     elif month == 'February':
15         days = '28 or 29'
16
17     return 'There are ' + str(days) + ' days in ' +
18     str(month)
19
20 # call the function you defined
21 number_of_days_in_month()
```

```
[  
4  
]:
```

Please enter the month name: February

'There are 28 or 29 days in February'

S5:

```
[  
5  
]:
```

```
1     # S 5:
2
3     def name_the_triangle():
4         # variable to keep name
5         name = ""
6
7         # ask for lengths
8         side_1 = float(input('Length for the first side:
9         '))
10        side_2 = float(input('Length for the second
11        side: '))
12        side_3 = float(input('Length for the third side:
13        '))
14
15        # conditions
16        if side_1 == side_2 and side_1 == side_3:
17            name = 'Equilateral'
```

```
15     elif side_1 == side_2 or side_1 == side_3 or
16         side_2 == side_3:
17             name = 'Isosceles'
18     else:
19         name = 'Scalene'
20
21     return name
22
23 # call the function you defined
24 print(name_the_triangle())
```

[
5
]:

Length for the first side: 10

Length for the second side: 45

Length for the third side: 20

Scalene

S6:

```
[  
6 1     # S 6:  
]:  
2  
3     def what_to_wear():
4         # ask for month and day number
5         month = input("Enter month name: ")
6         day = int(input("Enter day number: "))  
7
8         # keep the garment
9         garment = ""  
10
11        # decide what to wear
12        if month == 'January' or month == 'February':
13            garment = 'Coat'
14        elif month == 'March':
15            # March 15
```

```

16     if day <= 15:
17         garment = 'Coat'
18     else:
19         garment = 'Shirt'
20     elif month == 'April' or month == 'May':
21         garment = 'Shirt'
22     elif month == 'June':
23         # June 15
24         if day <= 15:
25             garment = 'Shirt'
26         else:
27             garment = 'T-shirt'
28     elif month == 'July' or month == 'August':
29         garment = 'T-shirt'
30     elif month == 'September':
31         # September 15
32         if day <= 15:
33             garment = 'T-shirt'
34         else:
35             garment = 'Sweater'
36     elif month == 'October' or month ==
37     'November':
38         garment = 'Sweater'
39     elif month == 'December':
40         # December 15
41         if day <= 15:
42             garment = 'Sweater'
43         else:
44             garment = 'Coat'
45
46     print('In {0}, day {1} we wear
47     {2}'.format(month, day, garment))
48
49     # call the function you defined

```

48 what_to_wear()

[
6
]:

Enter month name: September

Enter day number: 8

In September, day 8 we wear T-shirt

S7:

[7]
:

```
1 # S 7:  
2  
3 def which_day():  
4     # ask for a letter  
5     letter = input("Enter one of: F, M, S, T, W :  
6     ")  
7  
8     # check if the letter is in list  
9     if not (letter == 'F' or letter == 'M' or letter  
10    == 'S' or letter == 'T' or letter == 'W'):  
11        print("Wrong letter.")  
12        return  
13  
14    # if the letter is correct  
15    else:  
16        # letter controls  
17        # F  
18        if letter == 'F':  
19            day = 'FRIDAY'  
20        # M  
21        elif letter == 'M':  
22            day = 'MONDAY'  
23        # S  
24        elif letter == 'S':  
25            # ask additional question
```

```

24         first_day_of_weekend = input('Is it the
25             first day of the weekend? Yes - No: ')
26
27             if first_day_of_weekend == 'Yes':
28                 day = 'SATURDAY'
29             else:
30                 day = 'SUNDAY'
31             # T
32             elif letter == 'T':
33                 # ask additional question
34                 beginning_of_week = input('Is it at the
35                     beginning of the week? Yes - No : ')
36
37                 if beginning_of_week == 'Yes':
38                     day = 'TUESDAY'
39                 else:
40                     day = 'THURSDAY'
41             # W
42             elif letter == 'W':
43                 day = 'WEDNESDAY'
44
45             print('Your day:', day)
46
47             # call the function you defined
48             which_day()

```

[7]
:

Enter one of: F, M, S, T, W : T
 Is it at the beginning of the week? Yes - No :
 No
 Your day: THURSDAY

S8:

[
8 1 # S 8:
]:

```
2
3  def odds_evens():
4      n = input("Please enter a number: ")
5
6      # if int
7      if not n.isdigit():
8          print('Not a number')
9          return
10
11     # cast to int
12     n = int(n)
13
14     # Positivity
15     if n <= 0:
16         print('Not Positive.')
17         return
18     # It is positive
19     else:
20         # odd?
21         if n % 2 == 1:
22             print('ODD')
23         else:
24             # even
25             if 1 <= n <= 10:
26                 print('EVEN OF TEN')
27             elif 11 <= n <= 20:
28                 print('EVEN OF TWENTY')
29             elif n > 20:
30                 print("EVEN")
31
32     # call the function you defined
33     odds_evens()
```

]:

EVEN OR TWENTY

S9:

```
[  
9  1      # S 9:  
]:  
2  
3  def sum_of_odds_and_evens():  
4      # get numbers  
5      n1 = int(input("Number One: "))  
6      n2 = int(input("Number Two: "))  
7  
8      # if only one is Odd -> summation ODD  
9      # (n1 is odd and n2 is even)  
10     # (n1 is even and n2 is odd)  
11     if (n1 % 2 == 1 and n2 % 2 == 0) or (n1 % 2  
12     == 0 and n2 % 2 == 1):  
13         print("Sum is ODD")  
14     else:  
15         print("Sum is EVEN")  
16  
17     # call the function you defined  
18     sum_of_odds_and_evens()
```

```
[  
9  
]:
```

Number One: 8

Number Two: 3

Sum is ODD

S10:

```
[1  
0]: 1      # S 10:  
2  
3  def count_down():
```

```
4      # get numbers
5      num1 = int(input("Number One: "))
6      num2 = int(input("Number Two: "))
7
8      # variables for small and large
9      small = 0
10     large = 0
11
12     if num1 <= num2:
13         small = num1
14         large = num2
15     else:
16         small = num2
17         large = num1
18
19     print("small", small)
20     print("large", large)
21
22     # call recursion
23     print_down(small, large)
24
25
26     # recursion function
27     def print_down(end, value):
28
29         # exit condition
30         if value == end:
31             print(value)
32             return
33         else:
34             print(value)
35             print_down(end, value - 1)
36
37     # call the function you defined
```

38 count_down()

[1
0]:

Number One: 26

Number Two: 12

small 12

large 26

26

25

24

...

...

13

12

OceanofPDF.com

9. Functions II

This chapter is the second part of functions. We will learn more about functions and most of the main concepts related to them. Let's start.

Return Value

Most of the time functions return a value after execution. The caller of that function receives this returned value and uses it. There are functions which do not return any value, which we call as void functions.

Important Note: In a function, the code lines below the **return** statement is not executed. The **return** statement is the final line of execution in a function.

As an example, let's define a function with name **area_of_circle**.

```
[  
1 1 import math  
]:  
2  
3 def area_of_circle(radius):  
4     # calculate the area and assign to a variable  
5     a = math.pi * radius**2  
6  
7     # return the area  
8     return a
```

The function receives the radius of the circle as the parameter. It calculates the area of the circle, assigns it to a variable and returns that variable in line 8. Let's call this function and use the returning value.

```
[1]: area = area_of_circle(4)
[2]: print(area)
```

```
[1]: 50.26548245743669
[2]:
```

In cell 2, we call the function as `area_of_circle(4)`. And we assign the returned value to a variable called `area`. Then we print that variable.

Temporary Variable: In our `area_of_circle` function, the variable `a` is called a **temporary variable**. Its purpose is to keep the area and pass it to the return statement. In general, temp variables are used for debugging purposes.

Actually we don't have use the temp variable in `area_of_circle`. Let's see how:

```
[1]: # function without temp variable
[2]: def area_of_circle_without_temp(radius):
[3]:     # calculate the area and return it
[4]:     return math.pi * radius**2
```

In cell 3, we define a new function with name **area_of_circle_without_temp**. Instead of using a temp variable, we calculate the area and return it in the same line. We will get the same result if we call it with the same argument:

```
[  
4  1 area = area_of_circle_without_temp(4)  
]:  
2 print(area)
```

```
[  
4  50.26548245743669  
]:
```

Actually in the function call in cell 4, the **area** variable is also a temp variable. We don't have to use it just for printing the return value of the function. And in the next cell, let's skip it and call **area_of_circle_without_temp** inside the **print()** function. This approach is more functional. And it's related to Functional Programming.

Functional Programming is a programming paradigm where programs are constructed by applying and composing functions. It means using functions to the best effect for creating clean code. We will not go deeper on Functional Programming, but we will be using its programming approach during this book. I will talk about Functional Programming whenever needed.

```
[  
5  1 # More Functional Approach  
]:
```

```
2 print(area_of_circle_without_temp(4))
```

```
[5]: 50.26548245743669
```

Incremental Development

While developing computer programs, we cannot consider and code everything at once. That's why we have to concentrate on **Incremental Development**. Incremental Development is essential for **debugging**. If you cannot debug every step of your code, and make sure that it works correctly, you cannot be sure about its quality.

Example:

Let's say we are trying to calculate the distance (Euclidean distance) between two points. In Math it is very easy to calculate via Pythagoras' Theorem:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Figure 9-1: Pythagoras' Theorem to calculate the distance between two points

But this single line of equation is not that easy to do in Python. We have to plan every step to do this calculation. Let's see it in a function:

```
[6]: 1 # Step 1: Calculate the distances and print them
```

```
[1]:  
2  
3     def distance(x1, y1, x2, y2):  
4  
5         # first diff of x's  
6         dx = x2 - x1  
7  
8         # diff of y's  
9         dy = y2 - y1  
10  
11        # <----- DEBUG -----> #  
12        print("dx:", dx)  
13        print("dy:", dy)
```

In cell 6, we start defining our `distance` function. But we do not finish it. We stop just after we calculate the distances. Why do we stop and print the values of `dx` and `dy` instead of finishing it. The answer to this question is vital for code quality. We stop here, to be able to `debug` the code. We want to make sure our code calculates the distances correctly. Let's call the function and see if it works as we expect:

```
[  
7  1  distance(1, 6, 4, 10)  
]:  
  
[  
7  dx: 3  
]:  
    dy: 4
```

In cell 7, we call the function with four arguments: `x1`, `y1`, `x2`, `y2` with values `1, 6, 4, 10` respectively. And difference between

the `x` values is $4 - 1 = 3$ and the `y` values is $10 - 6 = 4$. And these are the values that get printed when we call the function. Which means our function calculates the distances correctly. Let's move on to the next step.

```
[8]: 1 # Step 2: Calculate the sum of squares and print
     it
2
3 def distance(x1, y1, x2, y2):
4
5     # first diff of x's
6     dx = x2 - x1
7
8     # diff of y's
9     dy = y2 - y1
10
11    # <----- DEBUG -----> #
12    # print("dx:", dx)
13    # print("dy:", dy)
14
15    # calculate sum of squares
16    sum_of_squares = dx**2 + dy**2
17
18    # <----- DEBUG -----> #
19    print("sum_of_squares:", sum_of_squares)
```

And let's call the function again to see if the sum of squares is calculated correctly:

```
[9]: 1 distance(1, 6, 4, 10)
]:
```

```
[9]: sum_of_squares: 25  
]:
```

It seems to calculate the sum of squares of the distances correctly. The result is 25. But is this always true? To be sure, let's call the function with another argument set:

```
[1 0]: 1 distance(2, 6, 10, 21)
```

```
[1 0]: sum_of_squares: 289
```

Perfect. Now we are almost sure that our function calculates the sum of squares of the distances correctly. Actually the approach we follow here is called **Test Driven Development (TDD)**. In TDD, you test every function and every step, to be sure that your code works the way you expect.

Now that we test each step by incremental development, let's finalize our function to be able to calculate and return the Euclidean Distance between two points.

```
[1 1]: 1 import math  
2  
3 def distance(x1, y1, x2, y2):  
4  
5     # first diff of x's  
6     dx = x2 - x1  
7  
8     # diff of y's  
9     dy = y2 - y1
```

```

10
11      # <----- DEBUG -----> #
12      # print("dx:", dx)
13      # print("dy:", dy)
14
15      # calculate sum of squares
16      sum_of_squares = dx**2 + dy**2
17
18      # <----- DEBUG -----> #
19      # print("sum_of_squares:", sum_of_squares)
20
21      # return the distance
22      return math.sqrt(sum_of_squares)

```

Let's call our function with different parameter values to see the result:

```
[1]
2]: 1 distance(1, 6, 4, 10)
```

```
[1]
2]: 5.0
```

```
[1]
3]: 1 distance(2, 6, 10, 21)
```

```
[1]
3]: 17.0
```

Compositions

In its core, Functional Programming is about dividing tasks into pieces and define separate functions for these tasks. Each function is responsible from its own task.

Example:

Let's say we have two points. And let's assume the line segment combining these two points as the radius of a circle. Let's try to calculate the area of the circle having this radius. Instead of doing all the steps in one giant function we will create separate functions for each task and call them when needed.

```
[1] 1      def area_of_circle_combining_two_points(x1,
4]: 2          y1, x2, y2):
3              """
4                  Calculates the area of circle with radius from
5                  point one to point two.
6                  Parameters: int x1, y1 first point, int x2, y2
7                  second point
8                  Returns: The area of circle
9                  """
10
11
12
13
14
15
16      return area
```

As you know, we already defined the `area_of_circle` and the `distance` functions. We tested them and debug each steps of them. So we are sure that they work correctly. In cell 14, we use them in

our new function, `area_of_circle_combining_two_points`.

Let's call it and see if it works:

```
[1 5]: 1 area_of_circle_combining_two_points(1, 6, 4, 10)
```

```
[1 5]: 78.53981633974483
```

Bool Functions: Functions which return either `True` or `False` are called as bool functions.

Since we always need to check if a number is either even or odd, we should define two bool functions for this task. Whenever we need to know if a number is odd or even, we can call one of these functions.

```
[1 6]: 1 # if a number is even or odd
2
3 # even fn
4 def is_even(x):
5     return x % 2 == 0
6
7 # odd fn
8 def is_odd(x):
9     return x % 2 == 1
```

Now that we have our bool functions, let's use them. They will tell us if a given number is either odd or even.

```
[1 7]: 1 is_even(11)
```

```
[1  
7]: False
```

```
[1  
8]: 1 is_odd(11)
```

```
[1  
8]: True
```

```
[1  
9]: 1 is_even(40)
```

```
[1  
9]: True
```

Functions are First-Class Citizens

In Python, Functions are First-Class Citizens, which means you can:

- assign function to variables
- pass functions as parameters to other functions
- reassign functions

Let's see what these actions mean by examples:

```
[2  
0]: 1 # define a function to return the cube of a number  
2 def cube(num):  
3     out = num**3  
4     return out
```

In cell 20, we define a function, `cube`, that takes a numeric parameter and returns its cube. Let's first call it and see if it works

as expected.

```
[2  
1]: 1 cube(5)
```

```
[2  
1]: 125
```

Our function works correctly. Now let's assign this function to a variable:

```
[2  
2]: 1 # assign this function to a variable  
2 q = cube
```

As you see in cell 22, we assign the `cube` function to a variable named `q`, as we do with regular value assignments. The variable `q` is a function now, because we assign a function to it. Let's see its type:

```
[2  
3]: 1 type(q)
```

```
[2  
3]: function
```

As we see in the output of cell 23, its type is `function`. Now let's call it:

```
[2  
4]: 1 # call q  
2 q(5)
```

```
[2  
4]: 125
```

In cell 24, we call the variable `q` as `q(5)`, because it's a function. And it returns the cube of 5.

When we call `q` as `q()`, Python executes the `cube()` function. The variable `q` is nothing but another name for the `cube()` function. This is called **aliasing**. Aliasing happens whenever one variable's value is assigned to another variable. In other words, we give a new name to the `cube` function. Now our function has 2 different names in the memory: `cube` and `q`.

Let's do another example on aliasing:

```
[2] 1 def say_hello(text):  
[5]: 2     print(text)
```

We define a function with name `say_hello`. It takes a string parameter and prints its value. First we will call it and pass an argument.

```
[2] 1 say_hello("Hi there Python")  
[6]:
```

```
[2] 1  
[6]: Hi there Python
```

Now let's assign this function to a variable. And then call the new variable:

```
[2] 1 hello = say_hello  
[7]:
```

```
[2] 1 hello("Hi yourself Developer")  
[8]:
```

[2
8]:

Hi yourself Developer

As you see in cells 27 and 28, we assign the `say_hello` function to a variable named `hello`. Then we call this new variable in the same way we call the function: `hello("Hi yourself Developer")`. And it gives exactly the same result.

Unknown Parameters: `*args`

In some cases, you may not know the actual number of parameters. That's where the special parameter `*args` comes in. It is used as a placeholder for any number of parameters. Let's see examples:

Let's say, we want to define a `summation` function. It's going to return the sum of all the parameters that we pass into it. But there is a problem. We do not know, how many numbers the caller will pass into the function. It can be only two numbers or a hundred numbers to sum up.

[2
9]:

```
1 # summation with unknown parameters
2
3 def summation(*args):
4     print('args:', args)
5     print(type(args))
```

In cell 29, we define a function with a special parameter: `*args`. Since we do not know the exact number of parameters, this `*args` will serve for all the parameters. Before doing the actual

summation let's do a quick debug and print `*args`, to see what it really is and its type.

```
[3 0]: 1 summation(5, 7)
```

```
[3 0]: args: (5, 7)  
<class 'tuple'>
```

In the output of cell 30, we see that the `args` are `(5, 7)` and its type is a `tuple`. To be sure, let's call it with another set of arguments and see the result one more time:

```
[3 1]: 1 summation(5, 7, 1, 4, 3)
```

```
[3 1]: args: (5, 7, 1, 4, 3)  
<class 'tuple'>
```

As you see in cell 31, the `args` are `(5, 7, 1, 4, 3)` and its type is a `tuple`. Perfect. We are now sure that, we can handle any number of parameters with `*args`. And also we can access any single parameter in it.

```
[3 2]: 1 # summation  
2  
3 def summation(*args):  
4     # get the sum of args with built-in sum() fn  
5     summation_result = sum(args)  
6  
7     print(summation_result)
```

Since we are interested in the sum of all parameters in the **args tuple**, we can use the built-in Python function **sum()** to get the summation. And that's what we do in cell 32, line 6. Let's call it and see if it works:

```
[3  
3]: 1 summation(5, 7)
```

```
[3  
3]: 12
```

```
[3  
4]: 1 summation(5, 7, 1, 4, 3)
```

```
[3  
4]: 20
```

We call the **summation()** function with different arguments in cells 33 and 34 and it prints the sum of all the parameters correctly. Let's redefine it and instead of using a temp variable, return directly the **sum()** function result:

```
[3  
5]: 1 # summation  
2  
3 def summation(*args):  
4     return sum(args)
```

```
[3  
6]: 1 sum_of_numbers = summation(5, 7, 1, 4, 3)  
2 sum_of_numbers
```

```
[3  
6]: 20
```

Since we know `args` is a `tuple`, we can loop over it. Let's create a simple loop and print the items in the `args`. We will do this in a function called `print_parameters`.

```
[3  
7]: 1 # print the arguments inside the *args  
2  
3 def print_parameters(*args):  
4     # loop over the arguments  
5     for arg in args:  
6         print(arg)
```

Let's call the `print_parameters` function with some arbitrary arguments:

```
[3  
8]: 1 print_parameters('A', 'B', 45, True, 'Python')
```

```
[3  
8]: A  
B  
45  
True  
Python
```

```
[3  
9]: 1 print_parameters('Book', [1,2,3], ('A', 'B'), True,  
'Python')
```

```
[3  
9]: Book  
[1, 2, 3]  
('A', 'B')  
True  
Python
```

lambda Function

Sometimes, we need to define a function without a name (anonymous function). We use **lambda** keyword for this purpose. In Python, lambda functions are known as **one line functions**.

The syntax of the lambda function is: **lambda arguments: expression**

Let's see examples on how we define lambda functions.

Let's say we want to split the given text into its words. Python has built-in **split()** function for this purpose. First let's see the **split()** function works:

```
[4 0]: 1 text = 'Hi there you Python'  
       2 text.split()  
  
[4 0]: ['Hi', 'there', 'you', 'Python']
```

In cell 40, we call **split()** function on a string and it returns a list to the words in it. Actually, a list of items which are separated with the space character.

Now let's define a **lambda** function to use the **split()** function:

```
[4 1]: 1 # we will define a lambda function to use split()  
       2  
       3 lambda x: x.split()  
  
[4 1]: <function __main__.<lambda>(x)>
```

In cell 41, we define a lambda function as: **lambda** x: x.split(). Here **lambda** is the keyword, x is the parameter and the code after the colon (:) is the function body. In the function body, it will split the parameter x as: x.split().

Now we have a **lambda** function but there is a problem. How will we call it? To be able to call a **lambda** function we have to assign it to a variable.

```
[4
2]: 1 # assign the lambda function to a variable
      2 # to be able to use it
      3
      4 split_text = lambda x: x.split()
```

In cell 42, we assign our lambda function to a variable named **split_text**. Now we can easily call it. Since **lambda** is a function that takes a parameter x, so is the **split_text**.

```
[4
3]: 1 # call the split_text function
      2 # it's just a name for the lambda function
      3 split_text('Hi there you Python')
```

```
[4
3]: ['Hi', 'there', 'you', 'Python']
```

And as expected, the **split_text** function returns a list of words in the given text. Because behind the scenes, that's what we define in the **lambda** function body.

Let's define a new **lambda** function. This time we want to define a function for multiplying two numbers. So the **lambda**

will have two parameters, `x` and `y`. And it will return the multiplication result of them.

```
[4 5]: 1 # Multiplication with lambda function
2
3 multiply = lambda x, y: x * y
```

In cell 45, we define a lambda function and assign it to a variable called `multiply`. Now let's call this `multiply` as a function.

```
[4 6]: 1 multiply(10, 6)
```

```
[4 6]: 60
```

```
[4 7]: 1 multiply(5, 3)
```

```
[4 7]: 15
```

Let's define another lambda function. This time for power operation. The name will be `exponential` and it will take two parameters.

```
[4 8]: 1 # power function with lambda
2
3 exponential = lambda num, p: num**p
```

Let's call it now:

```
[4 9]: 1 exponential(2, 5)
```

```
[4  
9]: 32
```

```
[5  
0]: 1 exponential(4, 3)
```

```
[5  
0]: 64
```

Important Note: We do not need an explicit `return` statement in the `lambda` function. Python executes lambda function body and returns the result automatically.

Functions Returning Functions

You can define a function that returns other functions. Lambda functions are used very commonly as return values. Let's see an example of how a function can return a lambda function.

```
[5  
1]: 1 # a function that returns a lambda function  
2  
3 def multiply_by(n):  
4     """  
5         Generic multiplication function.  
6         Parameter: int n  
7         Returns: lambda a: a * n  
8     """  
9     # return a lambda  
10    return lambda a: a * n
```

The `multiply_by` function in cell 51, returns a `lambda` function. And that `lambda` function takes one argument, `a`, and

returns `a * n`. Here `n` is the parameter of `multiply_by` function. So we will pass an argument `n` to the `multiply_by`, and the `lambda` function will use it as the multiplier. That way, our `multiply_by` will be a generic multiplier of `n`.

Let's pass an argument value of 2 to the `multiply_by` function. Since it will return a lambda function back, we will name the return value as `double_multiplier`.

```
[5 2]: 1 # pass the value 2 to the multiply_by fn
        2 # assign the returned function to a variable
        3 double_multiplier = multiply_by(2)
```

In cell 52, we call the `multiply_by` function with the value of 2 for the parameter `n`. We know that `multiply_by` will return a lambda function. And we also know that, this lambda function will take a parameter `a` and multiply it by 2. So the `double_multiplier` variable is actually a lambda function. Let's print its type and then call it with a value of 15.

```
[5 3]: 1 type(double_multiplier)
        2
[5 3]: 3
```

As you see in the output of cell 53, `double_multiplier` is a function. It is the lambda function that is returned when we call the `multiply_by` function. Now let's call the `double_multiplier` function:

```
[5 4]: 1 double_multiplier(15)
```

```
[5 4]: 30
```

We get the value of 30 when we call the **double_multiplier** function. Why? Because we know that it will multiply anything we pass to it by 2. The value 2 comes from the cell 52. It was the parameter value when we call the **multiply_by** function. Let's call it one more time:

```
[5 5]: 1 double_multiplier(9)
```

```
[5 5]: 18
```

Let's create another multiplier. This time let's make it 3. So we will create a function that multiplies anything with 3. Let's name it as **triple_multiplier**.

```
[5 6]: 1 # pass the value 3 to the multiply_by fn
2 # assign the returned function to a variable
3 triple_multiplier = multiply_by(3)
```

In cell 56, we call the **multiply_by** function with a value of 3. Which means the lambda function which it returns is going to use this value of 3 as its multiplier. And we named the returned function as **triple_multiplier**. Let's call it now.

```
[5 7]: 1 triple_multiplier(8)
```

[5
7]:

24

In cell 57, we call the `triple_multiplier` function with an argument value of 8 and get the result of 24. It multiplies the argument we pass with 3. This value of 3 comes from the cell 56. We passed it when we call the `multiply_by` function.

Let's see the documentation of `triple_multiplier` function. Actually we want to see its code.

[5
8]:

1 triple_multiplier??

[5
8]:

Signature: triple_multiplier(a)

Docstring: <no docstring>

Source: return lambda a: a * n

File: 9_functions_ii\ref\<ipython-input-51-6d3f>

Type: function

As you see in the output of cell 58, the source code for `triple_multiplier` function is: `return lambda a: a * n`. Which proves that it is the lambda function returned from `multiply_by` function. The same is true for `double_multiplier`:

[5
9]:

1 double_multiplier??

[5
9]:

Signature: double_multiplier(a)

Docstring: <no docstring>

Source: return lambda a: a * n

File: 9_functions_ii\ref\<ipython-input-51-6d3f>

Type: function

Let's do a final example on this `multiply_by` function. This time let's pass the value of 5:

```
[6 0]: 1 penta_multiplier = multiply_by(5)
```

Let's call this variable as `penta_multiplier`. Since we passed the value of 5 to the `multiply_by` function, `penta_multiplier` will multiply anything we pass by 5.

```
[6 1]: 1 penta_multiplier??
```

```
[6 1]: Signature: penta_multiplier(a)  
Docstring: <no docstring>  
Source:      return lambda a: a * n  
File:       9_functions_ii\ref\<ipython-input-51-6d3f>  
Type:       function
```

```
[6 2]: 1 penta_multiplier(6)
```

```
[6 2]: 30
```

Nested Functions

In Python we can define functions inside the body of other functions. These functions which we define in the others are called **nested functions**.

As an example we will use nested functions, to check whether a given number is a common multiple of 5 and 8.

```
[6
3]: 1 # 2 nested functions inside the main function
2
3 def is_it_common_multiple(n):
4     """
5         Checks whether the given number is a
6         common multiple of 5 and 8.
7         Parameter: int n
8         Returns: True if its multiplier of 5 and 8,
9         False otherwise
10        """
11
12        # nested function 1 -> 5
13        def multiple_of_five(n):
14            if n % 5 == 0:
15                return True
16            else:
17                return False
18
19        # nested function 2 -> 8
20        def multiple_of_eight(n):
21            if n % 8 == 0:
22                return True
23            else:
24                return False
25
26        # Check both conditions
27        if multiple_of_five(n) and
28            multiple_of_eight(n):
                return True
            else:
                return False
```

In cell 63, we define two nested functions, `multiple_of_five` and `multiple_of_eight`. They are both bool functions and in the `if` block (line 25) we call them. We pass the number `n` as the argument to them. And they return either `True` or `False`. If both return `True`, then this means the number `n` is a common multiple of 5 and 8.

Let's call our `is_it_common_multiple` function and see the results:

```
[6  
4]: 1 is_it_common_multiple(24)
```

```
[6  
4]: False
```

```
[6  
5]: 1 is_it_common_multiple(40)
```

```
[6  
5]: True
```

Mutable vs. Immutable

In Python, everything is an object. And every object has a type. An object can be either **Mutable** or **Immutable** depending on its type.

Immutable: Some types stay as they have assigned. They cannot be mutated. They are rigid bodies, you cannot change any of its parts.

Mutable: Mutable objects are the types which you can change (mutate). You can mutate any of its parts.

Python Built-In Types and Mutability:

Type	Name	Mutable/Immutable
int	Integer	Immutable
float	Float	Immutable
bool	Boolean	Immutable
str	String	Immutable
list	List	Mutable
tuple	Tuple	Immutable
dict	Dictionary	Mutable
set	Set	Mutable

Table 9-1: Types and their mutability

Let's see some examples to understand the mutability concept.

```
[6
6]: 1 # Example:
      2
      3 # IMMUTABLE -> String
      4
      5 # Python
      6 text = 'Tython'
```

In cell 66, we define a string (`str`) variable. As we see in *Table 9-1*, `str` is Immutable. Let's prove this. The value of the variable `text` is misspelled as `'Tython'`. It should be `'Python'`. The first letter is wrong. We will try to fix this.

First we have to understand the concept of **indexing**. Sequence types like `str`, `list`, `tuple` and `range` are **indexed**

collections. Which means you can access their items via indexes (indices). To be able to access any element with the index, we just place a pair of square brackets after the object. And inside the brackets we write the index number, like: **variable_name[index]**.

Important Note: In Python, the indexes are **zero-based**. Which means the first index is **0**.

Now that, we know indexing, let's get the first item in the **text** variable we defined in cell 66.

```
[6] 1 # index -> []
[7] 2 # get the first item in text variable
[7] 3 # first means index = 0
[7] 4
[7] 5 text[0]
```

```
[6]
[7]: 'T'
```

In cell 67, we get the first letter in the **text** variable via the code: **text[0]**. Here, 0 is the index of the first item. And the letter at the index 0 is 'T'.

Since we know how to access the first element in the **text** variable, let's try to modify it:

```
[6] 1 text[0] = 'P'
[8]:
```

```
[6]
[8]: TypeError: 'str' object does not support item
      assignment
```

In cell 68, we get the **TypeError** when we try to modify the **text** variable. And the error description says that we cannot mutate a string (**str**) object. Now comes a question: How will we change this string, if we cannot mutate it? The answer is, we reassign it. Instead of mutating a string we should reassign it. **Reassignment** doesn't mean **Mutation**.

Now, to fix the text variable let's reassign the correct value:

```
[6  
9]: 1 # reassign the text variable  
      2 text = 'Python'
```

Let's print its value to see if the text variable has changed:

```
[7  
0]: 1 print(text)  
      2  
      3 Python
```

As you see, if you want to change any part of a string you should reassign the new value to it. That's the case, because the **str** type is **Immutable**.

What if we want to modify a **Mutable Type**? A **List** for example. Let's see it with an example.

```
[7  
1]: 1 # Example:  
      2  
      3 # MUTABLE -> List  
      4  
      5 my_list = ['a', 'b', 'C', 'd', 'e']
```

```
6 my_list
```

```
[7 1]: ['a', 'b', 'C', 'd', 'e']
```

In cell 71, we define a list variable `my_list` and assign some value to it. Now let's say we want to change one of its items. The capital 'C' which is in the third order should be a small letter 'c'. We will fix it.

To fix it, first we have to find the index of item 'C'. Since indexes are zero based, it is at the index of 2. Let's print the item at `index = 2` and see the result.

```
[7 2]: 1 # get the item at index 2
2 my_list[2]
```

```
[7 2]: 'C'
```

Now we are sure that the item at `index = 2` is 'C'. Let's change it:

```
[7 3]: 1 # mutate the list
2 # change the value at index 2
3 my_list[2] = 'c'
```

Finally let's print `my_list` to see if it has changed.

```
[7 4]: 1 print(my_list)
[7 4]: ['a', 'b', 'c', 'd', 'e']
```

As you see in cell 74, the items in the `my_list` variable has changed. Since `list` is a Mutable type, we are able to change its elements.

Pass by Value Pass by Reference

In the previous section, we learned about Mutable & Immutable Types. Now let's see what will happen, if we pass these types to functions as arguments.

Rule of Thumb:

Immutable Types Pass by Value: If you pass an Immutable Object (`int`, `str`, etc.) as a parameter to a function; only its **duplicate** will pass to the function. Not itself, just a copy of it. This is called **pass by value**.

Mutable Types Pass by Reference: If you pass a Mutable Object (`list`, `dict`, `set`) as a parameter to a function; its **reference** will pass to the function. The object itself will be accessible inside the function. This is called **pass by reference**. Reference means their memory addresses.

```
[7]  
5]: 1      # Example:  
2  
3      # IMMUTABLE  
4      # str  
5  
6      language = 'Python'
```

```
7     print("----- Before passing to function ----- :  
8         ", language)  
9  
10    # function is changing the parameter  
11    def change_language(name):  
12        name = 'Java'  
13  
14    # call function and pass language  
15    change_language(language)  
16  
17    print("----- After passing to function ----- : ",  
18         language)
```

[7
5]:

```
----- Before passing to function ----- : Python  
----- After passing to function ----- : Python
```

In cell 75, we define a `str` variable `language` with the value of `'Python'`. And we print its value in line 7. Then we define a function `change_language` which modify the `name` parameter it takes. Whatever the value it takes as the parameter, it will override it as `'Java'`. And in line 14, we call the function with the `language` variable as the argument: `change_language(language)`. We know that it overrides the parameter it takes. Finally in line 16, we print the `language` variable one more time. Since this print is after the function call, you may expect the value of `language` variable to change as `'Java'`. But that's not the case. As you see in the output of cell 75, the final value of the `language` variable is still `'Python'`. It hasn't changed.

Why does the value of `language` variable stay unchanged? The reason is quite simple. Since it's a string and `str` is an Immutable type, when we pass it to the function as an argument we just pass a copy of it. Not the variable object itself. Remember the **Immutable Types pass by value**.

Let's see one more example with the Immutable Types. Let's try an integer this time. What will happen if we pass an integer to a function and try to change it inside the function body?

[7
6]:

```
1      # Example:  
2  
3      # IMMUTABLE  
4      # int  
5  
6      number = 45  
7      print("----- Before passing to function ----- :  
8      ", number)  
9      # function is changing the parameter  
10     def change_number(number):  
11         number = 1000  
12  
13     # call function and pass number  
14     change_number(number)  
15  
16     print("----- After passing to function ----- : ",  
number)
```

[7
6]:

```
----- Before passing to function ----- : 45  
----- After passing to function ----- : 45
```

In cell 76, we pass an integer to the `change_number` function. And inside the function body we assigned a new value to the incoming parameter. But when we print it as the final line, we see that the value of the variable `number` has not changed. That's because it's an integer and the `int` type is Immutable. So it only passes by value to the function.

Now, let's try a Mutable Type. A `list` for example. We will pass a list variable to a function as an argument and inside the function we will change its value. Let's see.

```
[7]
[7]: 1      # Example:
      2
      3      # MUTABLE
      4      # list
      5
      6      numbers = [1, 2, 3, 4, 5]
      7      print("----- Before passing to function ----- : ",
      numbers)
      8
      9      # function is changing the parameter
      10     def change_numbers(nums):
      11         nums[0] = 'A'
      12         nums[1] = 'B'
      13
      14     # call function and pass numbers
      15     change_numbers(numbers)
      16
      17     print("----- After passing to function ----- : ",
      numbers)
```

[7
7]:

----- Before passing to function ----- : [1, 2, 3,

4, 5]

----- After passing to function ----- : ['A', 'B',

3, 4, 5]

In cell 77, we pass a **list** variable, **numbers**, to the function. And inside the **change_numbers** function we modify the items in the indexes of **0** and **1**. Then we print the **numbers** variable once again after the function call. What we see is, its value has changed. The modification we do in the function body has changed the actual variable itself. That's because the **list** type is **Mutable**, and **Mutable types pass by reference**. Which means when you pass them to a function, you actually pass their references, their actual memory addresses, to the function. And the function can access and modify them.

QUIZ - Functions II

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_Functions_II.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *9_Functions_II*. Here are the questions for this chapter:

QUIZ – Functions II:

Q1:

Many times, we need input from the user.

Define a generic function to ask for user input.

Function name will be **get_input**.

Function will take the input text as parameter and use it while asking for input from the user.

```
[1]: 1      # Q 1:  
      2  
      3      # --- your solution here --- #  
      4  
      5  
      6      # call the function you defined  
      7      # Ask for an integer  
      8      number = get_input("Please enter a number: ")  
      9      print(number)  
     10  
     11      # Ask for the month name  
     12      month = get_input("Please enter month name: ")  
     13      month
```

```
[1]: Please enter a number: 78
```

78

Please enter month name: June
'June'

Q2:

We already know that, **get_input** function in Question 1 returns **str**.

Now we will define a new function named **is_integer**.

It will get a string parameter and will check if its value is integer or not.

It will return **True** if the parameter is integer.

```
[  
2 1      # Q 2:  
]:  
2  
3      # --- your solution here --- #  
4  
5  
6      # call the function you defined  
7  
8      # let's test is_integer function  
9      # first call with a string  
10     input_text = "Please enter a number: "  
11     user_input = get_input(input_text)  
12     is_input_integer = is_integer(user_input)  
13     print(is_input_integer)  
14  
15      # call with an integer  
16     input_text = "Please enter a number: "  
17     user_input = get_input(input_text)  
18     is_input_integer = is_integer(user_input)  
19     print(is_input_integer)
```

[
2
]:

Please enter a number: A

```
False
Please enter a number: -8
True
```

Q3:

Get an integer from the user, using functions in Q1 and Q2.

Which will guarantee that it is an integer.

If the input is not integer, ask recursively.

Function name will be **get_integer**.

Hints:

- Use Recursion

```
[ 1 # Q 3:
]: 2
 3 # --- your solution here --- #
 4
 5
 6 # call the function you defined
 7 # do not enter an integer for first several times
 8 num = get_integer("please enter an integer: ")
 9 print(num)
```

```
[ 3 please enter an integer: asdf
]: please enter an integer: qwerty
  please enter an integer: 78abc
  please enter an integer: 89
  89
```

Q4:

Define a function named **day_of_week**.

The function will ask the user to enter an integer between 1 and 7.

And it will decide the day name according to that number.

Hints:

- use functions in Q3
- check for number being in range 1-7

```
[  
4 1 # Q 4:  
]:  
2  
3 # --- your solution here --- #  
4  
5  
6 # call the function you defined  
7 day_of_week()
```

```
[  
4 Please enter a day number 1-7: asdfg  
]:  
Please enter a day number 1-7: rt  
Please enter a day number 1-7: 5  
'Friday'
```

Q5:

Run the functions below and examine the Stack Trace.

Stack Trace means the execution order of functions.

Make sure you understand the execution order.

```
[  
5 1 # Q 5:  
]:  
2  
3 def n(c):  
4     power = m(c, c)  
5     print(c, power)  
6     return power  
7
```

```
8  def m(x, y):  
9      x = x + 2  
10     return x**y  
11  
12 def p(x, y, z):  
13     summation = x + y + z  
14     sqr = n(summation)**2  
15     return sqr  
16  
17 a = 1  
18 b = a + 1  
19 print(p(a, b+1, a+b))
```

```
[  
5  7 4782969  
]:  
22876792454961
```

Stack Trace:

--- your solution here ---

Q6:

Define a function named **arithmetic_mean**.

The function will get parameters that are not known.

(unknown parameters)

Namely, the number of parameters is not known in advance.

The function will print the arithmetic mean of these parameters.

Hints:

- *args
- Arithmetic Mean = Summation / Number of Elements
- sum()
- len()

```
[  
6  1 # Q 6:  
]:
```

```
2
3 # --- your solution here --- #
4
5
6 # call the function you defined
7 mean = arithmetic_mean(2, 5, 11)
8 print(mean)
```

```
[6]: 6.0
```

Q7:

Define a function named **area_of_circle**.

It will take the radius as parameter and calculates the area of the circle.

The function body should be one line of code.

Hints:

- Area of Circle = $\pi * r^2$

```
[7]: 1 # Q 7:
2
3 # --- your solution here --- #
4
5
6 # call the function you defined
7 area_of_circle(10)
```

```
[7]: 314.1592653589793
```

Q8:

Define a function named **perimeter_of_circle**.

It will take the radius as parameter and calculates the perimeter of the circle.

The function body should be one line of code.

Hints:

- Perimeter of Circle = $2 * \pi * r$

```
[  
8  1 # Q 8:  
]:  
2  
3 # --- your solution here --- #  
4  
5  
6 # call the function you defined  
7 perimeter_of_circle(10)
```

```
[  
8  62.83185307179586  
]:
```

Q9:

Define a function named **area_of_rectangle**.

The parameters will be length (**l**) and width (**w**).

The function body should be one line of code.

Hints:

- Area of Rectangle = $l * w$

```
[  
9  1 # Q 9:  
]:  
2  
3 # --- your solution here --- #  
4  
5
```

```
6 # call the function you defined
7 area_of_rectangle(5, 12)
```

```
[9]: 60
```

Q10:

Define a function to calculate the surface area of a right cylinder.

The name will be **area_of_cylinder** and it will take two parameters.

The first parameter **r** will be the radius of top and base circles.

The second parameter **h** will be the height of cylinder.

Hints:

- cylinder is a combination of two circles and a rectangle
- use functions from previous questions to calculate the areas

```
[1]: 1 # Q 10:
2
3 # --- your solution here --- #
4
5
6 # call the function you defined
7 area_of_cylinder(10, 2)
```

```
[1]: 753.9822368615504
```

SOLUTIONS - Functions II

Here are the solutions for the quiz for Chapter 9 - Functions II.

SOLUTIONS - Functions II:

S1:

```
[1]: 1      # S 1:  
[1]:  
2  
3      def get_input(text):  
4          """  
5              Generic function to get input.  
6              Parameter: str text  
7              Returns: the user input, str  
8          """  
9  
10     # ask for input  
11     user_input = input(text)  
12  
13     return user_input  
14  
15  
16     # call the function you defined  
17     # Ask for an integer  
18     number = get_input("Please enter a number: ")  
19     print(number)  
20  
21     # Ask for the month name  
22     month = get_input("Please enter month name: ")  
23     print(month)
```

```
[1]: Please enter a number: 78
```

]:

78

Please enter month name: June
'June'

S2:

```
[  
2 1      # S 2:  
]:  
2  
3  def is_integer(text):  
4      """  
5          Checks if the text is integer.  
6          Parameter: str text  
7          Returns: True if text is integer  
8      """  
9  
10     # possibilities  
11  
12     # 1 - there may be spaces at the beginning and  
13     # end of string -> str.strip()  
14     text = text.strip()  
15  
16     # 2- There may be +,- signs at the beginning,  
17     # +9, -8 -> str.strip(<character>)  
18     text = text.strip('+-')  
19  
20     # Now we can check if integer -> isdigit()  
21     if text.isdigit():  
22         return True  
23     else:  
24         return False  
25     # call the function you defined
```

```
26
27  # let's test is_integer function
28  # first call with a string
29  input_text = "Please enter a number: "
30  user_input = get_input(input_text)
31  is_input_integer = is_integer(user_input)
32  print(is_input_integer)
33
34  # call with an integer
35  input_text = "Please enter a number: "
36  user_input = get_input(input_text)
37  is_input_integer = is_integer(user_input)
38  print(is_input_integer)
```

```
[2]:
```

Please enter a number: A

```
False
```

Please enter a number: -8

```
True
```

S3:

```
[3]:
```

```
1  # S 3:
2
3  def get_integer(input_text):
4      """
5          Gets integer from user (eventually).
6          Parameters: str input_text
7          Returns: int user_input
8      """
9
10 # ask for input
11 user_input = get_input(input_text)
12
```

```
13     # check if int -> exit condition
14     if is_integer(user_input):
15         return int(user_input)
16     else:
17         # ask again
18         return get_integer(input_text)
19
20
21 # call the function you defined
22 # do not enter an integer for first several times
23 num = get_integer("please enter an integer: ")
24 print(num)
```

```
[3]:
```

```
please enter an integer: asdf
please enter an integer: qwerty
please enter an integer: 78abc
please enter an integer: 89
89
```

S4:

```
[4]:
```

```
1     # S 4:
2
3     def day_of_week():
4
5         # get an integer first
6         day_num = get_integer("Please enter a day
7         number 1-7: ")
8
9         # check if 1-7
10        if not 1 <= day_num <= 7:
11            return "Invalid day number. Not in 1-7."
```

```

12     # If we passed -> 1-7
13
14     if day_num == 1:
15         return 'Monday'
16     elif day_num == 2:
17         return 'Tuesday'
18     elif day_num == 3:
19         return 'Wednesday'
20     elif day_num == 4:
21         return 'Thursday'
22     elif day_num == 5:
23         return 'Friday'
24     elif day_num == 6:
25         return 'Saturday'
26     else:
27         return 'Sunday'
28
29
30 # call the function you defined
31 day_of_week()

```

[
4
]:

Please enter a day number 1-7: asdfg
 Please enter a day number 1-7: rt
 Please enter a day number 1-7: 5
 'Friday'

S5:

[
5
]:

```

1     # S 5:
2
3     def n(c):
4         power = m(c, c)
5         print(c, power)

```

```
6     return power
7
8     def m(x, y):
9         x = x + 2
10        return x**y
11
12    def p(x, y, z):
13        summation = x + y + z
14        sqr = n(summation)**2
15        return sqr
16
17    a = 1
18    b = a + 1
19    print(p(a, b+1, a+b))
```

```
[5]: 7 4782969
      22876792454961
```

S6:

```
[6]: # S 6:
      2
      3     def arithmetic_mean(*args):
      4
      5         # calculate the summation
      6         summation = sum(args)
      7
      8         # number of elements
      9         number_of_elements = len(args) # len ->
length
     10
     11        return summation / number_of_elements
     12
```

```
13
14 # call the function you defined
15 mean = arithmetic_mean(2, 5, 11)
16 print(mean)
```

```
[6]: 6.0
]:
```

S7:

```
[7]: 1 # S 7:
[7]: 2
[7]: 3 import math
[7]: 4
[7]: 5 def area_of_circle(r):
[7]: 6     """Returns the area of circle"""
[7]: 7     return math.pi * r**2
[7]: 8
[7]: 9
[7]: 10 # call the function you defined
[7]: 11 area_of_circle(10)
```

```
[7]: 314.1592653589793
]:
```

S8:

```
[8]: 1 # S 8:
[8]: 2
[8]: 3 import math
[8]: 4
[8]: 5 def perimeter_of_circle(r):
[8]: 6     """Returns the perimeter of circle"""
[8]: 7
```

```
7     return 2 * math.pi * r
8
9
10    # call the function you defined
11    perimeter_of_circle(10)
```

```
[ 8       62.83185307179586
]:
```

S9:

```
[ 9
1  # S 9:
]: 2
3 def area_of_rectangle(l, w):
4     return l * w
5
6
7    # call the function you defined
8    area_of_rectangle(5, 12)
```

```
[ 9       60
]:
```

S10:

```
[1
0]: 1  # S 10:
2
3 def area_of_cylinder(r, h):
4     """
5         Calculates the total surface area of cylinder.
6         Parameters:
7             * r: int radius
8             * h: int height
9         Returns: int total surface area
```

```
10      """
11
12      # first calculate the area of circles (2)
13      circle = area_of_circle(r)
14
15      # lateral surface (rectangle)
16      # Perimeter of circle and height
17      # lateral area = perimeter * height
18
19      # perimeter
20      perimeter = perimeter_of_circle(r)
21
22      # rectangle
23      rectangle = area_of_rectangle(perimeter, h)
24
25      # total area
26      total_area = 2 * circle + rectangle
27
28      return total_area
29
30
31      # call the function you defined
32      area_of_cylinder(10, 2)
```

```
[1
0]: 753.9822368615504
```

10. Loops

This chapter is about loops. A **loop** is an iteration over a sequence. In Python, loops are built via **while** and **for** keywords. Let's learn the loops and some important concepts related to them.

while Loop

The **while** loop is one of the two types of loops we have in Python. The **while** loop executes a set of statements as long as its condition is true. Let's start the examples of the **while** loop.

Example: In this example, we want to print numbers from 10 to 1 in reverse order.

```
[  
1  1      # define a variable  
]:  
2  n = 10  
3  
4  print("----- before While loop -----")  
5  print("n:", n)  
6  
7  # while <condition>:  
8  #      .....  
9  #      .....  
10 #      .....  
11  
12 # the while loop with the condition of "n > 0"  
13 while n > 0:
```

```
14     print(n)
15     n = n - 1
16
17 print("----- after While loop -----")
18 print("n:", n)
```

```
[1]: ----- before While loop -----
n: 10
10
9
8
7
6
5
4
3
2
1
----- after While loop -----
n: 0
```

In cell 1, we see the simple structure of a while loop: `while n > 0`. It checks whether the condition (`n > 0`) is **True**. If it's **True**, then it executes the code in the `while` loop scope. If the condition becomes **False**, then it will exit the loop. Here are the detail steps for this example:

Operation of While Loop:

The `while` loop:

- Checks if the condition is **True**, before execution

- If the condition is **True**, it executes once (one iteration)
- Checks the condition again, if it's **True**, it executes again
- This cycle goes on, it checks for the condition before starting each iteration
- If the condition becomes **False**, then the loop stops

Condition Variable:

- It is important to set the condition variable
- In the example in cell 1, the condition variable is **n**
- The code in line 15, **n = n - 1**, is where we update the condition variable
- Since **n** decreases by 1 at each iteration, eventually it becomes 0 and the loop stops

If you do not set the condition variable properly, then the loop will never stop, which we call as an **Infinite Loop**. You should always check if your condition variable is updated properly in the **while** loop.

Example: Let's find the squares of odd numbers from 1 to 20.

```
[  
2 1 x = 1  
]:  
2  
3 while x <= 20:
```

```
4      # odd?
5      if x % 2 == 1:
6          print("square of {0} is {1}".format(x, x**2))
7
8      # set condition variable
9      x += 1
```

```
[2]:  
      square of 1 is 1  
      square of 3 is 9  
      square of 5 is 25  
      square of 7 is 49  
      square of 9 is 81  
      square of 11 is 121  
      square of 13 is 169  
      square of 15 is 225  
      square of 17 is 289  
      square of 19 is 361
```

In cell 2, we define a **while** loop with the condition of `x <= 20`. So it loops until `x` becomes 20. The last iteration is when `x = 20`. And in each iteration, first we check if `x` is odd or not (line 5). If it's an odd number, then we print it. In line 9, we update the condition variable, `x`, for the **while** loop. We increase it by 1. And the loops goes on.

Example: Let's find the sum of even numbers from 1 to 10.

```
[3]:  
      1      # define the condition variable
      2      k = 1
      3      # define a variable to keep the sum
```

```
4     summation = 0
5
6     while k < 11:
7         # even?
8         if k % 2 == 0:
9             summation += k
10
11        # set condition variable
12        k += 1
13
14     print(summation)
```

```
[3]: 30
```

In cell 3, we check if `k` is less than 11 in the `while` loop. If it is, then we check if it's an even number (line 8). And if it's even then we add it to the `summation` variable as: `summation += k`. In line 12, we update the condition variable by increasing it by 1 as: `k += 1`.

Example: Find the factors of a given number.

```
[4]: 1     # ask for a number
2     number = int(input("Please enter a number: "))
3
4     # condition variable
5     i = 2
6
7     while i < number:
8         # is number divisible by i? -> %
```

```
9     if number % i == 0:
10        print(i)
11
12    # set condition variable
13    i += 1
```

```
[4]: Please enter a number: 12
2
3
4
6
```

In cell 4, we get the number from the user. Then we set the condition variable, `i`, as 2. And the `while` loop checks if `i < number`. If it's `True`, then we check if the number is divisible by `i` as: `number % i == 0`. And if this is `True`, then we print the number `i`. In line 13, we update the condition variable.

Example: Find if a given number is a prime number?

```
[5]: 1 # ask for a number
      2 number = int(input("Please enter a number: "))
      3
      4 # assume the number is prime
      5 is_prime = True
      6
      7 i = 2
      8
      9 while i < number:
      10    # is number divisible by i? -> %
```

```

11  if number % i == 0:
12      is_prime = False
13
14      # set condition variable
15      i = i + 1
16
17      # check if is_prime is True or False
18      if is_prime == True:
19          print("{0} is PRIME".format(number))
20      else:
21          print("{0} is NOT PRIME".format(number))

```

[
5
]:

Please enter a number: 17
17 is PRIME

In cell 5, initially we assume the number is a prime number. Then we loop over every number that is less than this number: **while i < number**. Then we check if the **number** is divisible by **i**. If it is divisible, then this means **number** not prime. So we assign **False** to **is_prime** variable. If the **if** statement in line 11 never becomes **True**, then this means the **number** variable is a prime number and the **is_prime** variable will stay as **True**. Then we have the condition variable update in line 15.

Now that we know how to set a while loop, let's see a case where we do not update the condition variable.

[
6
]:

1 # Infinite loop
2

```
3 i = 1
4
5 while i < 10:
6     print(i)
7     # wrong conditional variable update
8     i -= 1
```

In cell 6, the condition variable is updated in the wrong way. Since we are checking that $i < 10$ and i starts from 1, then we should increase its value at each iteration. But in line 8, we decrease it instead of increasing. Which means it will never become 10, and our while loop will execute forever. Actually if you run the code in cell 6, Python will stop execution after printing some numbers. Otherwise it may consume all of your computer's memory.

for Loop

The second type of loop we have in Python, is the **for** loop. It's simpler than the **while** loop and more commonly used. You do not need to set and update a condition variable externally. The **for** loop manages this automatically for you. Let's see it with examples.

Example: Print numbers from 0 to 10.

```
[7]: 1 # a for loop with a range
      2 for i in range(10):
      3     print(i)
```

```
[  
7  
]:  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9
```

We see a basic structure of the **for** loop in cell 7. It loops over the **range(10)** which means the series of number from 0 to 9. And at every iteration the current number is named as **i**. It is called the **loop variable**. Then we print this variable **i** in line 3.

As you see the basic syntax of a **for** loop is as follows:

```
1 for <loop_variable> in <sequence>:  
2 .....  
3 .....
```

The **range()** function works as: **range(start, end, step)**. Here **start** is the starting number, **end** is the ending number and **step** is the increment at each step. Ending number is not included. The default values for start is 0 and for step size is 1.

```
[  
8 1 range(6, 14, 2)  
]:
```

In cell 8, we define a `range` that starts from 6 (included) and ends at 14 (excluded). The step size is 2, which means it increments by 2.

If we define a variable as `range(10)` this means it starts from 0 (default) increases by 1 (default) and ends at 10 (not included).

Example: Print numbers from 1 to 10, step size 2.

```
[  
9  1 # print items in a range  
]:  
2 for i in range(1, 10, 2):  
3     print(i)
```

```
[  
9  1  
]:  
3  
5  
7  
9
```

In cell 9, we print the items in the `range(1, 10, 2)` with a `for` loop. Which starts from 1, ends at 10 and increases by 1.

Example: Print numbers from 5 up to 30 (included) and step size is 3.

```
[1  
0]: 1 for i in range(5, 31, 3):  
2     print(i)
```

```
[1  
0]: 5  
8
```

```
11
14
17
20
23
26
29
```

In cell 10, we define the range as `range(5, 31, 3)` since we want the number 30 to be included.

Example: Find all positive factors of a given number.

```
[1
1]: 1 # ask for a number
      2 number = int(input("Please enter a number: "))
      3
      4 # factors -> start at 2 -> number
      5
      6 for x in range(2, number):
      7     # is number divisible by x
      8     if number % x == 0:
      9         print(x)
```

```
[1
1]: Please enter a number: 16
      2
      4
      8
```

In cell 11, we define a `for` loop on the `range(2, number)` to find the positive factors of the variable `number`. And at each iteration we check if `number` is divisible by `x`, which is the loop variable. If it is divisible, then we print `x`.

Example: Check if a given number is a prime number.

```
[1] 1 # ask for a number
2: 2 number = int(input("Please enter a number: "))
3:
4: 4 # assume prime
5: 5 is_prime = True
6:
7: 7 for p in range(2, number):
8: 8     # divisibility
9: 9     if number % p == 0:
10: 10         is_prime = False
11:
12: 12 if is_prime:
13: 13     print("{} is PRIME".format(number))
14: 14 else:
15: 15     print("{} is NOT PRIME".format(number))
```

```
[1] 1
2: 2: Please enter a number: 15
3: 15 is NOT PRIME
```

In cell 12, we assume the number is a prime number at the beginning: `is_prime = True`. Then we loop over the `range(2, number)`. At each iteration we check if `number` is divisible by the loop variable `p`. If it is divisible, then this means that `number` is not prime. So we set the `is_prime` variable to `False`.

Example: Print odd numbers from 20 to 1 (excluded), in **descending order**.

```
[1] 1 # range(start, end, step)
3: 3:
```

```
2 # step -> - (minus) -> backwards
3 for i in range(20, 1, -1):
4     if i % 2 == 1:
5         print(i)
```

```
[1
3]:
```

```
19
17
15
13
11
9
7
5
3
```

If we need a `range` in reverse order then we need to set the step size as a **negative** value. Negative step size means **decrease**. And that's what we do in cell 13. We loop over the `range(20, 1, -1)`, which starts from 20 (included) ends at 1 (excluded) and decreases by 1.

Example: Print odd numbers from 20 to 1 (included), in descending order.

```
[1
4]:
```

```
1 # range(start, end, step)
2 # step -> - (minus) -> backwards
3 for i in range(20, 0, -1):
4     if i % 2 == 1:
5         print(i)
```

```
[1
4]:
```

```
19
```

```
17  
15  
13  
11  
9  
7  
5  
3  
1
```

In cell 14, we loop over the `range(20, 0, -1)`, which means start from 20 and end at 0 (excluded) by decreasing 1 at each step.

Loop Over Strings with for

As we already know, a string is a sequence of characters (chars). So looping over a string gives us the characters in the string at each iteration. Let's see it with an example.

```
[1 5]: 1 # define a string  
2 text = 'Python'  
3  
4 # loop over the chars in the string  
5 for character in text:  
6     print(character)
```

```
[1 5]: P  
y  
t  
h  
o  
n
```

In cell 15 we loop over the `text` variable with the help of a `for` loop. At each iteration we get the characters in the `text` and print it.

Example: Concatenate the given text characters with "-". For example; the text Python should become P-y-t-h-o-n.

```
[1  
6]: 1 # ask for an input  
2 user_input = input("Please enter a text: ")  
3  
4 # variable for new input  
5 new_input = ""  
6  
7 # loop  
8 for letter in user_input:  
9     # add - after character and concatenate into  
10    new_input  
11    new_input += letter + "-"  
12  
13 print(new_input)
```



```
[1  
6]: Please enter a text: Python  
P-y-t-h-o-n-
```

In cell 16, we define a string variable as `new_input` to keep the concatenated text. Then we loop over the letters in the `user_input` variable. In line 10, we concatenate the current letter and the “-“ character with the `new_input` value as: `new_input += letter + "-".`

But there is a bug in this code. The last “-” is redundant. Don’t worry, we will fix it later in this chapter.

len

In Python, for sequence types (**str**, **list**, **tuple**, etc.), to get the length of the object we use the built-in **len()** function. It returns the number of items in the sequence.

```
[1] 1 len("Python")
```

```
[1] 6
```

In cell 17, we call the **len()** function with a string as: **len("Python")**. Which returns the number of characters in the string.

```
[1] 1 # define a list
[8]: 2 list_of_numbers_and_letters = [1, 2, 3, 'A', 'B']
      3
      4 # get the length of the list
      5 len(list_of_numbers_and_letters)
```

```
[1] 5
```

In cell 18, we call the **len()** function with a list as **len(list_of_numbers_and_letters)**. It returns the number of items (elements) in the list.

```
[1 9]: 1 # ask for an input
2 user_input = input("Please enter a text to learn its
length: ")
3
4 # get the length of the text
5 length = len(user_input)
6
7 print(length)
```

```
[1 9]: Please enter a text to learn its length: Python
Hands-On
15
```

In cell 19, we ask for the user input. We already know that, the `input()` function always return the user input as a string. That's why the `len()` function in line 5, returns the number of characters in it.

Now let's see the documentation of the `len()` function:

```
[2 0]: 1 len?
```

```
[2 0]: Signature: len(obj, /)
Docstring: Return the number of items in a
container.
Type: builtin_function_or_method
```

enumerate

In Python, when looping over sequences we might need the `index` of current iteration. In these cases we use the built-in `enumerate()` function to get the index.

Example: Remember in cell 16, we defined a **for** loop to concatenate the “-” char into the letters of a string. But there was a bug in it. Now, let’s refactor that code and remove the unnecessary “-” from the end of text.

```
[2
1]: 1 # ask for an input
2 user_input = input("Please enter a text: ")
3
4 # variable for new input
5 new_input = ""
6
7 # loop with enumerate
8 for index, letter in enumerate(user_input):
9     # concatenate the letter
10    new_input += letter
11
12    # if this is the last index
13    if index < len(user_input) - 1:
14        new_input += "-"
15
16 print(new_input)
```

```
[2
1]: Please enter a text: Python
      P-y-t-h-o-n
```

In cell 21, we use the **enumerate()** function to tell the **for** loop that we need the index. The **enumerate()** function returns a **tuple** of two values: **index** and the **item** in the sequence. We assign these values to **index** and **letter** variables in line 8 as: **for index, letter in enumerate(user_input)**. Now we can use the

index in the loop. In line 13, we check if the current `index` value is less than the final index in the user input. The final index for a sequence is always the length of the sequence minus 1. In our case it is `len(user_input) - 1`. And as long as the index is less than the final index we can concatenate a “-” char into it as: `new_input += " - "`. And that’s it. Now we don’t add any “-” chars after the last letter in the string. So the bug is fixed.

Let’s see the documentation for the `enumerate()` function:

```
[2] 1 enumerate?  
[2]:  
      Init signature: enumerate(iterable, start=0)  
      Docstring:    Return an enumerate object.  
      iterable  
      an object supporting iteration  
      The enumerate object yields pairs containing a  
      count (from start, which  
      defaults to zero) and a value yielded by the iterable  
      argument.  
      enumerate is useful for obtaining an indexed list:  
      (0, seq[0]), (1, seq[1]), (2, seq[2]), ...
```

Let’s talk a little bit on the final index, since it’s important when we loop over sequences. Let’s say we want to get the final index for the text of “pandas”:

```
[2] 1 len('pandas') - 1  
[3]:
```

[2
3]:

5

As you see in cell 23, the length of the text “pandas” is 6 but the final index is 5. That’s because indices start from 0 in Python. Let’s loop over this text and print the indexes:

[2
4]:

```
1 # 0, 1, 2, 3, 4, 5
2
3 for index, letter in enumerate('pandas'):
4     print(index)
```

[2
4]:

```
0
1
2
3
4
5
```

Index Example:

In this example we will start from 10 and print up to 100 (included) with step size of 5. As: **10, 15, 20, ..., 100**. In this list of numbers we want to find the number at **index of 3**. Index 3 means, the number in **4th order**. Because the index starts from 0.

First let’s see the number in the list:

[2
5]:

```
1 # first let's see the numbers
2 for number in range(10, 101, 5):
3     print(number)
```

```
[2 5]:  
10  
15  
20  
25  
30  
...  
...  
95  
100
```

Now let's find the number at index 3:

```
[2 6]:  
1 # index 3 -> element in 4th order  
2  
3 number_range = range(10, 101, 5)  
4  
5 for index, number in enumerate(number_range):  
6     if index == 3:  
7         print(number)
```

```
[2 6]:  
25
```

In cell 26, we define a temp variable as `number_range`, to keep the range of numbers. Then in line 5, we set the `for` loop with the `enumerate()` function. And in line 6 we check if the `index` is equal to 3. We print that number at `index = 3`.

We can set the same for loop without any temporary variables. Which means we can use the `range()` function directly in the `enumerate()` function as follows:

```
[2 7]: 1 # do it in one line
2 for i, num in enumerate(range(10, 101, 5)):
3     if i == 3:
4         print(num)
```

```
[2 7]: 25
```

Nested Loops

Most of the time we need to nest the loops in programming. Actually, it's very simple in Python. You need to keep the right level of indentation at your loops, that's all. Here is the syntax for a nested loop:

```
# outer loop
for ....:
    # nested loop
    for ....:
        # nested loop
        for ....:
            # nested loop
            for ....:
```

Let's see some examples on how we create nested loops.

Example: We will draw a square of stars on screen. It will have 3 stars on a row and 3 stars on a column as below:

```
* * *
* * *
* * *
```

First we will do this example with nested `while` loops. Then we will do it with nested `for` loops.

```
[2  
8]: 1 # while  
2  
3 i = 0  
4  
5 # rows  
6 while i < 3:  
7     stars = ""  
8  
9     # columns  
10    j = 0  
11  
12    while j < 3:  
13        stars += "* "  
14        j += 1  
15  
16    # print the row stars  
17    print(stars)  
18  
19    # update the condition variable  
20    i += 1
```

```
[2  
8]: * * *  
* * *  
* * *
```

In cell 28, the outer loop is for the rows. We loop over 3 rows. The row number is called `i` at each iteration. And in each row, we have a nested loop for the columns. At the beginning of each row we set an empty string to keep the stars as: `stars = ""`. Then in

line 12, we start the columns loop. The column number is `j`. For each column iteration we concatenate a star and a space ("* ") to the `stars` variable as: `stars += "* "`. And after we finish iterating over all the columns, we print the `stars` variable in line 17.

Now let's do the same thing with a pair of nested `for` loops. You will see it's simpler with `for` loops.

```
[2
9]: 1      # for
      2
      3      # iterate over rows
      4          # iterate over columns
      5
      6      # rows
      7      for i in range(3):
      8          stars = ""
      9
      10     # columns
      11     for j in range(3):
      12         stars += "* "
      13
      14     print(stars)
```

```
[2
9]:      * * *
          * * *
          * * *
```

In cell 29, the outer loop is for the rows and the inner one is for the columns. We use the `range()` function in both. The idea is the same. At each row, we set an empty string to keep the stars as: `stars = ""`. Then inside the columns loop we concatenate "`*` " to

the `stars` variable (line 12). And finally we print the `stars` variable in line 14.

Exit the Loop: break

In some cases you need to exit the loop before it finishes iteration. In other words, you need to terminate the loop before its completion. In Python, we have the `break` keyword for this purpose. Let's how it works with examples.

Example: Let's print the letters of a string. If we encounter a space character we will exit the loop.

```
[3
0]: 1 # ask for an input
      2 text = input("Please enter a text: ")
      3
      4 print("----- before the loop -----")
      5
      6 for character in text:
      7     # check is space
      8     if character == ' ':
      9         # exit the loop
     10        break
     11
     12        # if not space -> print it
     13        print(character)
     14
     15 print("----- after the loop -----")
```

```
[3
0]: Please enter a text: Python Hands-On is great
----- before the loop -----
```

```
P  
y  
t  
h  
o  
n  
----- after the loop -----
```

In cell 30, we loop over the letters in the string with a **for** loop. And inside the loop, we check if the current character is a space or not as: **if character == ' '**. If it's a space then we exit the loop with the **break** keyword in line 10.

Example: Start from 30 up to 100 and print the numbers. If the number is a multiple of 11 we will terminate the execution.

```
[3  
1]: 1 # multiple of 11  
2     print("----- before the loop -----")  
3  
4     for i in range(30, 101):  
5         # check for multiple of 11  
6         if i % 11 == 0:  
7             print(i)  
8             break  
9         else:  
10            # it is not a multiple of 11  
11            print(i)  
12  
13     print("----- after the loop -----")
```

```
[3  
1]: ----- before the loop -----  
30
```

```
31
32
33
----- after the loop -----
```

In cell 31, inside the **for** loop, we check if the number **i** is a multiple of 11 as: **if i % 11 == 0**. If it is, then we first print it and exit the loop with **break** keyword in line 8. If it's not a multiple of 11, we just print its value in the **else** block.

Skip to the Next Iteration: continue

Sometimes we need only to skip the current iteration and continue with the next one. For these cases we have the **continue** keyword in Python. The **continue** statement will not finalize the whole loop (like **break**), it will only finalize the current iteration. The loop keeps executing with the next one.

Example: We want to print only the even numbers from 1 to 10.

```
[3
2]: 1 for i in range(1, 11):
      2     # if i is odd then skip it
      3     if i % 2 == 1:
      4         continue
      5
      6     # if it comes here, which means it is even
      7     print(i)
```

```
[3
2]: 2
```

```
4
6
8
10
```

In cell 32, we have condition check inside the **for** loop. In line 3 we check if the current number, **i**, is an odd number. If it's odd then we skip the current iteration with the **continue** keyword in line 4. And the loop goes on with the next number. If it's not an odd number we print that number in line 7.

Example: Print the letters in a string. If we encounter the space character we will skip that character. Which means, we will print all the chars but space.

```
[3
3]: 1 # ask for an input
      2 text = input('Please enter a text: ')
      3
      4 for char in text:
      5     # check if space
      6     if char == ' ':
      7         continue
      8
      9     # if not space -> we are here
     10    print(char)
```

```
[3
3]: Please enter a text: Python Hands-On is Great
      P
      y
      t
      h
```

```
o  
n  
H  
a  
n  
d  
s  
-  
O  
n  
i  
s  
G  
r  
e  
a  
t
```

In cell 33 line 6, we check if the char is a space inside the **for** loop. And if it is a space, then we simply skip that char with the **continue** keyword.

for-else

Sometimes you need to know if your loop has completed successfully. Successful loop means a loop with no **break** statement executed in it. For these case we use **for-else** structures. Here is the syntax of a **for-else** block:

```
for .....:  
.....  
.....
```

```

break
.....
.....
else:
..... no break executed .....
..... the break executed in the loop .....

```

In the **for-else** structure, the **else** clause executes after the loop completes normally. This means that the loop did not encounter a **break** statement. If the **break** statement get executed inside the **for** loop then the **else** block will not run. The code will jump out of the whole **for-else** block.

Example: Print numbers from 2 to 10. If the number is 7, exit the loop (**break**). If the loop executes successfully we will print: "Loop executed successfully".

```

[3
4]: 1   print("----- Before For Loop -----")
2
3   for i in range(2, 10):
4     print(i)
5     if i % 7 == 0:
6       break
7   else:
8     print("Loop executed successfully")
9
10  print("----- After For Loop -----")

```

```

[3
4]: ----- Before For Loop -----
2
3

```

```
4
5
6
7
----- After For Loop -----
```

In cell 34, we have a condition check in the **for** loop. In line 5, we check if the number **i** is 7 or not. If it is 7, then we break the loop and the code in line 10 gets executed. Which means the **else** block never gets executed.

Let's see what happens if the number we check is 17 instead of 7:

```
[3
5]: 1  print("----- Before For Loop -----")
2
3  for i in range(2, 10):
4      print(i)
5      if i % 17 == 0:
6          break
7  else:
8      print("Loop executed successfully")
9
10 print("----- After For Loop -----")
```

```
[3
5]: ----- Before For Loop -----
2
3
4
5
6
7
8
```

9

Loop executed successfully
----- After For Loop -----

In cell 35, the condition check in the **for** loop never becomes True. Since there is no number 17 in the **range(2,10)**. And this means that we never encounter the **break** statement in the line 6. In other words, the loop completes successfully. The **else** block gets executed, and it prints the text of “Loop executed successfully” in line 8.

OceanofPDF.com

QUIZ - Loops

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_Loops.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *10_Loops*. Here are the questions for this chapter:

QUIZ - Loops:

Q1:

Define a function that draw a square of stars on the screen. The function name will be **sqaure_of_stars_with_while**. The parameter will be number of stars in one side of square.

Hints:

- ask for number of stars from the user
- use while loop

Expected output for 5 stars:

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

```
[ 1  # Q 1:
]: 2
3 # ----- your solution here -----
4
5
6 # call the function you defined
7 n = int(input("Enter the number of stars: "))
8 sqaure_of_stars_with_while(n)
```

```
[1]: Enter the number of stars: 5
]:  
* * * * *  
* * * * *  
* * * * *  
* * * * *  
* * * * *
```

Q2:

Define a function that draw a square of stars on the screen.
The function name will be **sqaure_of_stars_with_for**.
The parameter will be number of stars in one side of square.
Hints:

- ask for number of stars from the user
- use for loop

Expected output for 5 stars:

```
* * * * *  
* * * * *  
* * * * *  
* * * * *  
* * * * *
```

```
[2]: 1 # Q 2:  
]:  
2  
3 # ----- your solution here -----  
4  
5  
6 # call the function you defined  
7 n = int(input("Enter the number of stars: "))  
8 sqaure_of_stars_with_for(n)
```

```
[2]: Enter the number of stars: 5
```

```
]:  
* * * * *  
* * * * *  
* * * * *  
* * * * *  
* * * * *
```

Q3:

Define a function that draw a right triangle (lower triangle) of stars on the screen.

The function name will be **lower_triangle_with_while**.

The parameter will be number of stars in one side of triangle.

Hints:

- ask for number of stars from the user
- use while loop

Expected output for 5 stars (lower triangle):

```
*  
* *  
* * *  
* * * *  
* * * * *
```

```
[  
3 1 # Q 3:  
]:  
2  
3 # ----- your solution here -----  
4  
5  
6 # call the function you defined  
7 lower_triangle_with_while(5)
```

```
[  
3 :  
]:  
*  
* *
```

```
* * *
* * * *
* * * * *
```

Q4:

Define a function that draw a right triangle (lower triangle) of stars on the screen.

The function name will be **lower_triangle_with_for**.

The parameter will be number of stars in one side of triangle.

Hints:

- ask for number of stars from the user
- use for loop

Expected output for 5 stars (lower triangle):

```
*
```



```
* *
```



```
* * *
```



```
* * * *
```



```
* * * * *
```

```
[  
4 1 # Q 4:  
]:  
2  
3 # ----- your solution here -----  
4  
5  
6 # call the function you defined  
7 lower_triangle_with_for(5)
```

```
[  
4 :  
*  
* *  
* * *  
* * * *  
* * * * *
```

Q5:

Define a function that draw a right triangle (upper triangle) of stars on the screen.

The function name will be **upper_triangle_with_for**.

The parameter will be number of stars in one side of triangle.

Hints:

- ask for number of stars from the user
- use for loop

Expected output for 5 stars (upper triangle):

```
* * * * *
* * * *
* * *
* *
*
```

```
[ 1 # Q 5:
]: 2
3 # ----- your solution here -----
4
5
6 # call the function you defined
7 upper_triangle_with_for(5)
```

```
[ 5
]: * * * * *
* * * *
* * *
* *
*
```

Q6:

Define a function that draw an isosceles triangle of stars on the screen.

The function name will be **isosceles_triangle_with_for**.
The parameter will be number of stars in half of the long side of triangle.

Hints:

- ask for number of stars from the user
- call previous functions for drawing

Expected output for 5 stars (isosceles triangle):

```
*
```

```
* *
```

```
* * *
```

```
* * * *
```

```
* * * * *
```

```
* * * *
```

```
* * *
```

```
*
```

```
[  
6 1 # Q 6:  
]:  
2  
3 # ----- your solution here -----  
4  
5  
6 # call the function you defined  
7 isosceles_triangle_with_for(5)
```

```
[  
6 :  
*  
* *  
* * *  
* * * *  
* * * *
```

```
* * *
* *
*
```

Q7:

Define a bool function which checks if the given number (parameter) is a prime number.

The function name will be **is_prime** and it will return **True** if the number is prime, **False** otherwise.

```
[  
7 1 # Q 7:  
]:  
2  
3 # ----- your solution here -----  
4  
5  
6 # call the function you defined  
7 is_prime(7)
```

```
[  
7 True  
]:
```

Q8:

Define a function named **prime_factors**.

The function will take an integer parameter and will find all the prime factors of that number.

```
[  
8 1 # Q 8:  
]:  
2  
3 # ----- your solution here -----  
4  
5  
6 # call the function you defined  
7 prime_factors(24)
```

```
[  
8 2  
]:  
3
```

Q9:

Define a function named **trees_and_fives**.

It will loop over the integers from 1 to 50, both included.

It will print the numbers but:

- If the number is a multiple of 3 it will print "**Trees**" instead of the number
- If the number is a multiple of 5 it will print "**Fives**" instead of the number
- If the number is a multiple of both 3 and 5 it will print "**Trees&Fives**" instead of the number

```
[  
9 1 # Q 9:  
]:  
2  
3 # ----- your solution here -----  
4  
5  
6 # call the function you defined  
7 trees_and_fives()
```

```
[  
9 1  
]:  
2  
Trees  
4  
Fives  
Trees  
7  
8
```

Trees
Fives
11
Trees
13
14
Trees&Fives
16
17
Trees
...
...
38
Trees
Fives
41
Trees
43
44
Trees&Fives
46
47
Trees
49
Fives

Q10:

Define a function named **concat_sentence**.

The function will take a string as parameter (sentence).

It will first split this sentence into its words.

Then it will split each word into its characters.

And it will add "-" between each character.

Moreover it will add "__" (double underscore) between the words.

And it will return the final sentence.

Hints:

- to split into words -> split()

Sentence = "This is a long story Rango"

Expected Output:

T-h-i-s__i-s__a__l-o-n-g__s-t-o-r-y__R-a-n-g-o

```
[1 0]: 1 # Q 10:  
2  
3 # ----- your solution here -----  
4  
5  
6 # call the function you defined  
7 sentence = "This is a long story Rango"  
8 final_sentence = concat_sentence(sentence)  
9 print(final_sentence)
```

```
[1 0]: T-h-i-s__i-s__a__l-o-n-g__s-t-o-r-y__R-a-n-g-o
```

SOLUTIONS - Loops

Here are the solutions for the quiz for Chapter 10 - Loops.

SOLUTIONS - Loops:

S1:

```
[  
1 1      # S 1:  
]:  
2  
3 def sqaure_of_stars_with_while(n):  
4     """  
5     Function to draw a square of stars.  
6     Parameter: int n, number of stars  
7     Returns: None  
8     """  
9  
10    # rows  
11    i = 0  
12    while i < n:  
13  
14        stars = ""  
15  
16        # columns  
17        j = 0  
18        while j < n:  
19            stars += "* "  
20            j += 1  
21  
22        # print stars for row  
23        print(stars)  
24  
25        i += 1
```

```
26
27
28 # call the function you defined
29 n = int(input("Enter the number of stars: "))
30 sqaure_of_stars_with_while(n)
```

```
[1]:
```

Enter the number of stars: 5

```
      * * * * *
      * * * * *
      * * * * *
      * * * * *
      * * * * *
```

S2:

```
[2]:
```

1 # S 2:

```
2
3 def sqaure_of_stars_with_for(n):
4     """
5         Function to draw a square of stars.
6         Parameter: int n, number of stars
7         Returns: None
8     """
9
10    # rows
11    for i in range(n):
12        stars = ""
13
14        # columns
15        for j in range(n):
16            stars += "* "
```

```
18
19      # print row
20      print(stars)
21
22
23      # call the function you defined
24      n = int(input("Enter the number of stars: "))
25      sqaure_of_stars_with_for(n)
```

```
[2]:
```

```
Enter the number of stars: 5
```

```
* * * * *
* * * * *
* * * * *
* * * * *
* * * * *
```

S3:

```
[3]:
```

```
1      # S 3:
2
3      def lower_triangle_with_while(n):
4
5          # rows
6          i = 0
7          while i < n:
8
9              stars = ""
10
11             # columns
12             j = 0
13             while j <= i:
14                 stars += "* "
```

```
15           j += 1
16
17     print(stars)
18
19     i += 1
20
21
22 # call the function you defined
23 lower_triangle_with_while(5)
```

```
[3]:
```

```
*
* *
* * *
* * * *
* * * * *
```

S4:

```
[4]:
```

```
1     # S 4:
2
3     def lower_triangle_with_for(n):
4
5         # rows
6         for i in range(n):
7
8             stars = ""
9
10            # columns
11            for j in range(i+1):
12                stars += "* "
13
14            print(stars)
```

```
15
16
17 # call the function you defined
18 lower_triangle_with_for(5)

[ 4 ]:
    *
    * *
    * * *
    * * * *
    * * * * *
```

S5:

```
[ 5 ]:
1     # S 5:
]:
2
3     def upper_triangle_with_for(n):
4
5         # rows
6         for i in range(n, 0, -1):
7
8             stars = ""
9
10            # columns
11            for j in range(i):
12                stars += "* "
13
14            print(stars)
15
16
17 # call the function you defined
18 upper_triangle_with_for(5)
```

```
[5]:  
* * * * *  
* * * *  
* * *  
* *  
*
```

S6:

```
[6]:  
1 # S 6:  
2  
3 def isosceles_triangle_with_for(n):  
4  
5     # lower triangle  
6     lower_triangle_with_for(n - 1)  
7  
8     # upper triangle  
9     upper_triangle_with_for(n)  
10  
11  
12     # call the function you defined  
13     isosceles_triangle_with_for(5)
```

```
[6]:  
*  
* *  
* * *  
* * * *  
* * * *  
* * *  
* *
```

```
*
```

S7:

```
[  
7 1 # S 7:  
]:  
2  
3 def is_prime(n):  
4     """  
5     Checks if the number is prime or not.  
6     Parameter: int n  
7     Returns: True if n is prime, False otherwise  
8     """  
9  
10    prime = True  
11  
12    for i in range(2, n):  
13        # if n is divisible by i  
14        if n % i == 0:  
15            prime = False  
16  
17    # return the prime variable  
18    return prime  
19  
20  
21    # call the function you defined  
22    is_prime(7)
```

```
[  
7  
]:
```

True

S8:

```
[  
8 1 # S 8:  
]:
```

```

2
3  def prime_factors(n):
4
5      for i in range(2, n):
6
7          # if n is divisible by i -> i is a factor
8          if n % i == 0:
9
10             # check if i is prime
11             if is_prime(i):
12                 print(i)
13
14
15     # call the function you defined
16     prime_factors(24)

```

```

[8]: 2
      3

```

S9:

```

[9]: 1     # S 9:
       2
       3  def trees_and_fives():
       4
       5      tree = 'Trees'
       6      five = 'Fives'
       7
       8      for i in range(1, 51):
       9
      10         # both 3 and 5
      11         if i % 3 == 0 and i % 5 == 0:

```

```
12     print(tree+'&'+five)
13 elif i % 5 == 0:
14     print(five)
15 elif i % 3 == 0:
16     print(tree)
17 else:
18     print(i)
19
20
21 # call the function you defined
22 trees_and_fives()
```

```
[ 9 ]: 1
2
Trees
4
Fives
Trees
7
8
Trees
Fives
11
Trees
13
14
Trees&Fives
16
17
Trees
...
...
38
```

```
Trees
Fives
41
Trees
43
44
Trees&Fives
46
47
Trees
49
Fives
```

S10:

```
[1
0]: 1 # S 10:
2
3 def concat_sentence(sentence):
4
5     # take the words list
6     words = sentence.split() # split from space
7     chars
8
9     # variable for new sentence
10    concatted_sentence = ""
11
12    # loop #1 -> words
13    for word_index, word in enumerate(words):
14        # print(word)
15
16        concatted_word = ""
17
18        # loop #2 -> chars
19        for char_index, char in enumerate(word):
20            concatted_word += char
```

```

20
21      # add the '-' -> check last
22      if char_index < len(word) - 1:
23          concatted_word += '-'
24
25      # print(concatted_word)
26
27      concatted_sentence += concatted_word
28
29      # check for last index
30      if word_index < len(words) -1:
31          concatted_sentence += '__'
32
33      return concatted_sentence
34
35
36      # call the function you defined
37      sentence = "This is a long story Rango"
38      final_sentence = concat_sentence(sentence)
39      print(final_sentence)

```

[1
0]:

T-h-i-s__i-s__a__l-o-n-g__s-t-o-r-y__R-a-n-g-o

11. Strings

What is a String?

Textual data in Python is handled with `str` objects, or **strings**. Strings are immutable sequences of characters (chars). String literals are written in a variety of ways:

- Single quotes: '**allows embedded "double" quotes**'
- Double quotes: "**allows embedded 'single' quotes**"
- Triple quoted: **'''Three single quotes'''**, **"""Three double quotes"""**

Triple quoted strings may span multiple lines - all associated whitespace will be included in the string literal.

```
[  
1  fruit = 'orange'  
]:  
2  print(fruit)
```

```
[  
1      orange  
]:
```

Defining a string is very easy in Python. You just enclose a text in quotes as we do in cell 1. Since strings are sequences we can access its items via indices.

```
[1]: 1 # the first character in the string
[2]: fruit[0]
```

```
[1]: 'o'
[2]:
```

To get an item we just write its index in the square brackets, `[]`, right after the string name. For example to get the first element in the `fruit` variable we type: `fruit[0]`. Here **the first** means **index = 0**. Remember in Python index starts from 0.

Let's say we want to get the element in the third (3rd) order. Let's see the ordering and the indices side by side:

Order of items	1	2	3	4	5	6
fruit	o	r	a	n	g	e
Indices	0	1	2	3	4	5

Table 11-1: Ordering and Indices

```
[1]: 1 # we want the letter in fruit of order 3
[2]: 2 # order 3 = index 2
[3]: fruit[2]
```

```
[1]: 'a'
[2]:
```

Since we are looking for the item at 3rd order (index 2), we just write as: `fruit[2]`. At it returns the value of `'a'`. Here is another

example in cell 4:

```
[  
4 1 # the letter in 6th order  
]:  
2 # order 6 = index 5  
3 fruit[5]
```

```
[  
4 'e'  
]:
```

Let's say, we want to get the last element in the **fruit** variable. To do this, we first need the length of the string. Let's see it in the next section.

String Length

In Python, to get the length of sequence types we use the standard **len()** function. Let's find the length of the **fruit** variable we defined in cell 1.

```
[  
5 1 # length of fruit variable  
]:  
2 len(fruit)
```

```
[  
5 6  
]:
```

In cell 5, we pass the **fruit** variable to the **len()** function and it returns its length as 6. The length of a string means the total number of characters in it.

Now we are ready to get the last element in the string. Since the value of the **fruit** variable is ‘orange’ and its length is 6, we need **index = 5** for the last element. That’s because index starts from 0. Let’s print the last index and the last element in our variable:

```
[  
6 1 # the last element -> order 6  
]:  
2 # index = 5  
3 # last index = length - 1  
4  
5 last_index = len(fruit) - 1  
6 print('last index:', last_index)  
7  
8 last_element = fruit[last_index]  
9 print("last element:", last_element)
```

```
[  
6 last index: 5  
]:  
last element: e
```

Let’s find the last item in the fruit variable in just one line:

```
[  
7 1 # in one line  
]:  
2 # the last element  
3 fruit[len(fruit) - 1]
```

```
[  
7 'e'  
]:
```

To get the last element we put the last index in square brackets as: `fruit[len(fruit) - 1]`.

Let's see what happens, if you write a wrong index in the square brackets. An index which does not exist in the string:

```
[8]: 1 # wrong last index => fruit[6]
      2 fruit[len(fruit)]
```

```
[8]: IndexError: string index out of range
      ]:
```

As you see in cell 8, we get **IndexError** if we pass an index number which does not exist in the string. Here is another example:

```
[9]: 1 fruit[8]
      ]:
```

```
[9]: IndexError: string index out of range
      ]:
```

```
[10]: 1 x = 'Machine Learning'
      2 y = 'ALAN TURING'
      3 x, y
```

```
[10]: ('Machine Learning', 'ALAN TURING')
```

In cell 10, we define two string variables, `x` and `y`. In JupyterLab you can print objects by separating them via commas.

And that's what we do in line 3 as: `x, y`. Let's print their lengths with commas in the next cell:

```
[1] 1 len(x), len(y)
```

```
[1] 1 (16, 11)
```

String Slicing

A part of a sequence is called a **slice**. For slicing we use **indexing**. The syntax of slicing on a sequence is: `sequence[start : end : step]`. Here the **start** is included but the **end** is excluded. The default values for the start, end and step are as follows:

- **start**: if you leave blank, the **default is 0**
- **end**: if you leave blank, the **default is the last index**
- **step**: if you leave blank, the **default is 1**

```
[1] 1 s = 'Python Hands-On'  
[2]:
```

In cell 12, we define a string variable, with name `s` and value of `'Python Hands-On'`. Now let's see its letters and the corresponding indices for them:

s	P	y	t	h	o	n		H	a	n	d	s	-	O	n
inde	0	1	2	3	4	5	6	7	8	9	1	1	1	1	1
x											0	1	2	3	4

Table 11-2: Indices in the string

Now let's slice the string `s`. Let's start with the basics of slicing and then see the details:

```
[1 3]: 1 # We want to slice 'Python'  
2 s[0:6]
```

```
[1 3]: 'Python'
```

As you see in cell 13, the slice of `s[0,6]` is the word '**Python**'. This is the slice of letters from index 0 to 5. Since the last index is not included, we do not take the space character at index 6.

Actually in cell 13, we do not need to put the 0 as the start index, because the default value for start is 0. So we can write the same slice as follows:

```
[1 4]: 1 # s[0:6]  
2 s[:6]
```

```
[1 4]: 'Python'
```

How to read:

`s[0,6]`: Start slicing from index 0 up to index 6 (step is 1)

Let's do another example to get the '**Hands**' word out of the text. The letter '**H**' is at the index of 7 and the letter '**s**' is at index 11. But the ending index should be 12, if we want to include 11. So here is the slice:

```
[1  
5]: 1 # We want the word Hands  
2 s[7:12]
```

```
[1  
5]: 'Hands'
```

How to Read:

`s[7:12]`: Start slicing from index 7 up to index 12 (step is 1)

The **step** size, also known as **stride**, determines the amount you increase or decrease the index. In other words, step size indicates how big the stride is. Let's see some examples:

```
[1  
6]: 1 # define a new variable  
2 numbers = '123456789'  
3 numbers
```

```
[1  
6]: '123456789'
```

Let's say we want to get only the odd numbers out of the **numbers** string. Here the start should be the first index, which 0. And the end is the last index, which is 1 minus the length of the string. Both start and end indices are at their default values. So we can skip both of them. The only thing we need to type is the step size:

```
[1  
7]: 1 # get the odd numbers  
2 numbers[::-2]
```

```
[1  
7]: '13579'
```

Now, let's get only the even numbers. This time we will start from index 1. The end index will be the default value and the step size is 2 again:

```
[1  
8]: 1 # get even numbers  
2 numbers[1::2]
```

```
[1  
8]: '2468'
```

Slicing can also be used for copying objects. If you slice the whole object, then it will create a copy of it. Let's do it:

```
[1  
9]: 1 # we want to slice all  
2 # copy the whole sequence  
3 numbers[:]
```

```
[1  
9]: '123456789'
```

In cell 19, we slice the string with leaving start, end and step size as blank. Which means all three will have the default values and we will get the whole string. We get a copy of the string actually. You can assign this copy to a new variable as we do in cell 20:

```
[2  
0]: 1 # the complete slice means the copy  
2 copy_of_numbers = numbers[:]  
3 copy_of_numbers
```

```
[2 0]: '123456789'
```

Negative Index

We have two index types for slicing: Positive Index and Negative Index.

Positive Index:

- called the normal index or just ‘**index**’
- goes from **Left to Right**
- starts from **0**

Negative Index:

- called the reverse index or **negative index**
- goes from **Right to Left**
- starts from **-1**

As the first example, we will get the last element in the **numbers** variable with both index types. The length of the numbers variable is 9, which means the last index is 8. But this is the positive index.

```
[2 1]: 1 # get the last element
2 # index (normal)
3 numbers[8]
```

```
[2 1]: '9'
```

Now let's find the last element with the negative index:

```
[2 2]: 1 # get the last element
2 # negative index
3 numbers[-1]
```

```
[2 2]: '9'
```

For the next examples let's define a new string variable:

```
[2 3]: 1 s = 'Python'
2 s
```

```
[2 3]: 'Python'
```

Here is a simple rule:

- if we go from Start to End (Left to Right) index starts from 0
- If we go from End to Start (Right to Left) index starts from -1

s	P	y	t	h	o	n
index ----->	0	1	2	3	4	5
negative index <-----	-6	-5	-4	-3	-2	-1

Table 11-3: Positive and Negative Index for s variable

Let's get the same letters with both index types side by side:

```
[2 4]: 1 # last letter
2 s[5], s[-1]
```

```
[2 4]: ('n', 'n')
```

```
[2 5]: 1 # first letter
2 s[0], s[-6]
```

```
[2 5]: ('P', 'P')
```

```
[2 6]: 1 # 4th index (normal index)
2 # -2 index (negative index)
3 s[4], s[-2]
```

```
[2 6]: ('o', 'o')
```

Practical Way:

The **sum** of absolute values of the normal index and the negative index is equal to the **length** of the sequence. So if you have any of them, either positive or negative, it is very easy to find the other one.

In our case the length of the variable `s` is 6. And as you see in cell 26, the positive index is 4 and the negative index is -2. The sum of absolute values of them is 6, which is the length.

```
[2 7]: 1 # sum of absolute values is the length
2 # these indices points the same letter
3 s[-2], s[4]
```

```
[2 7]: ('o', 'o')
```

Reverse Slicing

Reverse Slicing is slicing the sequence from end to start (from right to left). We can use negative step size for Reverse Slicing.

Reverse Slicing:

- syntax: `sequence[end : start : -step]`
- goes from **end to start**
- the last index is **-1** (not 0)

Let's see how we do reverse slicing. First let's print the **numbers** variable with normal slicing:

```
[2 8]: 1 # print numbers (normal)
2 numbers[:]
```

```
[2 8]: '123456789'
```

Now let's use the reverse slicing to print the **numbers** variable in reverse order:

```
[2 9]: 1 # print numbers from end to start -> reverse order
2 numbers[::-1]
```

```
[2 9]: '987654321'
```

Step Size:

- **positive:** Left to Right (normal)
- **negative:** Right to Left (reverse)

[3
0]:

```
1 # print numbers from start to end incrementing by 2
2 # step is 2
3 numbers[::-2]
```

[3
0]:

```
'13579'
```

[3
1]:

```
1 # print numbers from end to start decrementing by 2
2 # step is -2
3 numbers[::-2]
```

[3
1]:

```
'97531'
```

As you see in cells 30 and 31, increment means positive step, and decrement means negative step.

numbers	1	2	3	4	5	6	7	8	9
index ----->	0	1	2	3	4	5	6	7	8
negative index <--- --	-9	-8	-7	-6	-5	-4	-3	-2	-1

Table 11-4: Positive and Negative Index for numbers variable

Let's get the slice of '456' with the normal index and default step size (1):

[3
2]:

```
1 # normal index ----->
```

```
2 # 456 -> index from 3 up to 6
3 numbers[3:6]
```

```
[3
2]: '456'
```

Now let's get the same slice, '456' with the negative index and positive step size.

```
[3
3]: 1 # negative index <-----
2 # 456 -> index from -6 up to -3
3 numbers[-6:-3:1]
```

```
[3
3]: '456'
```

As a final example, let's get the slice of '654' with negative index and negative step size:

```
[3
4]: 1 # negative index and negative step
2 numbers[-4:-7:-1]
```

```
[3
4]: '654'
```

Strings are Immutable

In Python, Strings are Immutable. You cannot change any part of a string. You can reassign a string but cannot mutate it.

```
[3
5]: 1 say_hello = 'Hello world'
2 say_hello[6]
```

```
[3  
5]: 'w'
```

In cell 35, we define a string variable. The letter at index 6 is ‘w’. Let’s say we want to change it to capital letter ‘W’.

```
[3  
6]: 1 # fix the value in index = 6  
2 say_hello[6] = 'W'
```

```
[3  
6]: TypeError: 'str' object does not support item  
assignment
```

And we get the **TypeError** when we try to mutate the string, in cell 36. **TypeError**: 'str' object does not support item assignment. Item assignment means mutation and it is not supported for strings.

If we want to change a part of a string we have to reassign a new value to it. That’s the way how we can change the letter ‘w’ in the `say_hello` variable. We will use slicing and concatenating for reassignment.

```
[3  
7]: 1 # first slice the part before letter w  
2 slice_1 = say_hello[:6]  
3 slice_1
```

```
[3  
7]: 'Hello '
```

```
[3  
8]: 1 # then slice the part after letter w  
2 slice_2 = say_hello[7:]  
3 slice_2
```

```
[3  
8]: 'orld'
```

Now that we have the part before letter ‘w’ and the part after it, in cells 37 and 38, we can concatenate them with the capital letter ‘W’. And we will reassign this concatenation to the **say_hello** variable.

```
[3  
9]: 1 # finally concatenate the parts and W  
2 say_hello = slice_1 + "W" + slice_2  
3 say_hello
```

```
[3  
9]: 'Hello World'
```

String Methods

Strings have built-in methods in Python. These methods are useful and can make your code much cleaner if you learn how to use them. [Here](#) you can find the complete list of string methods in official Python documentation.^[7]

upper(): Returns a copy of the string with all the cased characters converted to uppercase. The syntax is **my_string.upper()**. Here **my_string** does not change.

```
[4  
0]: 1 word = 'computer'  
2 word
```

```
[4  
0]: 'computer'
```

```
[4  
1]: 1 word_upper_case = word.upper()  
2 word_upper_case
```

```
[4  
1]: 'COMPUTER'
```

In cell 41, we call the `upper()` method on our string variable `word` as: `word.upper()`. And it returns a copy of the `word` in all uppercase. We assign this copy to a new variable called `word_upper_case`. Our variable `word`, stays unchanged.

lower(): Returns a copy of the string with all the cased characters converted to lowercase. The syntax is `my_string.lower()`. Here `my_string` does not change.

```
[4  
2]: 1 word = 'LEARNING'  
2 word
```

```
[4  
2]: 'LEARNING'
```

```
[4  
3]: 1 word_lower_case = word.lower()  
2 word_lower_case
```

```
[4  
3]: 'learning'
```

strip([chars]): Returns a copy of the string with the leading and trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`,

the `chars` argument defaults to removing whitespace. The syntax is `my_string.strip()`. Here `my_string` does not change.

```
[4
4]: 1 sentence = '    This is a sentence.    '
      2 sentence.strip()
```

```
[4
4]: 'This is a sentence.'
```

In cell 44, we defined a `sentence` variable, with the leading and trailing spaces. Then we call the `strip()` method on it. And it removes all the leading and trailing spaces. The `strip()` method does not change the original string. Let's prove it by printing the `sentence` variable again.

```
[4
5]: 1 sentence
```

```
[4
5]: '    This is a sentence.    '
```

We can also specify which set of characters we want to strip. As an example, let's say we want to remove the '@#' characters from a string.

```
[4
6]: 1 text = '#@gmail.com@#'
      2 new_text = text.strip('@#')
      3 new_text
```

```
[4
6]: 'gmail.com'
```

In cell 46, we have a set of '@#' characters in the `text` variable. We remove them via the `strip()` method as: `text.strip('@#')`. It returns a copy of this text without these chars. And we assign it to a new variable called `new_text`.

We have two variants of the `strip()` function: `lstrip()` and `rstrip()`.

`lstrip([chars])`: Works the same as the `strip()` method but only removes the **leading characters**.

`rstrip([chars])`: Works the same as the `strip()` method but only removes the **trailing characters**.

```
[4  
7]: 1 text = '----- dashes before string'  
     2 text.lstrip(' -')
```

```
[4  
7]: 'dashes before string'
```

In cell 47, we have a text with leading ' -' characters. And we remove these chars by `lstrip()` method.

`format()`: Decorates the string with variables.

```
[4  
8]: 1 A = 'Python'  
     2 B = 'Machine'  
     3 C = 'Learning'  
     4  
     5 arg = '{0} - {1} {2}'.format(A, B, C)  
     6 arg
```

[4
8]:

```
'Python - Machine Learning'
```

In cell 48, line 5, we have text which is formatted via the **format()** method as: '{0} - {1} {2}'.**format(A, B, C)**. Here the text is '{0} - {1} {2}'. And the braces {} are placeholders. The arguments in the **format(A, B, C)** method call, are placed into these placeholders respectively. Variable **A** will be in {0}, variable **B** will be in {1} and variable **C** will be in {2}. So the **arg** variable will become '**Python - Machine Learning**'.

[4
9]:

```
1 arg = '{0} - {1} {2}'  
2 arg.format(A, B, C)
```

[4
9]:

```
'Python - Machine Learning'
```

In cell 49, we do the same operation step by step. First let's define the variable **arg** with 3 placeholders in it. Then we replace these placeholders with the variables **A**, **B**, and **C**, with the **format()** method.

find(): Looks for char or substring inside a string. If it finds, it returns the lowest index where it finds. If it doesn't find, it returns **-1**.

[5
0]:

```
1 word = 'window'  
2 word.find('w')
```

[5
0]:

```
0
```

In cell 50, we look for the letter ‘w’ in the variable. The value of the variable is ‘window’ and it has 2 of letter ‘w’ in it. The **find()** method returns the lowest index which is 0.

```
[5  
1]: 1 word.find('in')
```

```
[5  
1]: 1
```

In cell 51, the **find()** method looks for the ‘in’ substring in the variable **word**. The first index where this substring starts is 1, so it returns 1.

```
[5  
2]: 1 word.find('t')
```

```
[5  
2]: -1
```

In cell 52, the **find()** method looks for letter ‘t’ in the variable **word**. The value of the variable is ‘window’. The letter ‘t’ doesn’t exist in this value, so it returns -1.

We can specify the **start** and **end** indices for the **find()** method. The start index is where it starts searching. And the end index means, up to which index it will search. The complete syntax for the **find()** method is: **find(text_to_search, start, end)**.

```
[5  
3]: 1 # specify starting index  
2 word.find('w', 1)
```

[5
3]:

5

In cell 53, the **find()** method starts from index 1. Which means it does not take the first ‘w’ letter at index 0. That’s why it returns 5. Which is the index of the second ‘w’ letter.

[5
4]:

```
1 # start - end -> find('x', start, end)
2 movie_name = 'Batman Dark Knight'
3 movie_name.find('a', 2, 5)
```

[5
4]:

4

In cell 54, the **find()** method starts from index 2 and ends at index 5. It looks for the letter ‘a’ only in this range. And it returns index 4. Which is where the first occurrence of letter ‘a’ exists.

capitalize(): Returns a copy of the sequence with the first letter capitalized and the rest lowercased.

[5
5]:

```
1 movie = 'the godfather'
2 movie.capitalize()
```

[5
5]:

'The godfather'

The first letter of the variable `movie`, in cell 55, becomes capital after we call **movie.capitalize()**.

title(): Returns a copy of the sequence with the first letter of each word capitalized and the rest lowercased.

```
[5] 1 movie = 'the godfather part two'  
[6]: 2 movie.title()
```

```
[5] [5]: 'The Godfather Part Two'  
[6]:
```

As we already know “returns a copy” means the original string stays unmodified. Let’s see the `movie` variable.

```
[5] [5]: 1 movie  
[7]: [7]: 'the godfather part two'
```

Here is another example for the `title()` method:

```
[5] [5]: 1 data = 'pYthon HANDS on'  
[8]: [8]: 2 data.title()  
[5] [5]: 'Python Hands On'
```

isdigit(): Returns `True` if all characters in the string are digits and there is at least one character, `False` otherwise.

```
[5] [5]: 1 txt = 'abc'  
[9]: [9]: 2 txt.isdigit()  
[5] [5]: False  
[9]: [9]:  
[6] [6]: 1 txt = '45'  
[0]: [0]: 2 txt.isdigit()
```

```
[6  
0]: True
```

```
[6  
1]: 1 '4.5'.isdigit()
```

```
[6  
1]: False
```

startswith(): Checks if the string starts with a char or a string. There is also **endswith()** which checks if the string ends with a char or a string.

```
[6  
2]: 1 planet = 'Mars'  
2 planet.startswith('T')
```

```
[6  
2]: False
```

```
[6  
3]: 1 planet.startswith('M')
```

```
[6  
3]: True
```

```
[6  
4]: 1 planet.startswith('Ma')
```

```
[6  
4]: True
```

replace(old, new): Returns a copy of the sequence with all occurrences of subsequence **old** replaced by **new**.

```
[6  
5]: 1 language = 'In 2008 Java is the most popular  
language.'  
2 language
```

[6
5]:

'In 2008 Java is the most popular language.'

Now let's replace the year and the programming language name in the **language** variable. And we will do it by chaining two **replace()** methods:

[6
6]:

```
1 new_language = language.replace('2008',  
2   '2021').replace('Java', 'Python')  
3 new_language
```

[6
6]:

'In 2021 Python is the most popular language.'

in Keyword

To check if a string contains a char or another string we use the **'in'** keyword. The **'in'** keyword returns **True** if the string contains a char or another string, **False** otherwise.

[6
7]:

```
1 # define two variables  
2 program = 'Python Django App'  
3 app = 'App'
```

We have two string variables: **program** and **app**. Now let's see if the value of the **app** variable exists in the value of the **program** variable:

[6
8]:

```
1 # check if app is in program  
2 app in program
```

[6
8]:

True

```
[6 9]: 1 'Dj' in program
```

```
[6 9]: True
```

```
[7 0]: 1 'Dj' in app
```

```
[7 0]: False
```

```
[7 1]: 1 '2' in 'abc1234'
```

```
[7 1]: True
```

Example: Ask for an input from the user. Check if the input includes the letter 'a' nor not. If it's **True** print as: 'This input has a in it'.

```
[7 2]: 1 def is_a_in_it():
2     # ask for input
3     text = input('Please enter a text: ')
4
5     # check if text has letter a in it
6     if 'a' in text:
7         print('This input has a in: {}'.format(text))
8     else:
9         print('This input has NO a in:
{}'.format(text))
```

In cell 72, we define the `is_a_in_it()` function. In line 6, it checks if the user input (`text`) has the letter 'a' in it. The syntax is: `if 'a' in text`. Let's call the function and test it:

```
[7  
3]: 1 is_a_in_it()
```

```
[7  
3]: Please enter a text: Python is the most popular  
language  
This input has a in: Python is the most popular  
language
```

Now let's convert the `is_a_in_it()` function into a `bool` function. It will return `True` if the text includes the letter '`a`' and `False` otherwise. And we will use the `in` keyword in the `return` statement. That way, we will have a one line `return` statement, without using any `if-else` block.

```
[7  
4]: 1 # one line return  
2 def is_a_in_it():  
3     # ask for input  
4     text = input('Please enter a text: ')  
5  
6     # return if text has letter a in it  
7     return 'a' in text
```

In cell 74, we return the expression of '`a`' `in` `text`. Which is a `bool` expression. Now let's call the function:

```
[7  
5]: 1 is_a_in_it()
```

```
[7  
5]: Please enter a text: Deep Learning  
True
```

Since the text "Deep Learning" has the letter '`a`' in it, our function returns `True`.

Now comes the best part. Actually we can write the `is_a_in_it()` function in only one line. Let's do it:

```
[7] 1 # one line function
6]: 2 def is_a_in_it():
3     # ask the input and return if it has a in it
4     return 'a' in input('Please enter a text: ')
```

In cell 76, we ask for the user input and check if it has the letter 'a' in it, in just one line. This is what we call the functional programming approach. No temporary variables and code is self-explanatory.

```
[7] 1 is_a_in_it()
7]: 2 Please enter a text: Machine Learning
    3 True
```

String Comparison

Sometimes we need to check two strings to see if they are equal or which one is greater (alphabetically comes later). We call this as String Comparison and it's important for you to understand how Python compares the strings.

```
[7] 1 # define a variable
8]: 2 fruit = 'apple'
3
4 # check its value
```

```
5 if fruit == 'apple':  
6     print('yes it is apple')  
7 else:  
8     print('no it is not apple')
```

[7]
8]: yes it is apple

In cell 78, line 5, in the `if` statement, we check if the value of the `fruit` variable is “apple” as: `fruit == 'apple'`. And it returns `True`. Now let’s compare two string variables:

```
[7]  
9]: 1 # define two variables  
2 orange = 'orange'  
3 apple = 'apple'  
4  
5 # compare them  
6 if orange < apple:  
7     print('orange is less than apple.')  
8 else:  
9     print('orange is greater than apple.')
```

[7]
9]: orange is greater than apple.

In cell 79, line 6, in the `if` statement, we check as: `orange < apple`. This check is actually an **alphabetical** order check. And since the letter ‘o’ comes after the letter ‘a’ this check returns **False**.

```
[8]  
0]: 1 # is letter 'o' is less than 'a'  
2 'o' < 'a'
```

```
[8  
0]: False
```

Python uses the [ASCII](#) codes of characters to compare the strings. ASCII stands for **American Standard Code for Information Interchange** and is a character encoding standard for electronic communication. Each character has an ASCII code. For example the ASCII code for letter ‘a’ is **97**. And the ASCII code for letter ‘o’ is **111**. That’s why the letter ‘o’ is greater than the letter ‘a’.

We have built-in `ord()` function to get the ASCII codes of characters in Python. Let’s see some examples:

```
[8  
1]: 1 # ASCII code -> ord()  
2 ord('a')
```

```
[8  
1]: 97
```

```
[8  
2]: 1 ord('o')
```

```
[8  
2]: 111
```

The ASCII code of the capital letters are less than their lowercase forms. For example the ASCII code for capital letter ‘A’ is **65**.

```
[8  
3]: 1 ord('A')
```

```
[8  
3]: 65
```

```
[8  
4]: 1 # see if 'A' is less than 'a'  
2 ord('A') < ord('a')
```

```
[8  
4]: True
```

```
[8  
5]: 1 'A' < 'a'
```

```
[8  
5]: True
```

As you see in cells 84 and 85, In Python '**A**' is less than '**a**'.

OceanofPDF.com

QUIZ - Strings

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_Strings.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *11_Strings*. Here are the questions for this chapter:

QUIZ – Strings:

Q1:

Define a function named **only_first_letters**.

It will take a string as the parameter.

And it will convert all the first letters of each word into upper case.

All the remaining letter will be lower case.

```
[  
1  1 # Q 1:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 proper_text = only_first_letters('pyThon iS grEAt')  
8 print(proper_text)
```

```
[  
1  Python Is Great  
]:
```

Q2:

Define a function taking a text and a letter as parameters.

The function name will be **count_the_letter**.

Function will return the number (count) of occurrences of the letter in the text.

Hints:

- use built-in function
- your function body should be one line of code
- <https://docs.python.org/3/library/stdtypes.html#string-methods>

```
[1] 1 # Q 2:  
[2] 2  
[3] 3 # ---- your solution here ----  
[4] 4  
[5] 5  
[6] 6 # call the function you defined  
[7] 7 text = input('Please enter a text: ')  
[8] 8 letter = input('Now a letter: ')  
[9] 9 count_the_letter(text, letter)
```

```
[1] 1 Please enter a text: Python Hands-On is a perfect  
[2] 2 book :)  
[3] 3 Now a letter: e  
[4] 4 2
```

Q3:

Define a function named **count_letters**.

It will ask for sentence from the user.

And it will count the occurrences of each letter and print it.

Hints:

- Do not count space character
- Use the function you defined in Q2.

Expected Output:

Please enter a sentence: pandas

letter p: 1 time(s)

letter a: 2 time(s)

```
letter n: 1 time(s)
letter d: 1 time(s)
letter s: 1 time(s)
```

```
[ 3 1 # Q 3:
]: 2
3 # ---- your solution here ----
4
5
6 # call the function you defined
7 count_letters()
```

```
[ 3 Please enter a sentence: pandas
]: 1
letter p: 1 time(s)
letter a: 2 time(s)
letter n: 1 time(s)
letter d: 1 time(s)
letter s: 1 time(s)
```

Q4:

Your text is: **'Monday Tuesday Wednesday Thursday Friday'**.

Do slicing operations on this text and answer the following questions:

- find the first 3 chars
- find the chars between index 7 and 10 (inc.)
- find the chars between index 2 and 12 (exc.)
- find the chars at indices 3, 4 and 5

```
[ 4 1 # Q 4:
]: 2
```

```
3 text = 'Monday Tuesday Wednesday Thursday  
Friday'  
4  
5  
6 # find the first 3 chars  
7 # ---- your solution here ----  
8  
9 # find the chars between index 7 and 10 (inc.)  
10 # ---- your solution here ----  
11  
12 # find the chars between index 2 and 12 (exc.)  
13 # ---- your solution here ----  
14  
15 # find the chars at indices 3, 4 and 5  
16 # ---- your solution here ----
```

[
4
]:

the first 3 chars: Mon

the chars between index 7 and 10 (inc.): Tues
the chars between index 2 and 12 (exc.): nday
Tuesd

the chars at indices 3, 4 and 5: day

Q5:

Your text is: '**Monday Tuesday Wednesday**'

Do slicing operations on this text and answer the following questions:

- find the chars from index 1 to 7 with step size 2
- find the chars from index 3 to 20 with step size 3
- find the chars with even indices (0, 2, 4, 6 ...)
- find the chars with odd indices (1, 3, 5, 7 ...)

[
5
]:

1 # Q 5:

```
2
3     text = 'Monday Tuesday Wednesday'
4
5
6     # find the chars from index 1 to 7 with step size 2
7     # ---- your solution here ----
8
9     # find the chars from index 3 to 20 with step size
10    3
11    # ---- your solution here ----
12
13    # find the chars with even indices (0, 2, 4, 6 ...)
14    # ---- your solution here ----
15
16    # find the chars with odd indices (1, 3, 5, 7 ...)
17    # ---- your solution here ----
```

[
5
]:

the chars from index 1 to 7 with step size 2: ody

the chars from index 3 to 20 with step size 3: d
eaWn

the chars with even indices (0, 2, 4, 6 ...): Mna
usa ensa

the chars with odd indices (1, 3, 5, 7 ...):
odyTedyWdedy

Q6:

Your text: **'Monday Tuesday Wednesday'**

Do slicing operations on this text and answer the following questions:

- find the last character
- find the 3rd character from last
- find last 3 characters
- print the text in reverse order

```
[6]: 1 # Q 6:  
[6]: 2  
[6]: 3 text = 'Monday Tuesday Wednesday'  
[6]: 4  
[6]: 5  
[6]: 6 # find the last character  
[6]: 7 # ---- your solution here ----  
[6]: 8  
[6]: 9 # find the 3rd character from last  
[6]: 10 # ---- your solution here ----  
[6]: 11  
[6]: 12 # find last 3 characters  
[6]: 13 # ---- your solution here ----  
[6]: 14  
[6]: 15 # print the text in reverse order  
[6]: 16 # ---- your solution here ----
```

```
[6]: the last character: y  
[6]: the 3rd character from last: d  
[6]: last 3 characters: day  
[6]: the text in reverse order: yadsendeW yadseuT  
[6]: yadnoM
```

Q7:

Your text: **'Monday Tuesday Wednesday'**

Do slicing operations on this text and answer the following questions:

- find the chars between 2nd and 6th from last
- find the chars between 2nd (inc.) and 6th from last
- find the negative index which correspond to the char with index 10

- find all characters except the last one
- find all characters except the first and the last one

```
[7]: 1 # Q 7:
      2
      3 text = 'Monday Tuesday Wednesday'
      4
      5
      6 # find the chars between 2nd and 6th from last
      7 # ---- your solution here ----
      8
      9 # find the chars between 2nd (inc.) and 6th from
      10 # ---- your solution here ----
      11
      12 # find the negative index which correspond to the
      13 # char with index 10
      14 # abs(index) + abs(negative_index) = length
      15 # ---- your solution here ----
      16
      17 # find all characters except the last one
      18 # ---- your solution here ----
      19
      20 # find all characters except the first and the last
      # ---- your solution here ----
```

```
[7]: the chars between 2nd and 6th from last: nesd
      the chars between 2nd (inc.) and 6th from last:
      nesda
      negative_index: -14
```

all characters except the last one: Monday
Tuesday Wednesda

all characters except the fist and the last one:
onday Tuesday Wednesda

Q8:

Define a function named **split_email**.

It will ask for an email from the user.

And it will split the email into user name and domain name using index.

Hints:

- if the input doesn't include '@' char, return a text as 'Invalid email format.'

Expected output:

Please enter an email: abc@acme.corp

User Name: abc

Domain: acme.corp

```
[  
8 1 # Q 8:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 split_email()
```

```
[  
8 Please enter an email: abc@acme.com  
]:
```

User Name: abc

Domain: acme.com

Q9:

Define a function named **reverse**.

It will ask for a sentence from the user and return the reverse of this sentence.

Expected Output:

Please enter a sentence: yesterday came suddenly
'ylneddus emac yadretsey'

```
[  
9  1 # Q 9:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 reverse()
```

```
[  
9  Please enter a sentence: yesterday came suddenly  
]:  
'ylneddus emac yadretsey'
```

Q10:

Define a function named **replace_with_next_letter**.

It will ask for an input from the user.

And will replace every character with the next letter in the English alphabet.

Hints:

- use nested functions
- there should be `is_lower()` and `is_upper()` functions to check whether the char is small or upper
- there should be `next_letter()` function to take a letter and return the next one from alphabet
- alphabets:

```
alphabet_lower = 'abcdefghijklmnopqrstuvwxyz'  
alphabet_upper = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
```

- use index to get the related letter

Expected Output:

Please enter a text: Python

Rzuipo

Please enter a text: abcZ

bcdA

```
[1
0]: 1      # Q 10:
2
3      # ---- your solution here ----
4
5
6      # call the function you defined
7      new_text = replace_with_next_letter()
8      print(new_text)
9
10     new_text = replace_with_next_letter()
11     print(new_text)
```

```
[1
0]: Please enter a sentence: Python
```

Rzuipo

Please enter a sentence: abcZ

bcdA

SOLUTIONS - Strings

Here are the solutions for the quiz for Chapter 11 - Strings.

SOLUTIONS - Strings:

S1:

```
[  
1 1      # S 1:  
]:  
2  
3      # Long Way  
4      # def only_first_letters(text):  
5  
6      #     # make all letters -> lower  
7      #     lower_text = text.lower()  
8  
9      #     # then, only make first letters as capital  
10     #     capital_text = lower_text.title()  
11  
12     #     return capital_text  
13  
14      # Short Way  
15      def only_first_letters(text):  
16          # chain  
17          return text.lower().title()  
18  
19  
20      # call the function you defined  
21      proper_text = only_first_letters('pyThon iS  
22      grEA't')  
23      print(proper_text)
```

```
[  
1      Python Is Great
```

```
]:
```

S2:

```
[  
2 1 # S 2:  
]:  
2  
3 def count_the_letter(text, letter):  
4     return text.count(letter)  
5  
6  
7     # call the function you defined  
8     text = input('Please enter a text: ')  
9     letter = input('Now a letter: ')  
10    count_the_letter(text, letter)
```

```
[  
2  
]:
```

Please enter a text: Python Hands-On is a perfect
book :)
Now a letter: e
2

S3:

```
[  
3 1 # S 3:  
]:  
2  
3 def count_letters():  
4  
5     sentence = input('Please enter a sentence: ')  
6  
7     letters_printed = ""  
8  
9     for letter in sentence:  
10  
11         # if it's a space -> continue
```

```
12     if letter == ' ':
13         continue
14
15     # this letter is not a space
16     count = count_the_letter(sentence, letter)
17
18     # print
19     # check if already printed
20     if not letter in letters_printed:
21         print("letter {0}: {1}")
22         time(s)".format(letter, count))
23         letters_printed += letter
24
25     # call the function you defined
26     count_letters()
```

[
3
]:

```
Please enter a sentence: pandas
letter p: 1 time(s)
letter a: 2 time(s)
letter n: 1 time(s)
letter d: 1 time(s)
letter s: 1 time(s)
```

S4:

[
4
]:

```
1     # S 4:
2
3     text = 'Monday Tuesday Wednesday Thursday
4     Friday'
5
```

```
6  # find the first 3 chars
7  # text[0:3] or text[:3]
8  print('the first 3 chars: ', text[:3])
9
10 # find the chars between index 7 and 10 (inc.)
11 # text[7:11]
12 print('the chars between index 7 and 10 (inc.): ',
13   text[7:11])
14
15 # find the chars between index 2 and 12 (exc.)
16 # text[2:12]
17 print('the chars between index 2 and 12 (exc.): ',
18   text[2:12])
19
20 # find the chars at indices 3, 4 and 5
21 # text[3:6]
22 print('the chars at indices 3, 4 and 5: ', text[3:6])
```

[
4
]:

the first 3 chars: Mon

the chars between index 7 and 10 (inc.): Tues
the chars between index 2 and 12 (exc.): nday
Tuesd
the chars at indices 3, 4 and 5: day

S5:

[
5
]:

```
1  # S 5:
2
3  text = 'Monday Tuesday Wednesday'
4
5
6  # find the chars from index 1 to 7 with step size 2
```

```

7  # text[1:7:2]
8  print('the chars from index 1 to 7 with step size
9   2: ', text[1:7:2])
10
10 # find the chars from index 3 to 20 with step size
11 3
11 # text[3:20:3]
12 print('the chars from index 3 to 20 with step size
13 3: ', text[3:20:3])
14
14 # find the chars with even indices (0, 2, 4, 6 ...)
15 # text[::-2]
16 print('the chars with even indices (0, 2, 4, 6 ...): '
17   , text[::-2])
18
18 # find the chars with odd indices (1, 3, 5, 7 ...)
19 # text[1::2]
20 print('the chars with odd indices (1, 3, 5, 7 ...): ',
21   text[1::2])

```

[
5
]:

the chars from index 1 to 7 with step size 2: ody
the chars from index 3 to 20 with step size 3: d
eaWn
the chars with even indices (0, 2, 4, 6 ...): Mna
usa ensa
the chars with odd indices (1, 3, 5, 7 ...):
odyTedyWdedy

S6:

[
6
]:

```

1  # S 6:
2
3  text = 'Monday Tuesday Wednesday'

```

```
4
5
6  # find the last character
7  # text[-1]
8  print('the last character: ', text[-1])
9
10 # find the 3rd character from last
11 # text[-3]
12 print('the 3rd character from last: ', text[-3])
13
14 # find last 3 characters
15 # -3, -2, -1
16 # text[-3:]
17 print('last 3 characters: ', text[-3:])
18
19 # print the text in reverse order
20 # text[::-1]
21 print('the text in reverse order: ', text[::-1])
```

```
[  
6  
]:
```

```
the last character: y
the 3rd character from last: d
last 3 characters: day
the text in reverse order: yadsendew yadseut
yadnom
```

S7:

```
[  
7  1  # S 7:  
]:  
2  
3  text = 'Monday Tuesday Wednesday'  
4  
5
```

```
6  # find the chars between 2nd and 6th from last
7  # "-6, -5, -4, -3, " -2, -1
8  # text[-6:-2]
9  print('the chars between 2nd and 6th from last: ',
10 text[-6:-2])
11
12 # find the chars between 2nd (inc.) and 6th from
13 last
14 # "-6, -5, -4, -3, -2, " -1
15 # text[-6:-1]
16 print('the chars between 2nd (inc.) and 6th from
17 last: ', text[-6:-1])
18
19 # find the negative index which correspond to the
20 char with index 10
21 # abs(index) + abs(negative_index) = length
22 length = len(text)
23 negative_index = length - 10
24 negative_index = -negative_index
25 print('negative_index: ', negative_index)
26
27 # find all characters except the last one
28 # text[:-1]
29 print('all characters except the last one: ',
text[:-1])
30
31 # find all characters except the fist and the last
32 one
33 # text[1:-1]
34 print('all characters except the fist and the last
35 one: ', text[1:-1])
```

[
7
]:

the chars between 2nd and 6th from last: nesd

```
the chars between 2nd (inc.) and 6th from last:  
nesda  
negative_index: -14  
all characters except the last one: Monday  
Tuesday Wednesda  
all characters except the fist and the last one:  
onday Tuesday Wednesda
```

S8:

```
[  
8 1 # S 8:  
]:  
2  
3 def split_email():  
4     email = input("Please enter an email: ")  
5  
6     # check @ and find index  
7     index = email.find('@')  
8  
9     # find() -> if finds index,  
10    # not find -1  
11    if index == -1:  
12        print('Invalid email format.')  
13    else:  
14        print("User Name:", email[:index])  
15        print("Domain:", email[index+1:])  
16  
17  
18    # call the function you defined  
19    split_email()
```

```
[  
8  
]:  
Please enter an email: abc@acme.com  
User Name: abc
```

Domain: acme.com

S9:

```
[  
9  1  # S 9:  
]:  
2  
3  def reverse():  
4      sentence = input('Please enter a sentence: ')  
5      # return the sentence in reverse order  
6      return sentence[::-1]  
7  
8  
9  # call the function you defined  
10 reverse()
```

```
[  
9  Please enter a sentence: yesterday came suddenly  
]:  
'ylneddus emac yadretsey'
```

S10:

```
[1  
0]: 1  # S 10:  
2  
3  def replace_with_next_letter():  
4  
5      sentence = input('Please enter a sentence: ')  
6  
7      alphabet_lower = 'abcdefghijklmnopqrstuvwxyz'  
8      alphabet_upper =  
'ABCDEFGHIJKLMNOPQRSTUVWXYZ'  
9  
10     new_sentence = ""  
11  
12     def is_lower(letter):
```

```
13     return letter in alphabet_lower
14
15     def is_upper(letter):
16         return letter in alphabet_upper
17
18     def next_letter(alphabet, letter):
19         index = alphabet.find(letter)
20         if index == len(alphabet) - 1:
21             index = -1
22         return alphabet[index + 1]
23
24     for letter in sentence:
25
26         # lower or upper
27         if is_lower(letter):
28             new_letter = next_letter(alphabet_lower,
letter)
29         elif is_upper(letter):
30             new_letter = next_letter(alphabet_upper,
letter)
31         else:
32             new_letter = letter
33
34         # concat the new_letter into new_sentence
35         new_sentence += new_letter
36
37     return new_sentence
38
39
40     # call the function you defined
41     new_text = replace_with_next_letter()
42     print(new_text)
43
44     new_text = replace_with_next_letter()
```

```
45     print(new_text)
```

[1
0]:

Please enter a sentence: Python

Rzuipo

Please enter a sentence: abcZ

bcdA

OceanofPDF.com

12. Project 2 - Words

This is our second project in this book. It will cover the topics what we have seen so far. Mainly it's going to be on functions, strings and string operations.

In this project we will search for words based on predefined criteria. For instance, which words does not include any vowels or the letters of which words are in alphabetical order.

The words are from [Lorem Ipsum](#). **Lorem Ipsum** is simply dummy text of the printing and typesetting industry.

You can find the source code of this project and the *words.txt* file in the [Github repository](#) of the book. It is in the folder named *12_Project_2_Words*.

Preparation

We will be reading a text file with Python. To be able to read a file we use the built-in `open()` function. The syntax is as follows: **`open(file, mode)`**.

Here the **file** is a physical file that is on the computer. We will open the *words.txt* file in this project. You must have *words.txt* file in the same folder where you run the code for the notebook file (.ipynb file) of this project. They need to be in the same folder for this project code to work properly.

The second parameter of the `open()` function is the `mode`. It is the reading mode. It tells in which mode you intend to open the file.

The mode options are as follows:

- `r`: read (default)
- `a`: append
- `w`: write
- `x`: create

Now let's start reading our `words.txt` file:

```
[  
1  1 # open()  
]:  
2  file = open('words.txt')
```

When we run cell 1, the `open()` function will open the file named '`words.txt`' and it will return a `file` object. This `file` object contains the actual file contents.

Now that we have the `file` object with all the content in it, let's read it. To start with, we will read it line by line. A single line of content at each reading operation. This is done via the `readline()` function.

```
[  
2  1 # let's read the first line  
]:  
2  file.readline()
```

```
[2]: 'Lorem\n'  
]:
```

In cell 2, we get the first line of content in the file object, which is '**L**orem\n'. Here is '\n' character is the new line character. Now let's read the second line in the file:

```
[3]: 1 # let's read the second line  
]: 2 file.readline()
```

```
[3]: 'ipsum\n'  
]:
```

As you see in cell 3, we get the second line in the file. And again it has the new line character after it as: '**i**psum\n'.

Every time we run the `file.readline()` function, Python reads the current line and move to the next line waiting for the next `readline()` function call. Like a cursor reading line by line.

Now let's see the most common special characters which we encounter when reading file content.

Special Characters:

- **space**
- \n : new line character
- \r : carriage return (enter)

Now let's run the `readline()` function for the third time and assign the returning value to a variable:

```
[  
4  1 # run readline for 3rd time  
]:  
2 # assign the result to a variable  
3 line = file.readline()  
4 line  
  
[  
4  'dolor\n'  
]:
```

Instead of reading the file line by line, let's print the first 6 lines with a loop. And since we need the index value, we will use the `enumerate()` function.

```
[  
5  1 # print the first 6 lines  
]:  
2 file = open('words.txt')  
3  
4 # we need index -> enumerate  
5 for index, line in enumerate(file):  
6     if index <= 5:  
7         print(line)  
  
[  
5  Lorem  
]:  
ipsum  
dolor  
sit
```

```
amet  
consectetur
```

In cell 5, line 2, we read the file again as: `file = open('words.txt')`. Whenever we read the file the cursor resets to the starting position. So we start to read the file from the first line.

We see extra blank lines in the output of cell 5. That's because every line we read ends with a new line characters like: `'Lorem\n'`. And this new line character, prints an empty blank line. Let's remove it now:

```
[  
6 1 # remove new lines  
]:  
2 file = open('words.txt')  
3  
4 for index, line in enumerate(file):  
5     if index <= 5:  
6         # split() -> splits the text from space chars  
6         (space, \n, \r)  
7         word_list = line.split()  
8         print(word_list)
```

```
[  
6 ['Lorem']  
]:  
['ipsum']  
['dolor']  
['sit']  
['amet']  
['consectetur']
```

In cell 6, line 7, we split the line as: `line.split()`. The `split()` function splits the text from any of the space characters which are **space**, new line (`\n`) and carriage return (`\r`). And it returns a list of strings. We call this list as `word_list` in line 7. Then we print this list. That's why in the output, you see each line as a separate list object.

Now that we learned how to open a file, how to read lines and how to split the lines; we are ready to start our exercises.

Exercises

Exercise 1:

We will print the words which have number of letters more than 10.

```
[  
7 1      # read the file  
]:  
2     file = open('words.txt')  
3  
4     for line in file:  
5         # split() -> \n  
6         word_list = line.split()  
7  
8         # word is the first element in list  
9         word = word_list[0]  
10  
11        # check number of letters  
12        if len(word) > 10:  
13            print(word)
```

```
[7]:  
consectetur  
pellentesque  
sollicitudin  
scelerisque  
ullamcorper  
condimentum  
Suspendisse
```

In cell 7, in the `for` loop we get each line in the file one by one: `for line in file`. Then we split the line and assign it to a variable as: `word_list = line.split()`. Since the `split()` function returns a list, we need the first element of this list. That's the element at index 0 and we get it as: `word = word_list[0]`. Then in line 12, we check if the length of the `word` is greater than 10. We print the `word` if it's `True`.

Exercise 2:

Define a new function. This function will search for the words which have no vowels in it. The vowels are `a`, `e`, `i`, `o` and `u`.

```
[8]:  
1  def check_for_vowels():  
2      # read the file  
3      file = open('words.txt')  
4  
5      for line in file:  
6          # split() -> \n  
7          word_list = line.split()  
8
```

```
9      # word is the first element in list
10     word = word_list[0]
11
12     # check if word includes any vowel
13     if not 'a' in word and \
14         not 'e' in word and \
15         not 'i' in word and \
16         not 'o' in word and \
17         not 'u' in word:
18         print(word)
```

In cell 8, the operations we do in the `check_for_vowels()` function is very similar to what we did in cell 7. The difference is in the `if` statement in line13. Since we do not want the `word` to include any of the vowels we check for each vowel one by one. And we combine these conditions with ‘`and`’ keyword. Be careful that we have the ‘`not`’ keyword at each condition.

The **backslash (\)** character we have here is for the next line. Normally every line is a separate block of code in Python. But when we want a block of code to continue on the next line we use the ‘\’ character for this. That way, we are able to combine multiple lines in one expression.

Now let’s call the `check_for_vowels()` function and see the words which have no vowels in them:

```
[ 9  1 check_for_vowels()
]:
```

```
[ 9  In
```

]:

trylyk
Antryl

As you see in the output of cell 9, our `check_for_vowels()` function has a bug in it. It returns the words with capital letter ‘I’, which is also a vowel. That’s because we didn’t consider a case for letters being uppercases. Let’s redefine our function and fix this bug.

```
[1]: 1  def check_for_vowels():
0]: 2
3      # read the file
4      file = open('words.txt')
5
6      for line in file:
7          # split() -> \n
8          word_list = line.split()
9
10     # word is the first element in list
11     word = word_list[0]
12
13     # convert into lower case
14     word = word.lower()
15
16     # check if included vowel
17     if not 'a' in word and \
18         not 'e' in word and \
19         not 'i' in word and \
20         not 'o' in word and \
21         not 'u' in word:
22         print(word)
```

We do code refactoring in cell 10. And we add line 14, where we convert the `word` into lowercase as: `word = word.lower()`. Now if we call it, it will handle the uppercase vowels as well.

```
[1] 1 check_for_vowels()
```

```
[1] 1 trylyk
```

Exercise 3:

Get the words which includes the letters '`ae`' together and in this order.

```
[1] 1 def check_for_ae():
[2] 2     # read the file
[3] 3     file = open('words.txt')
[4]
[5] 5     for line in file:
[6] 6         # split() -> \n
[7] 7         word_list = line.split()
[8]
[9] 9         # word is the first element in list
[10]10     word = word_list[0]
[11]
[12]12     # convert into lower case
[13]13     word = word.lower()
[14]
[15]15     # check if included vowel
[16]16     if 'ae' in word:
[17]17         print(word)
```

In cell 12, line 16, inside the `check_for_ae` function, we check if ‘ae’ is in the `word` as follows: `if 'ae' in word`. If it’s true then we print the `word`. Rest is the same as `check_for_vowels()` function. Let’s call it and see the words.

```
[1] 1 check_for_ae()  
[3]:
```

```
[1] 1  
[3]:  
    vitae  
    praesent  
    aenean  
    maecenas
```

Exercise 4:

We will define a bool function called `is_any_forbidden_letter`. It will take letters which are forbidden and a text as its parameters. It will check if this text includes any of these forbidden letters. And return either `True` or `False`.

```
[1] 1 def is_any_forbidden_letter(text, forbidden_letters):  
[4]: 2     # loop over every letter in the text  
3     for letter in text:  
4         if letter in forbidden_letters:  
5             return True  
6     else:  
7         return False
```

Now let’s call the function with 2 different sets of sentence and forbidden letters:

```
[1 5]: 1 sentence = 'This is a forbidden sentence.'  
2 forbidden_letters = 'ae'  
3 # call the function with these arguments  
4 is_any_forbidden_letter(sentence, forbidden_letters)
```

```
[1 5]: True
```

```
[1 6]: 1 sentence = 'This is the second forbidden sentence.'  
2 forbidden_letters = 'xy'  
3  
4 is_any_forbidden_letter(sentence, forbidden_letters)
```

```
[1 6]: False
```

Exercise 5:

Define a function named `only_uses_these_letters`. It will take some letters and a text. It will check if this text uses only these letters. If the text uses only these letters then it will return `True`. If the text uses any other letter which is not one these letters, it will return `False`.

```
[1 7]: 1 def only_uses_these_letters(text, letters):  
2     # loop over the chars in text  
3     for char in text:  
4         # if the char in letters or not  
5         # check if char is an alphabetical letter ->  
5         isalpha()  
6         if char.isalpha() and not char in letters:  
7             return False  
8     else:
```

```
9      return True
```

In cell 17, we define the `only_uses_these_letters` function. We loop over every character in the text. And in line 6, we first check if this character is alphabetical with help of `isalpha()` function. Because if it's not an alphabetical character then we do not need to check it. We are only interested in alphabetical characters. And if it's alphabetical and not in the `letters` then we return `False`.

In line 8, we have the `else` block of the `for` loop. Remember the `for-else` structure. `Else` will only execute if the code in the `for` loop runs successfully. Successfully means without any `break` or `return` statements. And if the `else` block runs, it will return `True`. `True` means that all the letters in this `text` are only consist of the `letters` parameter.

```
[1
8]: 1 sentence = 'check'
      2 letters = 'chek'
      3
      4 only_uses_these_letters(sentence, letters)
```

```
[1
8]: True
```

We call our `only_uses_these_letters` function in cell 18, and it returns `True`. That's because the value of the `sentence` variable only uses the `letters` of 'chek'. Now let's call it with another set of arguments:

```
[1 9]: 1 sentence = 'check this'  
2 letters = 'chek'  
3  
4 only_uses_these_letters(sentence, letters)
```

```
[1 9]: False
```

The function call in cell 19, returns **False**. Why? Because in the value of **sentence** variable there are letters other than the ones in **'chek'**. The letter **'t'** for example. It doesn't exist in the **letters** list.

Exercise 6:

Now let's use the function in Exercise 4, to get the Lorem words only using **'ens'** letters. No other letters.

```
[2 0]: 1 def only_uses_these_letters_lorem(letters):  
2     # read the file  
3     file = open('words.txt')  
4  
5     for line in file:  
6         # split() -> \n  
7         word_list = line.split()  
8  
9         # word is the first element in list  
10        word = word_list[0]  
11  
12        # convert into lower case  
13        word = word.lower()  
14  
15        if only_uses_these_letters(word, letters):
```

In cell 20, line 15, we call the **only_uses_these_letters(word, letters)** function with **word** and **letters** arguments. Since **only_uses_these_letters** is a bool function, it will return **True** if the **word** only uses these **letters**. Then we print the word if it's **True**. That's it. And that's the power of using functions in programming. Now let's call this function:

```
[2
1]: 1 letters = 'ens'
2 only_uses_these_letters_lorem(letters)
```

```
[2
1]: sesse
      nesnes
      senesses
```

Here we get the Lorem words which only use ‘ens’ letters.

Exercise 7:

Define a function named **uses_all**. It will take a text and some letters. If the text uses all of these letter at least once, it will return **True**. And it will return **False** if there is any letter that the text does not use.

```
[2
2]: 1     def uses_all(text, letters):
2         # define a variable to keep the result
3         it_uses_all = True
4
5         for letter in letters:
```

```
6      # if letter is not text -> text is not using
6      this letter
7      if not letter in text:
8          it_uses_all = False
9
10     # loop finishes
11     return it_uses_all
```

In the function in cell 22, we first initialize a variable as `it_uses_all` and assign `True` to it. Then we loop over the letters to check if any of them is not used by the text. This check is done with the ‘`not`’ keyword as: `not letter in text`. If this is `True`, that means there is a letter which does not exist in the text. So we assign `False` to our `it_uses_all` variable. And in line 11 we return the result.

```
[2
3]: 1 letters = 'abd'
      2 text = 'This is a bold sentence.'
      3
      4 uses_all(text, letters)
```

```
[2
3]: True
```

The function call in cell 23 returns `True`, because the `text` uses all of the ‘`abd`’ letters.

```
[2
4]: 1 letters = 'abdf'
      2 text = 'This is a bold sentence.'
      3
      4 uses_all(text, letters)
```

[2
4]: False

The function call in cell 24 returns `False`, because the `text` does not uses all of the '**abdf**' letters. For example there is no 'f' letter in the `text`.

Exercise 8:

Use the function in Exercises 6 and for Lorem letters find the words using all of the letters in '**aei**'.

```
[2  
5]: 1  def uses_all_lorem(letters):  
2      # read the file  
3      file = open('words.txt')  
4  
5      for line in file:  
6          # split() -> \n  
7          word_list = line.split()  
8  
9          # word is the first element in list  
10         word = word_list[0]  
11  
12         # convert into lower case  
13         word = word.lower()  
14  
15         if uses_all(word, letters):  
16             print(word)
```

In cell 25, we define a function named `uses_all_lorem`. It opens the file and reads it line by line with help of a `for` loop. Then in line 15, it calls the `uses_all` function. It passes the `word` and the `letters` into that function and according to the result it

prints the word. Now let's see these words that use all of the letters in 'aei'.

```
[2
6]: 1 letters = 'aei'
      2 uses_all_lorem(letters)
```

```
[2
6]: feugiat
      penatibus
      parturient
      venenatis
      vitae
      aliquet
      sapien
      viverra
      vehicula
      habitasse
```

Exercise 9:

Use the functions defined in Exercise 4 and Exercise 6 to find the Lorem words which use only “fir” letters and use them all. The function name will be **only_uses_all**.

```
[2
7]: 1 def only_uses_all(letters):
      2     # read the file
      3     file = open('words.txt')
      4
      5     for line in file:
      6         # split() -> \n
      7         word_list = line.split()
      8
      9         # word is the first element in list
```

```
10     word = word_list[0]
11
12     # convert into lower case
13     word = word.lower()
14
15     # call the functions in E4 and E6
16     if only_uses_these_letters(word, letters)
17     and uses_all(word, letters):
18         print(word)
```

In cell 27, line 16, we call the functions which we defined in Exercises 4 and 6. They are both bool functions and each function tells us a result. The `only_uses_these_letters(word, letters)` function call tells us that the `word` only uses these `letters`. And the `uses_all(word, letters)` function call tells us that the `word` uses all of these `letters`. If both cases are `True`, then we are sure that the `word` uses only these `letters` and uses them all.

```
[2
8]: 1 letters = 'fir'
      2
      3 only_uses_all(letters)
```

```
[2
8]: frifirri
```

We call our `only_uses_all` function in cell 28. And we get only one word from the Lorem list. This word is “**frifirri**” which only uses the letters in “**fir**” and uses them all.

Exercise 10:

If the letters in a word is in alphabetical order we call it as an **ordered_word**.

Define a function named **is_ordered_word**. It will take a string parameter as **text**. And it will return **True** if it is an ordered word.

Some examples of ordered words: **abc**, **agor**, **def**.

```
[2
9]: 1     def is_ordered_word(word):
2         # previous letter is the first letter in word
3         prev = word[0]
4
5         for letter in word:
6             # check if prev is smaller
7             if letter < prev:
8                 return False
9
10        # reassign the previous as the current
11        # letter
12        prev = letter
13
14        # return True -> because loop has completed
15        return True
```

In the **is_ordered_word** function, for each letter we keep track of the previous one. We first initialize the **prev** variable as the first letter in the word: **prev = word[0]**. Then we set a **for** loop for the letters in the word. In line 7, we check if the letter is less than the **prev** value as : **letter < prev**. This should never become **True** if the word is an ordered word. Because in ordered

words it should be just the opposite. Every previous letter should be less than the next one. And if it's **True** than we return **False**. If it's not **True**, then in line 11 we update the value of the **prev** variable as the current letter. Because for the next iteration this letter will be the previous one. If we do not encounter the **return** in line 8 in any iteration, this means the word is an ordered word, and we can return **True** in line 14.

Let's check some words to see if they are ordered words or not:

```
[3  
0]: 1 word = 'agor'
```

```
2
```

```
3 is_ordered_word(word)
```

```
[3  
0]: True
```

```
[3  
1]: 1 word = 'python'
```

```
2
```

```
3 is_ordered_word(word)
```

```
[3  
1]: False
```

```
[3  
2]: 1 word = 'hnty'
```

```
2
```

```
3 is_ordered_word(word)
```

```
[3  
2]: True
```

Exercise 11:

Use function in Exercise 9 to find out which Lorem words are ordered.

```
[3
3]: 1  def is_ordered_word_lorem():
2      # read the file
3      file = open('words.txt')
4
5      for line in file:
6          # split() -> \n
7          word_list = line.split()
8
9          # word is the first element in list
10         word = word_list[0]
11
12         # convert into lower case
13         word = word.lower()
14
15         # check if this word is an ordered word
16         if is_ordered_word(word):
17             print(word)
```

In cell 33, we define our `is_ordered_word_lorem` function. It opens the file and loops over the every line in it. Splits the line, get the first item in it, which is the `word`. Then converts the `word` into lowercase. These are the operations we have been doing all over this project. In line 16, we call the `is_ordered_word(word)` function to see if the word is an ordered one. If it is, then we print it.

Now let's see which words in the Lorem list are ordered:

```
[3 4]: 1 is_ordered_word_lorem()
```

```
[3 4]: in
at
in
eu
et
dis
ac
ex
a
est
```

Before finalizing this project there is one more case we should see. Let's try to see if the word '**Nam**' is an ordered word. Let's make an educated guess. What do you think? Is it ordered?

```
[3 5]: 1 word = 'Nam'
2
3 is_ordered_word(word)
```

```
[3 5]: True
```

If your answer was 'No it is not ordered.' then you failed. Why? Because the letter '**N**' is less than the letter '**a**'. That's because the ASCII value for the letter '**N**' is less than the one for the letter '**a**'. Remember we talked about ASCII values when did String Comparison. Let's see this in a new cell and be sure about it:

```
[3 6]: 1 # if 'N' is less than 'a'
```

2 'N' < 'a'

[3
6]: True

And that's it. We finished our second project. We did lots of exercises on functions and strings. Now you should have a much better understanding of functions and how and why to use them. You are now aware of the power of functions and Functional Programming approach. A power that you should always use in your programming life.

OceanofPDF.com

13. Assignment 2 – Words

We finished our second project, Project 2 - Words. Now it's your turn to recreate the same project. This chapter is the assignment on the Project 2 - Words.

In this assignment you will search for words based on predefined criteria. For instance, which words does not include any vowels or the letters of which words are in alphabetical order.

The words are from [Lorem Ipsum](#). Lorem Ipsum is simply dummy text of the printing and typesetting industry.

To complete the assignment, follow the instructions where you see a **#TODO** statement. You can also download the assignment Jupyter file in the [Github repository](#) of the book. You can find the solutions in the previous chapter.

The Assignment

Complete the code with respect to **#TODO** statements.

Preparation:

To read the words:

open() function

- `open(file, mode)`
- mode:

- r: read (default)
- a: append
- w: write
- x: create

```
[1: 1 # open()
]: 2 file = open('words.txt')
```

```
[2: 1 # let's read the first line
]: 2 file.readline()
```

```
[2: 1 'Lorem\n'
]:
```

```
[3: 1 # let's read the second line
]: 2 file.readline()
```

```
[3: 1 'ipsum\n'
]:
```

Special Chars:

- \n : new line character
- \r : carriage return (enter)

```
[4: 1 # run readline for 3rd time
]:
```

```
2 # assign the result to a variable
3 line = file.readline()
4 line
```

```
[  
4 'dolor\n'  
]:
```

```
[  
5 1 # print the first 6 lines
]:  
2 file = open('words.txt')
3
4 # we need index -> enumerate
5 for index, line in enumerate(file):
6     if index <= 5:
7         print(line)
```

```
[  
5 Lorem
]:  
 ipsum
 dolor
 sit
 amet
 consectetur
```

```
[  
6 1 # remove new lines
]:  
2 file = open('words.txt')
3
4 for index, line in enumerate(file):
5     if index <= 5:
```

```
6      # split() -> splits the text from space chars
6      (space, \n, \r)
7      word_list = line.split()
8      print(word_list)
```

```
[  
6      ['Lorem']  
]:  
      ['ipsum']  
      ['dolor']  
      ['sit']  
      ['amet']  
      ['consectetur']
```

Exercise 1:

We will print the words which have number of letters more than 10.

```
[  
7      1      # read the file  
]:  
      2      # TODO - Read the words into a file  
      3  
      4      for line in file:  
      5  
      6      # split() -> \n  
      7      word_list = # TODO - split the line  
      8  
      9      # word is the first element in list  
     10      word = # TODO - get the first element of list  
     11  
     12      # check number of letters  
     13      # TODO - check if the length of the letter is  
             greater than 10
```

```
[7]: consectetur
      pellentesque
      sollicitudin
      scelerisque
      ullamcorper
      condimentum
      Suspendisse
```

Exercise 2:

Define a new function. This function will search for the words which have no vowels in it. The vowels are **a**, **e**, **i**, **o** and **u**.

```
[8]: 1 def check_for_vowels():
      2
      3     # read the file
      4     # TODO - Read the words into a file
      5
      6     for line in file:
      7
      8         # split() -> \n
      9         # TODO - split the line
     10
     11         # word is the first element in list
     12         # TODO - get the first element of list
     13
     14         # check if included vowel
     15         # TODO - check if the word not in vowels
```

```
[9]: 1 check_for_vowels()
[10]:
```

```
[9]: In  
]: trylyk  
Antryl
```

As you see in the output of cell 9, our `check_for_vowels()` functions have a bug in it. It returns the words with capital letter ‘I’, which is also a vowel. That’s because we didn’t consider a case for letters being lower case. Let’s redefine our function and fix this bug.

```
[1]: 1 def check_for_vowels():  
2  
3     # read the file  
4     # TODO - Read the words into a file  
5  
6     for line in file:  
7  
8         # split() -> \n  
9         # TODO - split the line  
10  
11        # word is the first element in list  
12        # TODO - get the first element of list  
13  
14        # convert into lower case  
15        # TODO - convert the word into lower case  
16  
17        # check if included vowel  
18        # TODO - check if the word not in vowels
```

```
[1]: 1 check_for_vowels()
```

[1]
1]: trylyk

Exercise 3:

Get the words which includes the letters 'ae' together and in this order.

[1]
2]:

```
1     def check_for_ae():
2
3         # read the file
4         # TODO - Read the words into a file
5
6         for line in file:
7
8             # split() -> \n
9             # TODO - split the line
10
11            # word is the first element in list
12            # TODO - get the first element of list
13
14            # convert into lower case
15            # TODO - convert the word into lower case
16
17            # check if included vowel
18            # TODO - check if 'ae' is in the word
```

[1]
3]:

```
1 check_for_ae()
```

[1]
3]:

```
vitae
praesent
aenean
maecenas
```

Exercise 4:

We will define a bool function called **is_any_forbidden_letter**. It will take letters which are forbidden and a text as its parameters. It will check if this text includes any of these forbidden letters. And return either **True** or **False**.

```
[1 4]: 1 def is_any_forbidden_letter(text, forbidden_letters):  
2  
3     for letter in text:  
4         # TODO - check if letter in forbidden letters  
5         # TODO - return true  
6     else:  
7         # TODO - return false
```

```
[1 5]: 1 sentence = 'This is a forbidden sentence.'  
2 forbidden_letters = 'ae'  
3 # call the function with these arguments  
4 is_any_forbidden_letter(sentence, forbidden_letters)
```

```
[1 5]: True
```

```
[1 6]: 1 sentence = 'This is the second forbidden sentence.'  
2 forbidden_letters = 'xy'  
3  
4 is_any_forbidden_letter(sentence, forbidden_letters)
```

```
[1 6]: False
```

Exercise 5:

Define a function named `only_uses_these_letters`. It will take some letters and a text. It will check if this text uses only these letters. If the text uses only these letters then it will return `True`. If the text uses any other letter which is not one these letters, it will return `False`.

```
[1
7]: 1 def only_uses_these_letters(text, letters):
2
3     for char in text:
4         # if the char in letters or not
5         # check if char is an alphabetical letter ->
6         # isalpha()
7         # TODO - check if char is alphabetical letter
8         # and it is not in letters
9         # TODO - return false
10    else:
11        # TODO - return true
```

```
[1
8]: 1 sentence = 'check'
2 letters = 'chek'
3
4 only_uses_these_letters(sentence, letters)
```

```
[1
8]: True
```

```
[1
9]: 1 sentence = 'check this'
2 letters = 'chek'
3
4 only_uses_these_letters(sentence, letters)
```

[1
9]: False

Exercise 6:

Now let's use the function in Exercise 4, to get the Lorem words only using 'ens' letters. No other letters.

[2
0]:

```
1 def only_uses_these_letters_lorem(letters):  
2  
3     # read the file  
4     # TODO - Read the words into a file  
5  
6     for line in file:  
7  
8         # split() -> \n  
9         # TODO - split the line  
10  
11         # word is the first element in list  
12         # TODO - get the first element of list  
13  
14         # convert into lower case  
15         # TODO - convert the word into lower case  
16  
17         # TODO - check if the word only uses these  
letters
```

[2
1]:

```
1 letters = 'ens'  
2 only_uses_these_letters_lorem(letters)
```

[2
1]:

```
sesse  
nesnes  
senesses
```

Exercise 7:

Define a function named `uses_all`. It will take a text and some letters. If the text uses all of these letter at least once, it will return **True**. And it will return **False** if there is any letter that the text does not use.

```
[2
2]: 1  def uses_all(text, letters):
2
3      it_uses_all = True
4
5      for letter in ..... :
6          # if letter is not in text -> text is not using
7          # this letter
8          # TODO - check if letter is not in text
9          # TODO - assign false to it_uses_all
10
11      # loop finishes
12      # TODO - return it_uses_all
```

```
[2
3]: 1  letters = 'abd'
2  text = 'This is a bold sentence.'
3
4  uses_all(text, letters)
```

```
[2
3]: True
```

```
[2
4]: 1  letters = 'abdf'
2  text = 'This is a bold sentence.'
3
4  uses_all(text, letters)
```

[2
4]:

False

Exercise 8:

Use the function in Exercises 6 and for Lorem letters find the words using all of the letters in 'aei'.

[2
5]:

```
1  def uses_all_lorem(letters):  
2  
3      # read the file  
4      # TODO - Read the words into a file  
5  
6      for line in file:  
7  
8          # split() -> \n  
9          # TODO - split the line  
10  
11         # word is the first element in list  
12         # TODO - get the first element of list  
13  
14         # convert into lower case  
15         # TODO - convert the word into lower case  
16  
17         # TODO - check if the word uses all the  
letters
```

[2
6]:

```
1 letters = 'aei'  
2 uses_all_lorem(letters)
```

[2
6]:

feugiat
penatibus
parturient

```
venenatis
vitae
aliquet
sapien
viverra
vehicula
habitasse
```

Exercise 9:

Use the functions defined in Exercise 4 and Exercise 6 to find the Lorem words which use only “fir” letters and use them all. The function name will be `only_uses_all`.

```
[2
7]: 1  def only_uses_all(letters):
2
3      # read the file
4      # TODO - Read the words into a file
5
6      for line in file:
7
8          # split() -> \n
9          # TODO - split the line
10
11         # word is the first element in list
12         # TODO - get the first element of list
13
14         # convert into lower case
15         # TODO - convert the word into lower case
16
17         # TODO - check if the word uses only these
18         # letters and uses them all
```

```
[2  
8]: 1 letters = 'fir'  
2  
3 only_uses_all(letters)
```

```
[2  
8]: frifirri
```

Exercise 10:

If the letters in a word is in alphabetical order we call it as an **ordered_word**.

Define a function named **is_ordered_word**. It will take a string parameter as text. And it will return **True** if it is an ordered word.

Some examples of ordered words: abc, agor, def.

```
[2  
9]: 1 def is_ordered_word(word):  
2  
3     # previous letter is the first letter in word  
4     prev = # TODO - assign the first letter in  
5     word to prev  
6  
7     for letter in word:  
8  
8         # check if prev is smaller  
9         # TODO - check if the letter is smaller  
10        than prev  
11        # TODO - return false  
12  
12        # reassign the previous as the current  
13        letter  
13        prev = # TODO - assign the current letter  
13        to prev
```

```
14
15      # return True -> because loop has completed
16      return True

[3
0]: 1 word = 'agor'
2
3 is_ordered_word(word)

[3
0]: True

[3
1]: 1 word = 'python'
2
3 is_ordered_word(word)

[3
1]: False

[3
2]: 1 word = 'hnty'
2
3 is_ordered_word(word)

[3
2]: True
```

Exercise 11:

Use function in Exercise 9 to find out which Lorem words are ordered.

```
[3
3]: 1 def is_ordered_word_lorem():
2
3      # read the file
4      # TODO - Read the words into a file
5
```

```
6  for line in file:
7
8      # split() -> \n
9      # TODO - split the line
10
11     # word is the first element in list
12     # TODO - get the first element of list
13
14     # convert into lower case
15     # TODO - convert the word into lower case
16
17     # check if this word is an ordered word
18     # TODO - check if the word is an ordered
word
```

```
[3
4]: 1 is_ordered_word_lorem()
```

```
[3
4]: in
at
in
eu
et
dis
ac
ex
a
est
```

```
[3
5]: 1 word = 'Nam'
2
3 is_ordered_word(word)
```

```
[3
5]: True
```

```
[3 6]: 1 # if 'N' is less than 'a'  
2 'N' < 'a'
```

```
[3 6]: True
```

OceanofPDF.com

14. List

What is a List?

A **list** is a sequence used to store multiple items in a single variable. **List** is one of four built-in data types in Python used to store collections of data. The other three are **Dictionary**, **Tuple** and **Set**. We will see each of them in the succeeding chapters. In most of other programming languages there are Arrays, while in Python we have **Lists**.

Lists are created using a pair of square brackets `[]`. And the items in a list are separated by commas. Since Lists are of sequence types, you can access their items with **indices**.

Let's define a **list** with 4 items in it:

```
[  
1  1 my_list = [10, 20, 30, 40]  
]:  
2  my_list
```

```
[  
1  [10, 20, 30, 40]  
]:
```

In cell 1, we define a list with name **my_list** and it has 4 numbers in it which are 10, 20, 30 and 40. Let's define another one:

```
[  
2  1 letters = ['a', 'b', 'c', 'd', 'e']  
]:
```

```
[2] print(letters)
```

```
[2]: ['a', 'b', 'c', 'd', 'e']
```

The list in cell 2 is called `letters` and has 5 strings in it. This is a list of only strings. Now let's print its type to see if it is really a list:

```
[3] 1 type(letters)
```

```
[3]: list
```

Let's define a list which has items of different types. Some items will be numbers and some will be strings. Even there will be a bool item in it.

```
[4] 1 mixed_list = [2, 'A', 'B', True, 'mail']
```

```
[4]: 2 mixed_list
```

```
[4]: [2, 'A', 'B', True, 'mail']
```

As you see in cell 4, a list may have items of different types in it. Even it can include other lists. Which means we can create **nested lists** in Python. Let's do it:

```
[5 1 nested_list = ['city', 'name', 4, True, [1, 2, 3], 'A',
]: 2 nested_list
[5
5 1 ['city', 'name', 4, True, [1, 2, 3], 'A', ['a', 'b']]
]:
```

The list in cell 5 have items which are also lists. For example the item `[1, 2, 3]` is a list. Now let's get some items of the **nested_list**:

```
[6 1 # item at index 0
]: 2 nested_list[0]
```

```
[6
6 1 'city'
]:
```

```
[7 1 # item at index 3
]: 2 nested_list[3]
```

```
[7
7 1 True
]:
```

```
[8 1 # item at index 4
]: 2 nested_list[4]
```

```
[8
8 1 [1, 2, 3]
```

```
[1]: [9: 1 # item at index 6
]: 2 nested_list[6]
[9: 3 ['a', 'b']
]:
```

We know that the item at index 4 is a list. Now, let's try to access the items in that nested list. To do this, we will be using **double indexing** as `[][]`:

```
[1: 1 # access the items in the nested lists
0: 2 # double indexing
  3 nested_list[4][2]
[1: 3
0: ]:
```

In cell 10, we first get the item at index 4 in the list as: `nested_list[4]`. This gives us the nested list of `[1, 2, 3]`. Now we can get its items with index. And to get the item at index 2 we use: `nested_list[4][2]`. Which returns 3.

Let's do another example to get the items of the nested list. This time let's first see the nested list.

```
[1: 1 # item at index 6 is a nested list
1: 2 nested_list[6]
[1: 3 ['a', 'b']
1: ]:
```

```
[1]: 1 # get the item 0 at of the nested list
[2]: 2 nested_list[6][0]

[1]: 'a'
```

Sometimes we need an empty list, which is a list with no items in it. To create an empty list we simply use a pair of square brackets with nothing in it as: [].

```
[1]: 1 # Empty list
[3]: 2 empty = []
[3]: 3 empty
```



```
[1]: []
[3]: []
```

Lists are Mutable

In the Chapter 11 - Strings, we learned that Strings are Immutable. However the **Lists are Mutable**. That means you can change any of its items without reassigning it. Let's see it with examples:

```
[1]: 1 cars = ['audi', 'Ford', 'Mazda']
[4]: 2 cars
```



```
[1]: []
[4]: ['audi', 'Ford', 'Mazda']
```

In cell we define a list named `cars`, and it has 3 car brands in it. But there is a typo in the first item, ‘`audi`’. It should start with a

capital letter ‘A’. Let’s fix it:

```
[1 5]: 1 # mutate the cars list
2 cars[0] = 'Audi'
3 cars
```

```
[1 5]: ['Audi', 'Ford', 'Mazda']
```

As you see in cell 15, we mutate the cars list. We change the item at index 0 as ‘**Audi**’.

Let’s define another list and change one of its items:

```
[1 6]: 1 nums = [1, 2, 3, 4, 5]
2 nums[4]
```

```
[1 6]: 5
```

```
[1 7]: 1 nums[4] = '55555'
2 nums
```

```
[1 7]: [1, 2, 3, 4, '55555']
```

If you try to access an item with an index which does not exist in the list, you will get **IndexError**. Let’s see an example:

```
[1 8]: 1 # Access an index which does not exist
2 # IndexError: list index out of range
3 nums[6]
```

```
[1 8]: IndexError: list index out of range
```

For our `nums` list, the maximum index is `4`. And as you see in cell 18, we got an `IndexError` when we try to get item at index `6`, which does not exist.

As a final example, let's get items in the `cars` list with negative indices. First let's remember the positive and negative indices on the `cars` list:

<code>cars</code>	<code>'Audi'</code>	<code>'Ford'</code>	<code>'Mazda'</code>
<code>index -----></code>	<code>0</code>	<code>1</code>	<code>2</code>
<code>negative index <-----</code>	<code>-3</code>	<code>-2</code>	<code>-1</code>

Table 14-1: Positive and Negative Indices on cars list

[1
9]: 1 cars[-1]

[1
9]: 'Mazda'

[2
0]: 1 cars[-2]

[2
0]: 'Ford'

[2
1]: 1 cars[-3]

[2
1]: 'Audi'

Loop Over a List

It is very easy to loop over a `list` with `for` loops. At each iteration, the `for` loop gives an item in the `list` one by one. Let's loop over the `cars` list, to print its items:

```
[2]: 1 # loop over cars list
      2 for car in cars:
      3     print(car)
```

```
[2]: Audi
      Ford
      Mazda
```

Let's say we need to print the index for each car. Remember that we use `enumerate()` function to get the index in the `for` loop.

```
[2]: 1 # we need the index -> enumerate()
      2 for index, car in enumerate(cars):
      3     print("{} - {}".format(index, car))
```

```
[2]: 0 - Audi
      1 - Ford
      2 - Mazda
```

Now, let's say we want to change the items in the loop. To do this, we will use our `nums` list which we defined in cell 16. Let's redefine it and loop over it. In the loop we will multiply each item by 10 and print the result.

```
[2 4]: 1 nums = [1, 2, 3, 4, 5]
2 nums
```

```
[2 4]: [1, 2, 3, 4, 5]
```

```
[2 5]: 1 # loop over the list -> multiply elements by 10
2 for num in nums:
3     num *= 10
4     print(num)
```

```
[2 5]: 10
20
30
40
50
```

In cell 25, we loop over the items of `nums` list. At each iteration we get one item as `num`. And we multiply this `num` by 10 and reassign back to itself. Remember we call this operation as a shorthand multiplication: `num *= 10`. Then we print the new value of `num`.

Here comes the question: What happens to the `nums` list when we change its items in the `for` loop? To see the answer let's print the `nums` list one more time:

```
[2 6]: 1 nums
```

```
[2 6]: [1, 2, 3, 4, 5]
```

As you see in cell 26, it hasn't changed. The reason is, we only changed the copies of the items in the loop. Not the actual items in the list. So the list stayed untouched. To be able to mutate the list in the loop we have to call its items with their index numbers.

```
[2] 1 # we want to change the original list in loop
    2 # we need index to update the element -> list
    3 for index, num in enumerate(nums):
        4     # calculate the new num
        5     new_num = num * 10
        6
        7     # update the list via index
        8     nums[index] = new_num
```

In cell 27, line 8, we update the item in the list with the index as: `nums[index] = new_num`. Now the list is mutated. Let's see its final value by printing it again:

```
[2] 1 nums
[8]:
```



```
[2] [10, 20, 30, 40, 50]
[8]:
```

As you see in cell 28, the items in the list are changed. And the list has been mutated.

Let's do one more example. Remember our list of cars. Now we want to change every item of this list by adding the text of 'Car' after it. For example the item 'Audi' will be 'Audi Car'.

```
[2] 1 # cars list
[9]:
```

```
2 # we will update each car as ' Car' -> Audi Car,  
2 Ford Car, Mazda Car  
3 for index, car in enumerate(cars):  
4     # concatenate ' Car' with the item  
5     cars[index] = car + ' Car'
```

In cell 29, line 5, we changed the item `car` which is at `index` as: `cars[index] = car + ' Car'`. And we mutate the list in this way. Let's print the `cars` list and see the new items:

```
[3  
0]: 1 cars  
[3  
0]: ['Audi Car', 'Ford Car', 'Mazda Car']
```

List Operations

In Python, we are allowed to do addition (`+`) and multiplication (`*`) operations on Lists. They don't work like normal mathematical operations. Lists have their own logic for them. Let's see with examples:

+ **Operator:** The addition (`+`) operator concatenates the lists one after another. The same as we did with Strings.

```
[3  
1]: 1 # define two lists  
2 a = [10, 20, 30]  
3 b = [4, 8, 12]  
4  
5 # now add the lists  
6 c = a + b  
7 print(c)
```

```
[3  
1]: [10, 20, 30, 4, 8, 12]
```

In cell 31, we add two lists and the result is a new list with the items of both being concatenated.

```
[3  
2]: 1 # define three lists  
2 a = [10, 20, 30]  
3 b = [4, 8, 12]  
4 c = ['A', 'B', 'C']  
5  
6 # add them  
7 d = a + b + c  
8 d
```

```
[3  
2]: [10, 20, 30, 4, 8, 12, 'A', 'B', 'C']
```

We define 3 lists in cell 32 and add them. The result is just a large list with all the items from these lists respectively.

* **Operator:** Multiplying a list by a number means, repeating the list by that amount. It's like concatenating the list by itself by that number of times. The same as we did with Strings.

```
[3  
3]: 1 car = 'Car'  
2 3 * car
```

```
[3  
3]: 'CarCarCar'
```

In cell 33 we have an example on Strings. We multiply the string variable `car` by `3` and the result is `'CarCarCar'`. Let's see

the same operation on Lists.

```
[3 4]: 1 cars = ['Audi', 'Ford', 'BMW']  
2 2 * cars
```

```
[3 4]: ['Audi', 'Ford', 'BMW', 'Audi', 'Ford', 'BMW']
```

As you see in cell 34, we multiply the `cars` list by 2 and the result is 2 times of the items in the list, repeated respectively.

```
[3 5]: 1 [5] * 4
```

```
[3 5]: [5, 5, 5, 5]
```

In cell 35, `[5]` is a list and we multiply it by `4`. The result is `[5, 5, 5, 5]`.

```
[3 6]: 1 [11, 22] * 3
```

```
[3 6]: [11, 22, 11, 22, 11, 22]
```

Finally, in cell 36 we multiply the list of `[11, 22]` by `3` and the result is `[11, 22, 11, 22, 11, 22]`.

List Slicing

We covered slicing in very detail in Chapter 11 - Strings. The good news is, List slicing works in the same way as Strings. Let's see it with examples:

```
[3  
7]: 1 my_list = ['a', 'b', 'c', 'd', 'e', 'f']  
2 my_list
```

```
[3  
7]: ['a', 'b', 'c', 'd', 'e', 'f']
```

```
[3  
8]: 1 # we want elements from index 1 to 4 (exc)  
2 my_list[1:4]
```

```
[3  
8]: ['b', 'c', 'd']
```

In cell 38, we get a slice of `my_list` with items from index `1` to index `4`. Since 4 is excluded, the result is `['b', 'c', 'd']`.

```
[3  
9]: 1 # we want first 4 items  
2 # indices: 0, 1, 2, 3  
3 my_list[0:4]
```

```
[3  
9]: ['a', 'b', 'c', 'd']
```

In cell 39, we get the first 4 items. These are the items at indices of 0, 1, 2, and 3. And the slice is `my_list[0:4]`. The starting index is 0 and we already know that 0 is the default value for start. So we can omit it as follows:

```
[4  
0]: 1 # omit 0 for start  
2 my_list[:4]
```

```
[4  
0]: ['a', 'b', 'c', 'd']
```

We get the same result in cells 39 and 40.

Let's say we want to get all the items in the list. Remember we omit all 3 parameters; start, end, step. We leave them as default values as: `[:]`. And this slice gives us a copy of the complete list. Here it is:

```
[4  
1]: 1 # get all items of list  
2 my_list[:]
```

```
[4  
1]: ['a', 'b', 'c', 'd', 'e', 'f']
```

Since the complete slice is a copy we can assign it to a new variable. Actually this is a very common way of copying lists in Python. Let's do it in the next cell:

```
[4  
2]: 1 my_list_copy = my_list[:]  
2 my_list_copy
```

```
[4  
2]: ['a', 'b', 'c', 'd', 'e', 'f']
```

Now let's say, we need the items whose index numbers are even numbers. The first index is 0 and it's an even number. If we keep a step size of 2, up to the end, we will get all the items with even indices:

```
[4  
3]: 1 # get the items with even indices  
2 # 0, 2, 4, ...  
3 my_list[::2]
```

```
[4  
3]: ['a', 'c', 'e']
```

Now let's say we need the items that have odd index numbers. This time we have to start from 1, and again the step size is 2. We will go up to the end, so we can leave the end parameter as default:

```
[4  
4]: 1 # get the elements with odd index  
2 # 1, 3, 5 ...  
3 my_list[1::2]
```

```
[4  
4]: ['b', 'd', 'f']
```

As a final example, let's say we need to copy the entire list in reverse order. We can achieve this with negative step size. Remember, the negative step size means “reverse order”.

```
[4  
5]: 1 # copy the reverse of the list  
2 my_list_copy_reverse = my_list[::-1]  
3 my_list_copy_reverse
```

```
[4  
5]: ['f', 'e', 'd', 'c', 'b', 'a']
```

Copy of a List and the IDs

Copying a list is creating a new list with the same items in it. In this section we will copy lists and we will make sure that the original list and its copy are two different objects. Let's start.

```
[4  
6]: 1 my_list = ['a', 'b', 'c', 'd', 'e', 'f']
```

```
2 my_list
```

```
[4 6]: ['a', 'b', 'c', 'd', 'e', 'f']
```

```
[4 7]: 1 # copy the list with complete slice
2 my_list_copy = my_list[:]
3 my_list_copy
```

```
[4 7]: ['a', 'b', 'c', 'd', 'e', 'f']
```

In cell 47, we copy the `my_list` with complete slicing. And the name of the new list is `my_list_copy`. Now let's modify the new list by changing its items at indices 0 and 1:

```
[4 8]: 1 # mutate the new list -> if it affects the original list
2 my_list_copy[0] = 'A'
3 my_list_copy[1] = 'B'
```

We mutated the `my_list_copy` list in cell 48. Now let's print the two lists together to see their items:

```
[4 9]: 1 print('My List:', my_list)
2 print('My List Copy', my_list_copy)
```

```
[4 9]: My List: ['a', 'b', 'c', 'd', 'e', 'f']
      My List Copy ['A', 'B', 'c', 'd', 'e', 'f']
```

As we see in cell 49, the original `my_list` stayed untouched while the copy list, `my_list_copy`, has been mutated. Because when we copy the list, Python gives us a completely new variable.

id() method: To be able to make sure that two variables are different, we use the built-in `id()` method. The `id()` method returns the identity of the object. In Python, each object has a unique identity number which is called its ‘`id`’. It is an integer and it remains constant for that object throughout its lifetime. You can think of the `id` as the **memory address** of an object.

```
[5  
0]: 1 # get the id of my_list  
2 id(my_list)
```

```
[5  
0]: 1989929659136
```

```
[5  
1]: 1 # get the id of my_list_copy  
2 id(my_list_copy)
```

```
[5  
1]: 1989929662400
```

In cells 50 and 51, we get the ids of variables `my_list` and `my_list_copy`. Now, let’s check if they are the same object. Being the same object means having the same id number.

```
[5  
2]: 1 # check if they are the same object  
2 id(my_list) == id(my_list_copy)
```

```
[5  
2]: False
```

As you see in cell 52, their id numbers are different. So they are not the same object.

List Methods

In Python, we have built-in methods which operate on Lists. In this section we will cover some of them. One quick note: the term **“in-place”** means **mutation**. The object gets mutated if the operation is in-place.

Built-in List Methods:

- **append()**: adds an element at the end of the list (in-place)
- **insert()**: adds an element at specified index (in-place)
- **extend()**: concatenates a list to another (in-place)
- **sort()**: sorts the list (in-place)
- **sorted()**: sorts the list and returns a new list (does not mutate the original list)

Let's see how the list methods work with examples:

append(): Adds an element at the end of the list.

```
[5  
3]: 1 letters = ['a', 'b', 'c', 'd', 'e']  
     2 letters
```

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

```
[5  
4]: 1 letters.append('f')  
     2 letters
```

```
[5  
4]: ['a', 'b', 'c', 'd', 'e', 'f']
```

In cell 54, we append the letter ‘f’ at the end of the **letters** list. Let’s append more items:

```
[5  
5]: 1 # one at a time  
2 letters.append('a')  
3 letters.append('x')  
4 letters.append('t')  
5 letters
```

```
[5  
5]: ['a', 'b', 'c', 'd', 'e', 'f', 'a', 'x', 't']
```

insert(): Adds an element at the specified position. The syntax is: **list.insert(index, element)**. It insert the element at the specified index.

```
[5  
6]: 1 vowels = ['a', 'e', 'i', 'o', 'u']  
2 vowels
```

```
[5  
6]: ['a', 'e', 'i', 'o', 'u']
```

```
[5  
7]: 1 # insert 'E' after 'e'  
2 # at index 2  
3 vowels.insert(2, 'E')  
4 vowels
```

```
[5  
7]: ['a', 'e', 'E', 'i', 'o', 'u']
```

In cell 57, we insert the letter ‘E’ at index `2` into the `vowels` list as: `vowels.insert(2, 'E')`.

`extend()`: Adds the elements of a list to the end of the current list. In other words, it concatenates a list to another one.

```
[5  
8]: 1 # define two different lists  
2 letters = ['a', 'b', 'c', 'd', 'e']  
3 t = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
[5  
9]: 1 # extend the letters list with t  
2 letters.extend(t)
```

```
[6  
0]: 1 # print the final value of letters  
2 letters
```

```
[6  
0]: ['a', 'b', 'c', 'd', 'e', 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In cell 59, we extend the `letters` list with list `t`. And the final value of the `letters` list the concatenation of elements from both lists respectively.

`sort()`: Sorts the list in-place, which means the list gets mutated when we call the `sort()` method on it. So be careful when you use it.

```
[6  
1]: 1 # define a list with unsorted elements  
2 t = [4, 2, 3, 1, 5, 6, 9, 8, 7]  
3 # sorts the list in-place  
4 t.sort()  
5 t
```

```
[6 1]: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

In cell 61, we define a list `t` with unsorted elements in it. Then we call the `sort()` method on this list. And we see that the list `t` has been mutated when we print its final value in line 5.

```
[6 2]: 1 # define an unsorted list
2 a = ['c', 'b', 'a', 'd']
3 a.sort()
4 a
```

```
[6 2]: ['a', 'b', 'c', 'd']
```

In cell 62, we sort the list `a` and as you see it has been mutated.

reverse: We can declare the sort direction via the `reverse` parameter. It has two options: `True` or `False`.

- `reverse=False`: sorts in ascending order (default)
- `reverse=True`: sorts in descending order

```
[6 3]: 1 # reverse = False
2 a = ['c', 'b', 'a', 'd']
3 a.sort(reverse=False)
4 a
```

```
[6 3]: ['a', 'b', 'c', 'd']
```

In cell 63, we pass the `reverse` parameter as `reverse=False` and Python sorts the list in **ascending** order.

```
[6 4]: 1 # reverse = True  
2 a.sort(reverse=True)  
3 a
```

```
[6 4]: ['d', 'c', 'b', 'a']
```

In cell 64, we pass the `reverse` parameter as `reverse=True` and Python sorts the list in **descending** order.

sorted(): Sorts the list in the specified order and returns a sorted copy of the list. The syntax is: `sorted(my_list)`. The difference between `sort()` and `sorted()` is, `sort()` mutates the list, while `sorted()` does not.

```
[6 5]: 1 a = ['c', 'b', 'a', 'd']  
2 a
```

```
[6 5]: ['c', 'b', 'a', 'd']
```

```
[6 6]: 1 # sorted(<list>)  
2 b = sorted(a)  
3 b
```

```
[6 6]: ['a', 'b', 'c', 'd']
```

In cell 66, we use the `sorted()` function to get a sorted copy of the list `a`. And we name this new sorted list as `b`. As you see in the output, `b` is sorted in ascending order. To prove that `a` has not changed let's print its value again:

```
[6 7]: 1 # a is unchanged  
2 a
```

```
[6 7]: ['c', 'b', 'a', 'd']
```

As you see in cell 67, the value of the list `a` has not changed. Let's do another example. This time let's use the `reverse` parameter.

```
[6 8]: 1 # sort in reverse order  
2 d = sorted(a, reverse=True)  
3 d
```

```
[6 8]: ['d', 'c', 'b', 'a']
```

In cell 68, we pass the `reverse=True` parameter to the `sorted()` function and it returns a new sorted list, which we name as `d`. And as you see in the output, `d` is sorted in descending order.

Deleting Elements from a List

We have various ways to delete elements from a List. Let's see them one by one.

1- `pop()`

If we know the index to delete, we use the `pop()` method. We just pass the index of the item we want to delete as the argument. The `pop()` method removes the element from the list and returns that element.

```
[6 9]: 1 a_list = ['x', 'y', 'z', 't']  
2 a_list
```

```
[6 9]: ['x', 'y', 'z', 't']
```

```
[7 0]: 1 # delete element at index 2  
2 deleted_element = a_list.pop(2)  
3 deleted_element
```

```
[7 0]: 'z'
```

In cell 70, we delete the element at index 2 from `a_list`. We know that the element at index 2 is ‘z’. When we call the `pop()` method as `a_list.pop(2)` it removes the item at index 2 and returns it back to us. And we assign the result to the `deleted_element` variable, which is ‘z’.

Now if we print the value of `a_list` again we will see that the letter ‘z’ has been deleted:

```
[7 1]: 1 # print a_list after we delete an element  
2 a_list
```

```
[7 1]: ['x', 'y', 't']
```

If you call the `pop()` method without passing any index parameter to it, by default, it will delete the **last element** in the list. Let’s see an example for this case.

```
[7 2]: 1 del_el = a_list.pop()  
2 del_el
```

```
[7 2]: 't'
```

In cell 72, we call the `pop()` method as: `a_list.pop()`. Since we do not pass any index value as the argument to it, it removes the last element from the list. This element is the letter ‘t’. Now if we print the `a_list` one more time, we will see that ‘t’ has been removed:

```
[7 3]: 1 a_list
```

```
[7 3]: ['x', 'y']
```

2- `del`

If we do not need the deleted element than we can use the `del` keyword. The `del` statement removes the element from the list and returns `None`. Its syntax is: `del my_list[index]`.

```
[7 4]: 1 my_list = ['k', 'l', 'm', 'n', 'o']  
2 my_list
```

```
[7 4]: ['k', 'l', 'm', 'n', 'o']
```

```
[7 5]: 1 # delete element at index 3  
2 del my_list[3]
```

In cell 75, we delete the element at index 3. Now let's see the final value of `my_list`. You will see, the letter '`n`' is not in the list anymore, because it was the item at index 3.

```
[7]  
[6]: 1 # my_list after deletion  
2 my_list
```

```
[7]  
[6]: ['k', 'l', 'm', 'o']
```

We can also delete a slice with `del` statement. Let's assume we want to delete the elements from index 1 to 5. This is the slice of `my_list[1:5]` and the elements are '`l`', '`m`', '`n`' and '`o`'.

```
[7]  
[7]: 1 # redefine the list  
2 my_list = ['k', 'l', 'm', 'n', 'o', 'p', 'r']  
3 my_list
```

```
[7]  
[7]: ['k', 'l', 'm', 'n', 'o', 'p', 'r']
```

```
[7]  
[8]: 1 # delete indices from 1-5  
2 del my_list[1:5]
```

In cell 78, we delete the elements from index 1 to 5 as: `del my_list[1:5]`. Now let's see if these elements have been removed:

```
[7]  
[9]: 1 # my_list after deletion  
2 my_list
```

```
[7]  
[9]: ['k', 'p', 'r']
```

3- `remove()`

If we know the element itself, which we want to delete we can use the `remove()` method. Its syntax is: `remove(element)`. Here the parameter is not an index it's the element itself. The `remove(element)` method call will remove the element from the list.

```
[8
0]: 1 # redefine my_list
      2 my_list = ['k', 'l', 'm', 'n', 'o', 'p', 'r']
      3 my_list
```

```
[8
0]: ['k', 'l', 'm', 'n', 'o', 'p', 'r']
```

```
[8
1]: 1 # remove element 'm' from my_list
      2 my_list.remove('m')
```

```
[8
2]: 1 # my_list after remove
      2 my_list
```

```
[8
2]: ['k', 'l', 'n', 'o', 'p', 'r']
```

In cell 81, we remove the element ‘m’ as: `my_list.remove('m')`. In cell 82, we print `my_list` and see that the letter ‘m’ has been removed.

```
[8
3]: 1 # remove two more elements
      2 my_list.remove('o')
      3 my_list.remove('r')
```

```
[8 4]: 1 # my_list after remove
2 my_list
```

```
[8 4]: ['k', 'l', 'n', 'p']
```

In cell 83, we remove two elements, ‘o’ and ‘r’ one by one.

If you call the `remove()` method with an element which does not exist in the list, you will get **ValueError**. Let’s see it:

```
[8 5]: 1 # try to remove an element which does not exist
2 my_list.remove('x')
```

```
[8 5]: ValueError: list.remove(x): x not in list
```

In cell 85, we call the `remove()` method with the element ‘x’. Since `my_list` has no element as ‘x’ we got **ValueError**: `list.remove(x): x not in list`.

List & String

A String is a sequence of characters. A List is a sequence of values. It’s very easy to convert one into another. Let’s start by converting a String to a List. To convert a String variable to a List we use the `list()` constructor. A **constructor** is a special purpose method which is used for instantiating an object. Here the `list()` constructor creates a list object out of a string.

Let’s define a string first and then convert this string to a list:

```
[8  
6]: 1 day = 'Monday'  
2 type(day)
```

```
[8  
6]: str
```

```
[8  
7]: 1 # convert the string into a list  
2 day_letters = list(day)  
3 day_letters
```

```
[8  
7]: ['M', 'o', 'n', 'd', 'a', 'y']
```

In cell 87, we convert the string to a list as: `list(day)`. And it returns the list of characters in the string. Each letters becomes an element in the `day_letters` list. Let's see also the type of `day_letters` to be sure that it's a list:

```
[8  
8]: 1 type(day_letters)
```

```
[8  
8]: list
```

Another way of converting a string into a list is the built-in `split()` method. If we call `string.split()` it will return a list of words in the string.

```
[8  
9]: 1 text = 'today is another day'  
2 text
```

```
[8  
9]: 'today is another day'
```

```
[9 0]: 1 # get the words as a list -> split()  
2 words = text.split()  
3 words
```

```
[9 0]: ['today', 'is', 'another', 'day']
```

In cell 90, we split the string as: `text.split()`. If you do not pass any arguments into the `split()` method, it splits at the whitespace characters by default. And that's what happens in cell 90. It splits the text at the whitespaces and returns a list of the words in it.

We can specify the character at which we want to split. Let's assume we have a text with '-' chars in it and we want to split this text at the '-' chars.

```
[9 1]: 1 # split from another char than space -> -  
2 mailbox = 'spam-spam-ham-spam-ham'  
3 mailbox
```

```
[9 1]: 'spam-spam-ham-spam-ham'
```

```
[9 2]: 1 mails = mailbox.split('-')  
2 mails
```

```
[9 2]: ['spam', 'spam', 'ham', 'spam', 'ham']
```

In cell 92, we split the `mailbox` string at the '-' chars as: `mailbox.split('-')`. And it returns the list of words.

Now let's do the conversion in the opposite direction. We will convert a list into a string this time. One of the simplest way to do this is using the built-in `join()` method. As its name implies, it joins the elements of the list and creates a string. It's just the opposite of the `split()` method.

```
[9  
3]: 1 words = ['monday', 'is', 'the', 'first', 'day']  
2 words
```

```
[9  
3]: ['monday', 'is', 'the', 'first', 'day']
```

```
[9  
4]: 1 # join with '-' char between items  
2 sentence = '-'.join(words)  
3 sentence
```

```
[9  
4]: 'monday-is-the-first-day'
```

In cell 94, we join the elements of the `words` list into a string. And we name this string as `sentence`. The syntax is as: `'-'.join(words)`. Here, `'-'` is the character with which we want to join. Python will put this character in-between each item pairs in the list. And the argument to the `join()` method is the list.

Let's join the list elements again. But this time we will place the whitespace character in-between the items:

```
[9  
5]: 1 # join with whitespace char between items  
2 sentence = ' '.join(words)  
3 sentence
```

```
[9  
5]: 'monday is the first day'
```

As you see `join()` is just the opposite of the `split()` function.

Creating a List with `range()` Function

Ranges are very useful data types especially when we are dealing with sequential numeric data. And it's possible to convert a Range into a List. As you may guess, it is done with the `list()` constructor.

Let's remember the `range` type. Its syntax is: `range(start, end, step)`. Here start is the starting number which is included, end is the ending number which is excluded and step is step size by which the range either increments or decrements.

```
[9  
6]: 1 # range(start, end, step)  
2 # start = 1  
3 # end = 10  
4 # step = 2  
5 range(1, 10, 2)
```

```
[9  
6]: range(1, 10, 2)
```

Let's print the items in the range we define in cell 96:

```
[9  
7]: 1 for i in range(1, 10, 2):  
2     print(i)
```

```
[9 7]:  
1  
3  
5  
7  
9
```

Let's say we need the list of numbers from 2 up to 30 and the step size 3. A list as [2, 5, 8, 11, ..., 30]. It would be difficult to type each number one by one if we try to create this list manually. But the `range()` function do this very easily for us.

```
[9 8]:  
1 # range of numbers from 2 up to 30 and the step size  
3  
2 my_range = range(2, 33, 3)
```

```
[9 9]:  
1 # convert this range into a list  
2 range_list = list(my_range)  
3 range_list
```

```
[9 9]: [2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32]
```

In cell 99, we convert `my_range` variable into the `range_list` variable which is of type list. We use the `list()` constructor for this as: `list(my_range)`.

Let's do another example for converting a range into a list:

```
[10 0]:  
1 # start from 20 up to 100, step size 10  
2 range_two = range(20, 100, 10)  
  
[10 1]:  
1 # convert this range into a list
```

```
2 list_two = list(range_two)
3 print(list_two)
```

[10
1]: [20, 30, 40, 50, 60, 70, 80, 90]

Objects and Values

In programming, it is very important to be sure what your variables (objects) are. Are they completely different objects or do they refer to the same one. What happens when you assign an object to another? We will use the `id()` function to answer most of these questions.

Let's start by defining two string variables. And they will have the same value.

```
[10  
2]: 1 # Two strings with the same value
      2 a = 'apple'
      3 b = 'apple'
```

In cell 102, we define two string variables both have the same value as ‘apple’. Now let's see their ids in the memory.

```
[10  
3]: 1 # the id of variable a
      2 id(a)
```

[10
3]: 2092223467696

```
[10  
4]: 1 # the id of variable b
      2 id(b)
```

[10
4]:

```
2092223467696
```

You might get surprised that you see they have the same id numbers. How is that possible? Didn't we define two separate variables as **a** and **b**? Yes we did actually.

Let's ask Python whether they have the same **values** or not. Remember, we use the equality check (`==`) for **value comparison**.

[10
5]:

```
1 # equality check  
2 a == b
```

[10
5]:

```
True
```

In cell 105, we see that their values are equal, something we know already. Now let's ask Python whether they are the **same variables** or not. This check is the **identity check** and we use the **`id()`** function for it.

[10
6]:

```
1 # identity check  
2 id(a) == id(b)
```

[10
6]:

```
True
```

Interesting, right? Python tells us that they are the same object. In other words, **a** and **b** refers to the same address in the memory, instead being 2 different objects. To understand the reason we first need to know Primitive and Non-Primitive Types in Python.

In Python Types are:

- **Primitive Types:** int, float, string, bool
- **Non-Primitive Types:** list, dict, tuple, set

These two variables, **a** and **b**, have the same values and they are strings (**str**) which is a Primitive Type. Since their values are the same, and they are Primitives, Python does not create two separate variables, instead, it combines both to the same address, to save memory. That's why you see the same id numbers for both of them.

Now let's reassign another value to the variable **b**, and see what happens:

```
[10
7]: 1 # reassign another value to variable b
      2 b = 'orange'
      3 b
```

```
[10
7]: 'orange'
```

Now let's get the ids of **a** and **b** one more time:

```
[10
8]: 1 # the id of variable a
      2 id(a)
```

```
[10
8]: 2092223467696
```

```
[10
9]: 1 # the id of variable b
      2 id(b)
```

[10
9]:

```
2092223339760
```

As you see in cells 108 and 109, the id of variable **a** is still the same number which is **2092223467696**. But the id of **b** has changed. Now it's **2092223339760**. Which means that, from now on, **a** and **b** are not the same object. Their memory addresses are not the same anymore. Since we change the value of **b**, Python creates a new address (id) for it.

Let's do the equality and identity checks one more time for **a** and **b**:

[11
0]:

```
1 # equality check
2 a == b
```

[11
0]:

```
False
```

[11
1]:

```
1 # identity check
2 id(a) == id(b)
```

[11
1]:

```
False
```

So far in this section, the variables we work with were Strings, which are of Primitive types. Now let's do the same operations on the Lists. As we learned above, Lists are Non-Primitive Types.

[11
2]:

```
1 # Define Two Lists
2 x = [1, 2, 3]
3 y = [1, 2, 3]
```

In cell 112, we define two list variables; `x` and `y`. And both have the same set of elements. We can see this when we do the equality check (`==`) in cell 113. Remember the equality check is done on the values.

```
[11  
3]: 1 # equality check  
      2 # == checks for values  
      3 x == y
```

```
[11  
3]: True
```

We can see that their values are equal in cell 113. Now let's print their id numbers:

```
[11  
4]: 1 # id of x  
      2 id(x)
```

```
[11  
4]: 209222259456
```

```
[11  
5]: 1 # id of y  
      2 id(y)
```

```
[11  
5]: 2092222212480
```

The id of `x` is different from the id of `y`. Although they have the same values, Python creates them as 2 distinct variables. Why? Because lists are Non-Primitive types. Let's also prove this with the identity check:

```
[11  
6]: 1 # identity check  
2 # object equality (not their values)  
3 id(x) == id(y)
```

```
[11  
6]: False
```

Since the list is Non-Primitive, Python instantiates (creates) 2 different variables even if they have the same values.

Before ending this section let's summarize the equality and identity checks one more time:

`x == y` (equality check):

It's the **Value Comparison**. It checks the values of two variables.

If it's **True** then we say: 'the value of x is equal to the value of y'

`id(x) == id(y)` (identity check):

It's the **Object Comparison**. It checks two objects (address - id).

If it's **True** then we say: 'x and y are identical'

is Statement

We know that `id(x) == id(y)` is the object comparison. It checks whether x and y are identical or not. In Python we have a more elegant way for identity check: `is` statement. It's syntax is: `x is y`. And it returns `True` if x and y are the same object (identical), `False` otherwise.

```
[11  
7]: 1 # define Two lists  
     2 a = [1, 2, 3, 4, 5]  
     3 b = [1, 2, 3, 4, 5]
```

```
[11  
8]: 1 # value comparison  
     2 a == b
```

```
[11  
8]: True
```

In cell 118, we check the values of two lists, and since their values are the same `a == b` returns `True`.

```
[11  
9]: 1 # object comparison  
     2 id(a) == id(b)
```

```
[11  
9]: False
```

In cell 119, we check if the two lists, `a` and `b`, are the same object via `id(a) == id(b)` expression. And it returns `False`, since Lists are Non-Primitive. Python creates distinct objects for Non-Primitive types even if they have the same values.

```
[12  
0]: 1 # is statement  
     2 # if they are identical  
     3 a is b
```

```
[12  
0]: False
```

In cell 120, we do the object comparison with the `is` statement. We ask Python if ‘`a is b`’ and it returns `False`, because they are two distinct objects.

Let’s do another example with the `int` type now. Since `int` is a Primitive type, we know that the two variables will be identical if they instantiate with the same value.

```
[12
1]: 1 # define 2 integers
      2 # int is Primitive
      3 a = 2
      4 b = 2
```

```
[12
2]: 1 # value comparison
      2 a == b
```

```
[12
2]: True
```

```
[12
3]: 1 # object comparison
      2 a is b
```

```
[12
3]: True
```

In cells 122 and 123, two `int` variables, `a` and `b`, have the same values and they are identical. In other words they are the same variable (object) because ‘`a is b`’ returns `True`.

Now let’s reassign one of them with a different value. We know that, as soon as their values differ, Python separates them into two distinct variables. Let’s see:

```
[12  
4]:  
1 # reassign b  
2 b = 888
```

```
[12  
5]:  
1 # object comparison  
2 a is b
```

```
[12  
5]:  
False
```

We reassign variable **b** with the value of **888** and now ‘**a is b**’ returns **False**. Which means they are different variables.

Aliasing

Aliasing means different names referring to the same object. Let’s assume we have two variables **x** and **y** with the same id numbers. Since their ids are the same, they are identical. In other words they refer to the same object in the memory. We call **x** and **y** as **aliases**.

Aliasing in Primitive Types:

Let’s see aliasing on integers. We already know that **int** is a Primitive Type.

```
[12  
6]:  
1 # assignment in Primitive Types  
2 x = 24  
3 y = x
```

```
[12  
7]:  
1 # object comparison  
2 x is y
```

[12
7]:

```
True
```

In cell 126, we define an int variable `x` with value 24. Then we assign `x` to `y`. Now `y` is also an int variable with value 24. In cell 127, we check if they are the same object via '`x is y`' and it returns `True`. Which means they are **aliases** of the same object.

Now let's reassign the variable `y` with a different value:

[12
8]:

```
1 # reassign y  
2 y = 40
```

[12
9]:

```
1 # object comparison  
2 x is y
```

[12
9]:

```
False
```

After reassignment of `y`, we see that `x` and `y` are not the same object anymore. Python creates a new memory address for the variable `y`. `x` and `y` are not aliases now. As a result, we can say that, **for Primitive Types, reassignment breaks aliasing**. They become two different objects after reassignment.

Aliasing in Non-Primitive Types:

Now, let's see aliasing on Lists. We know that `list` is a Non-Primitive Type.

[13
0]:

```
1 # Assignment in Non-Primitive Types  
2 a = [1, 2, 3, 4, 5, 6]
```

```
3 b = a  
[13 1]: 1 # object comparison  
2 a is b  
[13 1]: True  
[13 2]: 1 # print their values  
2 print('a:', a)  
3 print('b:', b)  
[13 2]: a: [1, 2, 3, 4, 5, 6]  
b: [1, 2, 3, 4, 5, 6]
```

In cell 130, we create a list with name `a` and then assign it to another variable with name `b`. Now both `a` and `b` are lists. In cell 131, do object comparison to see if they are the same object as: `a is b`. And it returns `True`, which means they are the aliases of the same object. In cell 132, we print both lists and see that they have the same values.

Now let's reassign the variable `b` with a new list. And see what happens:

```
[13 3]: 1 # reassign b  
2 b = [10, 20, 30, 40, 50, 60]  
[13 4]: 1 # print their values  
2 print('a:', a)  
3 print('b:', b)
```

[13
4]:

```
a: [1, 2, 3, 4, 5, 6]  
b: [10, 20, 30, 40, 50, 60]
```

In cell 133, we reassign the variable **b**. And when we print the values, we see that the values are different now. They are not identical anymore. So we conclude that; **for Non-Primitive Types, reassignment breaks aliasing**. They become two different objects after reassignment.

What if we **mutate** one of them, instead of reassigning? As we know, mutation is changing a part of the variable, not reassigning it to a completely new value.

[13
5]:

```
1 # Assignment in Non-Primitive Types  
2 a = [1, 2, 3, 4, 5, 6]  
3 b = a
```

[13
6]:

```
1 # object comparison  
2 a is b
```

[13
6]:

```
True
```

In cells 135 and 136 we create a list variable **a**, assign it to the variable **b** and see that they are identical. Nothing new, so far. Let's mutate the variable **b** now:

[13
7]:

```
1 # mutate b  
2 b[0] = 'A'  
3 b[1] = 'B'
```

```
[13  
8]:  
1 # print b  
2 b
```

```
[13  
8]:  
['A', 'B', 3, 4, 5, 6]
```

```
[13  
9]:  
1 # a has been mutated  
2 a
```

```
[13  
9]:  
['A', 'B', 3, 4, 5, 6]
```

We mutate the list **b** in cell 137. We change its elements at index 0 and 1. And when we print the variable **a**, we see that it has also mutated. Let's print both of them side by side to see that they are completely equal:

```
[14  
0]:  
1 # print their values  
2 print('a:', a)  
3 print('b:', b)
```

```
[14  
0]:  
a: ['A', 'B', 3, 4, 5, 6]  
b: ['A', 'B', 3, 4, 5, 6]
```

As you see in the output of cell 140, **a** and **b** have exactly the same values. But we only mutate **b**, we didn't touch the variable **a**. How is this possible? The answer is that, they are still the same object. Let's prove it by using the **is** statement:

```
[14  
1]:  
1 # object comparison  
2 a is b
```

[14
1]:

True

As you see in cell 141, they are identical. Which means they are still the same object. In other words, they are just the aliases of the same object. So we conclude that; **for Non-Primitive Types, mutation does not break aliasing**. They remain referring to the same memory address even if we mutate one of them.

List as a Function Argument

Most of the time, we pass lists as function arguments. And we need to mutate or reassign them inside the function body. In this section, let's see what happens to the original list when you modify it in the function scope.

We will do an example for this section. In this example we will create a list of letters and define a function with name **add_upper_cases**. As its name implies the function will append the uppercase forms of the letters to the list. And we want to know what happens to our original list which we pass to the function as an argument.

Mutate the List inside the function:

[14
2]:

```
1      # Example
2      letters = ['a', 'b', 'c']
3
4      print("before passing to function:", letters)
```

```
5
6  def add_upper_cases(list_of_letters):
7      # mutation
8      list_of_letters.insert(1, 'A')
9      list_of_letters.insert(3, 'B')
10     list_of_letters.insert(5, 'C')
11
12 # call the function with the list
13 add_upper_cases(letters)
14
15 print("before passing to function:", letters)
```

[14
2]:

```
before passing to function: ['a', 'b', 'c']
before passing to function: ['a', 'A', 'b', 'B', 'c',
'C']
```

In cell 142, we define a list variable with name `letters` and assign the value `['a', 'b', 'c']` to it. And we print its initial value.

In line 6, we define a function, `add_upper_cases`. It takes a list as an argument. And it inserts a capital letter after each odd index to the list. For example, the code `list_of_letters.insert(1, 'A')`, inserts letter ‘A’ at index 1 of the `list_of_letters` list.

In line 13, we call this function with `letters` list as the argument as: `add_upper_cases(letters)`. And finally we print the `letters` list one more time in line 15.

The output of the cell 142 shows us that the list has been changed. We see that after the function call, its elements have been increased. We can also see this by just calling the `letters` list in a separate cell:

[14
3]: 1 letters

[14
3]: ['a', 'A', 'b', 'B', 'c', 'C']

Here is the conclusion:

When you pass Non-Primitive Types (List here) as a function argument:

- they **pass by reference** (they pass as the same object)
- if you **mutate** the parameter inside the function, then **original object will be mutated**

Reassign the List inside the function:

Now, instead of mutating it, let's reassign the argument inside the function body.

```
[14  
4]: 1      # Example
      2      letters = ['a', 'b', 'c']
      3
      4      print("before passing to function:", letters)
      5
      6      def add_upper_cases(list_of_letters):
      7          # reassignment
      8          list_of_letters = ['X', 'Y', 'Z']
      9
     10     # call the function with the list
     11     add_upper_cases(letters)
     12
     13     print("before passing to function:", letters)
```

[14
4]:

before passing to function: ['a', 'b', 'c']

before passing to function: ['a', 'b', 'c']

In cell 144, inside the `add_upper_cases` function we reassign the list parameter with a new value. And in the cell output we see that our original `letters` list stays unchanged.

Here is the conclusion:

When you pass Non-Primitive Types (List here) as a function argument:

- they **pass by reference** (they pass as the same object)
- if you **reassign** the parameter inside the function, Python will **create a new object** in the function scope. The original object will not be changed

QUIZ - List

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_List.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *14_List*. Here are the questions for this chapter:

QUIZ - List:

Q1:

Define a function named **sum_of_list**.

The parameter will be a list.

The function will add all the elements in the list and return the summation result.

Expected Output:

```
nums = [1, 2, 3, 4, 5, 'a']
```

```
sum_of_list(nums) -> 15
```

```
[  
1 1 # Q 1:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 nums = [1, 2, 3, 4, 5, 'a']  
8 summation = sum_of_list(nums)  
9 print(summation)
```

```
[  
1 15  
]:
```

Q2:

Define a function named **two_levels_sum**.

The parameter will be a list of two levels (nested list).
The function will add all the elements including the nested lists and return the summation result.

Hints:

- `isdigit()`
- `type()`

Expected Output:

```
a_list = [[5, 8], [1, 4, 7], [10], 3, 'a']  
summation = two_levels_sum(a_list) -> 38
```

```
[  
2 1 # Q 2:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 a_list = [[5, 8], [1, 4, 7], [10], 3, 'a']  
8 summation = two_levels_sum(a_list)  
9 print(summation)
```

```
[  
2 38  
]:
```

Q3:

The function you defined in Q2 can only add lists of two levels.

Now we will convert this into a generic function that can take any levels of nested lists.

Function name will be **nested_sum**.

It will add all the elements of all lists including nested lists and return the overall summation result.

Hints:

- Recursion
- isdigit()
- type()

Expected Output:

```
a_list = [[-5, [9, 1, [6, 2]], 8], [1, 4, 7, [8, 2]], [10, [-1, 2]], 3]
summation = nested_sum(a_list) -> 57
```

```
[ 1 # Q 3:
]: 2
3 # ---- your solution here ----
4
5
6 # call the function you defined
7 a_list = [[-5, [9, 1, [6, 2]], 8], [1, 4, 7, [8, 2]], [10,
7 [-1, 2]], 3]
8 summation = nested_sum(a_list)
9 print(summation)
```

```
[ 3 57
]:
```

Q4:

Define a function named **squares**.

It will calculate the square of each item in the list and append it to a new list.

It will return the new list of squares.

Expected Output:

```
numbers = [1,2,3,4,5]
squares(numbers) -> [1, 4, 9, 16, 25]
```

```
[ 4 1 # Q 4:
]:
```

```
2
3 # ---- your solution here ----
4
5
6 # call the function you defined
7 numbers = [1,2,3,4,5]
8 result = squares(numbers)
9 print(result)
```

```
[  
4  [1, 4, 9, 16, 25]  
]:
```

Q5:

Define a function named **sum_of_squares**.

It will calculate the squares of items first and will add them to create items of a new list.

Each item in the new list will be the sum of items from the old list starting from index 0 up to the item's index.

For example in the new list the item in index 4 is going to be the sum of squares of items at indices 0-1-2-3-4 of parameter list.

And it will return the new list.

Hints:

- use `sum_of_list` function you defined in Q1

Function Call:

```
a_list = [1,2,3,4,5]
result = sum_of_squares(a_list)
print(result)
```

Expected Output:

```
[1, 5, 15, 37, 83]
```

```
[  
5  1 # Q 5:  
]:  
2
```

```
3 # ---- your solution here ----
4
5
6 # call the function you defined
7 a_list = [1,2,3,4,5]
8 result = sum_of_squares(a_list)
9 print(result)
```

```
[5]: [1, 5, 15, 37, 83]
```

Q6:

Define a function that return the difference between items at odd indices (1,3,5...) and items at even indices (0,2,4...).
The function name will be **odd_even_difference**.

Hints:

- index starts from zero
- use lists and indices only
- do not use loop
- use sum_of_list function you defined in Q1

Function Call:

```
a_list = [1,2,3,4,5,6]
diff = odd_even_difference(a_list)
print(diff)
```

Expected Output:

```
3
```

```
[6]: 1 # Q 6:
      2
      3 # ---- your solution here ----
      4
      5
      6 # call the function you defined
```

```
7 a_list = [1,2,3,4,5,6]
8 diff = odd_even_difference(a_list)
9 print(diff)
```

```
[  
6      3  
]:
```

Q7:

Define a function that crop the list parameter.

The function will be **crop_the_list**.

Cropping means remove the first and the last elements from the list.

The function will mutate the original list parameter namely it will modify the list in-place.

It will not return any value.

Test the function and prove it mutates the list which you pass as argument.

Function Call:

```
nums = [1,2,3,4,5,6,7]
print('the list before passing as parameter:', nums)
crop_the_list(nums)
print('the list before passing as parameter:', nums)
```

Expected Output:

the list before passing as parameter: [1, 2, 3, 4, 5, 6, 7]

the list before passing as parameter: [2, 3, 4, 5, 6]

```
[  
7  1      # Q 7:  
]:  
2  
3      # ---- your solution here ----  
4  
5  
6      # call the function you defined  
7  nums = [1,2,3,4,5,6,7]  
8  print('the list before passing as parameter:',  
      nums)
```

```
9     crop_the_list(nums)
10    print('the list before passing as parameter:',  
         nums)
```

[
7
]:

the list before passing as parameter: [1, 2, 3, 4, 5,
6, 7]

the list before passing as parameter: [2, 3, 4, 5, 6]

Q8:

Define a function named **sort_the_list**.

It will take a list and sort type (is_reverse) as parameters.

is_reverse will be boolean and it will decide if the sort will be ascending or descending. It's default value will be False.

- if **is_reverse** is True -> sort will be descending
- if **is_reverse** is False -> sort will be ascending

The function will return the new sorted list.

Hints:

- **sorted()**
- no loops
- do not mutate the original list

Function Call:

```
a = [12, 4, 2, 1, 6, 3, 45]
print('a:', a)
sorted_a = sort_the_list(a, False)
print('sorted - ascending', sorted_a)
sorted_a = sort_the_list(a, True)
print('sorted - descending', sorted_a)
print('a after sort_the_list function call:', a)
```

Expected Output:

```
a: [12, 4, 2, 1, 6, 3, 45]
sorted - ascending [1, 2, 3, 4, 6, 12, 45]
sorted - descending [45, 12, 6, 4, 3, 2, 1]
a after sort_the_list function call: [12, 4, 2, 1, 6, 3, 45]
```

```
[8]: 1 # Q 8:  
[8]: 2  
[8]: 3 # ---- your solution here ----  
[8]: 4  
[8]: 5  
[8]: 6 # call the function you defined  
[8]: 7 a = [12, 4, 2, 1, 6, 3, 45]  
[8]: 8 print('a:', a)  
[8]: 9  
[8]: 10 sorted_a = sort_the_list(a, False)  
[8]: 11 print('sorted - ascending', sorted_a)  
[8]: 12  
[8]: 13 sorted_a = sort_the_list(a, True)  
[8]: 14 print('sorted - descending', sorted_a)  
[8]: 15  
[8]: 16 print('a after sort_the_list function call:', a)
```

```
[8]: a: [12, 4, 2, 1, 6, 3, 45]  
[8]: sorted - ascending [1, 2, 3, 4, 6, 12, 45]  
[8]: sorted - descending [45, 12, 6, 4, 3, 2, 1]  
[8]: a after sort_the_list function call: [12, 4, 2, 1, 6, 3, 45]
```

Q9:

The Consecutive Numbers in Math are actually Lists in Python.

Gauss' Method for Summing Consecutive Numbers is a method that gives you the summation of numbers. And that's how Gauss' Method works:

- First it sorts the list in ascending order
- Then sorts the list in descending order

- Adds the items in both lists at the same index
- Divides the overall summation by 2 (because each element is written 2 times)

Example: Sum of all integers from 1 to 10. 11 is the pairwise sum at each column.

1	2	3	4	5	6	7	8	9	10
10	9	8	7	6	5	4	3	2	1
+	+	+	+	+	+	+	+	+	+
11	11	11	11	11	11	11	11	11	11

Summation = $11 * 10$

Result = Summation / 2

In this question you are going to sum all the number starting from 2 up to 100 with step size 3.

You should use Gauss' Method in Python to calculate the overall sum.

The function name will be **gauss** and the parameters and default values will be:

- start -> 1
- end -> 100
- step_size -> 2

Hints:

- use `range()` to create the list

Function Call:

```
conseq_sum = gauss(2, 100, 3)
```

```
conseq_sum
```

Expected Output:

```
1650.0
```

```
[  
9 1 # Q 9:  
]:  
2
```

```
3 # ---- your solution here ----
4
5
6 # call the function you defined
7 conseq_sum = gauss(2, 100, 3)
8 print(conseq_sum)
```

```
[ 9      1650.0
]:
```

Q10:

Define a function to check if a given list is sorted in descending order or not.

The function name will be **is_it_sorted_descending**.

It will return True if the list items are in descending order, False otherwise.

Function Calls:

```
a_list = [2, 3, 4, 7]
print(is_it_sorted_descending(a_list))
a_list = [7, 7, 4, 3, 2, 1, 1]
print(is_it_sorted_descending(a_list))
a_list = ['a', 'b', 'c', 'd']
print(is_it_sorted_descending(a_list))
a_list = ['d', 'c', 'b', 'a']
print(is_it_sorted_descending(a_list))
```

Expected Output:

```
False
True
False
True
```

```
[1
0]: 1      # Q 10:
2
3      # ---- your solution here ----
4
5
```

```
6  # call the function you defined
7  a_list = [2, 3, 4, 7]
8  print(is_it_sorted_descending(a_list))
9
10 a_list = [7, 7, 4, 3, 2, 1, 1]
11 print(is_it_sorted_descending(a_list))
12
13 a_list = ['a', 'b', 'c', 'd']
14 print(is_it_sorted_descending(a_list))
15
16 a_list = ['d', 'c', 'b', 'a']
17 print(is_it_sorted_descending(a_list))
```

[1
0]:

False

True

False

True

OceanofPDF.com

SOLUTIONS - List

Here are the solutions for the quiz for Chapter 14 - List.

SOLUTIONS - List:

S1:

```
[  
1  1      # S 1:  
]:  
2  
3  def sum_of_list(a_list):  
4      # variable to keep sum  
5      summation = 0  
6  
7  for i in a_list:  
8      # control for being integer  
9      if str(i).isdigit():  
10         summation += i  
11  
12 return summation  
13  
14  
15 # call the function you defined  
16 nums = [1, 2, 3, 4, 5, 'a']  
17 summation = sum_of_list(nums)  
18 print(summation)
```

```
[  
1  15  
]:
```

S2:

```
[  
1  1      # S 2:  
2
```

]:

```
2
3 def two_levels_sum(p_list):
4     summation = 0
5
6     for item in p_list:
7         # type check -> list
8         if type(item) == list:
9             # item is a list -> loop over its elements
10            for inner_item in item:
11                if str(inner_item).isdigit():
12                    summation += inner_item
13                # type check -> int
14                elif str(item).isdigit():
15                    summation += item
16
17     return summation
18
19
20 # call the function you defined
21 a_list = [[5, 8], [1, 4, 7], [10], 3, 'a']
22 summation = two_levels_sum(a_list)
23 print(summation)
```

[
2
]:

38

S3:

[
3
]:

```
1     # S 3:
2
3     def nested_sum(p_list):
4         summation = 0
```

```

5
6     for item in p_list:
7         if type(item) == int:
8             summation += item
9         elif type(item) == list:
10            # if the item is list -> we will loop over
11            # its elements
12            # recursion
13            summation += nested_sum(item)
14
15
16
17    # call the function you defined
18    a_list = [[-5, [9, 1, [6, 2]], 8], [1, 4, 7, [8, 2]],
19    [10, [-1, 2]], 3]
20    summation = nested_sum(a_list)
21    print(summation)

```

[
3
]: 57

S4:

```

[  
4  1      # S 4:  
]:  
2  
3      def squares(p_list):  
4          # new_list = []  
5          new_list = list()  
6  
7          for i, element in enumerate(p_list):  
8              new_list.append(element**2)  
9

```

```
10     return new_list
11
12
13     # call the function you defined
14     numbers = [1,2,3,4,5]
15     result = squares(numbers)
16     print(result)
```

```
[  
4      [1, 4, 9, 16, 25]  
]:
```

S5:

```
[  
5  1      # S 5:  
]:  
2
3  def sum_of_squares(p_list):  
4      new_list = []  
5
6      for element in p_list:  
7          # get total of new list  
8          new_list_total = sum_of_list(new_list)  
9
10         # get square of the current element and add  
11         it new_total  
12         new_total = new_list_total + element**2  
13
14         # append new total to new_list  
15         new_list.append(new_total)  
16
17         return new_list  
18
19     # call the function you defined
```

```
20     a_list = [1,2,3,4,5]
21     result = sum_of_squares(a_list)
22     print(result)
```

```
[  
5      [1, 5, 15, 37, 83]  
]:
```

S6:

```
[  
6     1      # S 6:  
]:  
2  
3     def odd_even_difference(p_list):  
4         # odd elements  
5         odds = p_list[1::2]  
6  
7         # odd summation  
8         odds_sum = sum_of_list(odds)  
9  
10        # even elements  
11        evens = p_list[::-2]  
12  
13        # event summation  
14        evens_sum = sum_of_list(evens)  
15  
16        # get difference  
17        diff = odds_sum - evens_sum  
18  
19        return diff  
20  
21  
22        # call the function you defined  
23        a_list = [1,2,3,4,5,6]  
24        diff = odd_even_difference(a_list)
```

```
25     print(diff)
```

```
[  
6  
]:
```

3

S7:

```
[  
7  1     # S 7:  
]:  
2  
3  def crop_the_list(p_list):  
4      # remove the first element  
5      p_list.pop(0)  
6  
7      # remove the final element  
8      p_list.pop()  
9  
10  
11     # call the function you defined  
12     nums = [1,2,3,4,5,6,7]  
13     print('the list before passing as parameter:',  
14     nums)  
15     crop_the_list(nums)  
16     print('the list before passing as parameter:',  
17     nums)
```

```
[  
7  
]:
```

the list before passing as parameter: [1, 2, 3, 4, 5, 6, 7]

the list before passing as parameter: [2, 3, 4, 5, 6]

S8:

```
[  
8  1     # S 8:  
]:
```

```

2
3 def sort_the_list(p_list, is_reverse = False):
4     sorted_list = []
5
6     # if it is reverse
7     if is_reverse:
8         sorted_list = sorted(p_list,
reverse=is_reverse)
9     else:
10        sorted_list = sorted(p_list)
11
12 return sorted_list
13
14
15 # call the function you defined
16 a = [12, 4, 2, 1, 6, 3, 45]
17 print('a:', a)
18
19 sorted_a = sort_the_list(a, False)
20 print('sorted - ascending', sorted_a)
21
22 sorted_a = sort_the_list(a, True)
23 print('sorted - descending', sorted_a)
24
25 print('a after sort_the_list function call:', a)

```

[
8
]:

```

a: [12, 4, 2, 1, 6, 3, 45]
sorted - ascending [1, 2, 3, 4, 6, 12, 45]
sorted - descending [45, 12, 6, 4, 3, 2, 1]
a after sort_the_list function call: [12, 4, 2, 1, 6,
3, 45]

```

S9:

```
[9]: 1      # S 9:  
[9]: 2  
[9]: 3      def gauss(start=1, end=100, step=2):  
[9]: 4  
[9]: 5          # create a list with range()  
[9]: 6          a_list = list(range(start, end, step))  
[9]: 7  
[9]: 8          # reverse list  
[9]: 9          r_list = a_list[::-1]  
[9]: 10  
[9]: 11          # summation  
[9]: 12          summation = 0  
[9]: 13  
[9]: 14          for i in range(len(a_list)):  
[9]: 15              pair_sum = a_list[i] + r_list[i]  
[9]: 16              summation += pair_sum  
[9]: 17  
[9]: 18          result = summation / 2  
[9]: 19  
[9]: 20          return result  
[9]: 21  
[9]: 22  
[9]: 23          # call the function you defined  
[9]: 24          conseq_sum = gauss(2, 100, 3)  
[9]: 25          print(conseq_sum)
```

```
[9]: 1650.0  
[9]:
```

S10:

```
[10]: 1      # S 10:
```

```
2
3     def is_it_sorted_descending(p_list):
4
5         # create a new list
6         sorted_list = p_list[::]
7
8         # sort the new list in descending order ->
9         sort()
10        sorted_list.sort(reverse=True)
11
12        if p_list == sorted_list:
13            return True
14        else:
15            return False
16
17
18        # call the function you defined
19        a_list = [2, 3, 4, 7]
20        print(is_it_sorted_descending(a_list))
21
22        a_list = [7, 7, 4, 3, 2, 1, 1]
23        print(is_it_sorted_descending(a_list))
24
25        a_list = ['a', 'b', 'c', 'd']
26        print(is_it_sorted_descending(a_list))
27
28        a_list = ['d', 'c', 'b', 'a']
29        print(is_it_sorted_descending(a_list))
```

[1
0]:

False
True
False
True

OceanofPDF.com

15. Dictionary

What is a Dictionary?

Dictionary (dict) is a collection of data values in **key:value** pairs. Dictionaries store multiple values like Lists. But they are different. In a List, indices are integer (int) and they are defined implicitly by Python. We don't define indices for Lists. But for a Dictionary, we define keys manually, and they don't have to be integers. They may be any suitable type.

Let's say we want to store employee data; name, age, department.

```
[  
1  employee = {  
]:  
2      'name': 'Clark Kent',  
3      "age": 37,  
4      "department": 'News Service'  
5  }
```

The **employee** object we define in cell 1 is a Dictionary. We can think of a Dictionary as a set of **key:value** pairs. In the **employee** dictionary we define, '**name**' is a key and '**Clark Kent**' is a value. So the key:value pair is: '**name': 'Clark Kent**'.

The element structure in a dictionary is: **{ key : value }**. Each key is mapped to a value. Here are the key:value pairs for our

employee dictionary.

Key	Value
name	Clark Kent
age	37
department	News Service

Table 15-1: Key-Value pairs in employee dictionary

The keys must be unique in a Dictionary. You cannot have duplicate keys.

Let's define another dictionary with name **numbers**:

```
[  
2 1 numbers = {  
]:  
2   1: "one",  
3   2: "two",  
4   3: 'three',  
5   4: 'four'  
6 }
```

In the **numbers** dictionary the keys are integers while the values are strings. Dictionaries are helpful when you need to store tabular data in a variable.

Creating a Dictionary

Dictionaries are created with curly brackets: `{}`. A pair of empty brackets creates an empty dictionary.

```
[  
3 1 # empty dict  
]:  
2 cars_empty = {}
```

The `cars_empty` variable we define in cell 3, is an empty dictionary. Let's print its type:

```
[  
4 1 type(cars_empty)  
]:  
4 dict  
]:
```

Now let's create a dictionary with elements in it. The elements are key:value pairs.

```
[  
5 1 # dict with elements  
]:  
2 cars = {  
3   'Audi': 'Germany',  
4   'Mazda': 'Japan',  
5   'Fiat': 'Italy'  
6 }
```

We define a dictionary named `car` in cell 5. It has the car brands as **keys** and their countries as **values**. Let's print it.

```
[  
6 1 print(cars)  
]:  
6 {'Audi': 'Germany', 'Mazda': 'Japan', 'Fiat': 'Italy'}
```

```
[1]:
```

dict():

Another way to create a dictionary is using the `dict()` constructor. Let's define an empty dictionary by using its constructor and print its type:

```
[7]: 1 # empty dict with constructor
[7]: 2 cars_empty_2 = dict()
```

```
[8]: 1 type(cars_empty_2)
[8]:
```

```
[8]: dict
[8]:
```

We can also pass a set of key:value pairs enclosed with a pair of braces to the `dict()` constructor.

```
[9]: 1 # dict with elements
[9]: 2 cars_2 = dict({
[9]: 3     'Audi': 'Germany',
[9]: 4     'Mazda': 'Japan',
[9]: 5     'Fiat': 'Italy'
[9]: 6 })
```

```
[10]: 1 cars_2
```

```
[10]: {'Audi': 'Germany', 'Mazda': 'Japan', 'Fiat': 'Italy'}
```

Index vs Key:

- In **Lists** we access the elements via **indices** in square brackets as: `my_list[index]`.
- In **Dictionaries** we do not use indices. We use **keys** in the square brackets as: `my_dict[key]`.

When we access an element in a dictionary as `my_dict[key]` it returns the **value** corresponding to that key. Let's access the elements in the `cars` dict by using their keys:

```
[1] 1 # dict[key]
1]: 2 cars['Audi']
```

```
[1] 1 'Germany'
1]:
```

In cell 11, on the `cars` dictionary we call the item with the key **'Audi'** as: `cars['Audi']`. And it returns **'Germany'** which is the value corresponding that key. Let's call one more time with the key as **'Fiat'**:

```
[1] 1 cars['Fiat']
2]:
```

```
[1] 1 'Italy'
2]:
```

What happens if you pass a key that does not exist in the dictionary? Let's see:

```
[1] 1 # no element with key -> Ford
[3]: 2 cars['Ford']
```

```
[1] 1
[3]: 2 KeyError: 'Ford'
```

As you see in cell 13, we call the dictionary element with a nonexistent key, ‘**Ford**’, and it returns **KeyError**, which tells us that there is no key as ‘**Ford**’.

Adding Elements to a Dictionary

We will see two different ways to add elements to a Dictionary. Let’s start with the first one, which is the value assignment with key as: **my_dict[key] = value**.

Let’s define a **numbers** dictionary to store the numbers and their text forms. The keys will be the numbers and the values will be their spelling.

```
[1] 1 numbers = {
[4]: 2     1: "one",
3     2: "two",
4     3: 'three',
5     4: 'four'
6 }
```

Let’s add number 5 to this dictionary:

```
[1] 1 # add with key
[5]: 2 numbers[5] = 'five'
```

Now, in our `numbers` dictionary, we have a new item, which is `5: 'five'`. Let's print the dictionary and see the new item.

```
[1] 1 numbers
```

```
[1] {1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five'}
```

Let's add another item. But this time, we will deliberately make a mistake in the value:

```
[1] 1 numbers[6] = 'sixty'
```

```
2 numbers
```

```
[1] {1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five', 6: 'sixty'}
```

There is a typo in the value of number `6`. It should be `'six'`. Let's fix it:

```
[1] 1 # to fix it -> we will reassign that element
```

```
2 numbers[6] = 'six'
```

```
3 numbers
```

```
[1] {1: 'one', 2: 'two', 3: 'three', 4: 'four', 5: 'five', 6: 'six'}
```

In cell 18, we reassign the item with key `6` as: `numbers[6] = 'six'`. Since the keys are unique for dictionaries, Python first checks if this key already exists. If it does, then it changes its value. If it doesn't exist then Python will create a new element with this key and value.

Let's add some elements one after another:

```
[1
9]: 1 # add some items
      2 numbers[7] = 'seven'
      3 numbers[8] = 'eight'
      4 numbers[9] = 'nine'
```

Let's print the final value of our numbers dictionary:

```
[2
0]: 1 numbers
      [2
0]: {1: 'one',
      2: 'two',
      3: 'three',
      4: 'four',
      5: 'five',
      6: 'six',
      7: 'seven',
      8: 'eight',
      9: 'nine'}
```

The second way for adding elements to the dictionary is using the `update()` method. It's syntax is: `dict.update({ key:value })`. Important point to keep in mind is, the argument you pass to it must be enclosed with a pair of braces `{}`.

Let's define a `car` dictionary (dict for short) to see the examples:

```
[2
1]: 1 # define a dict
      2 car = {
      3     "brand": 'Ford',
```

```
4     'model': "Mustang",
5     'year': 1964
6 }
7
8 print(car)
```

```
[2
1]: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964}
```

Now let's add a new element to the `car` dict. We will first define the new item as a dictionary with a single element. Then we will update the `car` dict with this new variable.

```
[2
2]: 1 # define the new element as a dict
2 element_to_add = {'color': 'red'}
3
4 # add this element to the dict
5 car.update(element_to_add)
6
7 print(car)
```

```
[2
2]: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964,
      'color': 'red'}
```

In cell 22, `element_to_add` is a new dict with just a single item. Then we add it to the `car` dict as: `car.update(element_to_add)`.

We can add the new element directly inside the parentheses of `update()` function as follows:

```
[2
3]: 1 # inside parentheses
2 car.update({ 'age': 50 })
```

In cell 23, we add the new item in form of key:value pair directly. And let's print the `car` dict again:

```
[24]: 1 car
```

```
[24]: {'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'red', 'age': 50}
```

The `update()` function can take more than one element as the argument. We get the chance to add multiple elements at once in that way. Let's see it:

```
[25]: 1 # add more than one key:value pair
      2 elements_to_add = {
      3     'price': 100000,
      4     'motor': 1.6,
      5     'age': 72
      6 }
      7
      8 car.update(elements_to_add)
```

Now let's print the `car` to see the new items in it:

```
[26]: 1 car
```

```
[26]: {'brand': 'Ford',
      'model': 'Mustang',
      'year': 1964,
      'color': 'red',
      'age': 72,
      'price': 100000,
```

```
'motor': 1.6 }
```

As a final example, let's add multiple items directly inside the parentheses of the **update()** function:

```
[2 1 # add multiple items in parentheses
7]: 2 car.update(
3   'mileage': 60000,
4   'id': 'FM-64145879',
5   'type': 'coupe'
6 )
7
8 car
```

```
[2 1 {'brand': 'Ford',
7]: 2   'model': 'Mustang',
3   'year': 1964,
4   'color': 'red',
5   'age': 72,
6   'price': 100000,
7   'motor': 1.6,
8   'mileage': 60000,
9   'id': 'FM-64145879',
10  'type': 'coupe'}
```

Deleting Elements from a Dictionary

In the previous chapter, we learned two different ways to delete items from a List which were: **pop()** and **del**. We use both for Dictionaries too. Let's see them with examples:

pop():

The `pop()` removes the element from the dictionary and returns the deleted element. When calling the `pop()` function on Dictionaries, we pass the key of the item as the argument.

```
[2  
8]: 1 # define an empty dict  
2 number_names = {}  
3  
4 # add items with keys  
5 number_names['one'] = 1  
6 number_names['two'] = 2  
7 number_names['three'] = 3  
8 number_names['four'] = 4  
9 number_names['five'] = 5  
10 number_names['six'] = 6  
11 number_names['seven'] = 7  
12 number_names['eight'] = 8  
13  
14 # print the dict  
15 number_names
```

```
[2  
8]: {'one': 1,  
      'two': 2,  
      'three': 3,  
      'four': 4,  
      'five': 5,  
      'six': 6,  
      'seven': 7,  
      'eight': 8}
```

We define the `number_names` dict in cell 28. Now let's delete the item with key '`eight`':

```
[2
9]: 1 # remove the item and assign it to a variable
      2 deleted_element = number_names.pop('eight')
      3
      4 # print the deleted item
      5 deleted_element
```

```
[2
9]: 8
```

In cell 29, we pass the key of the item we want to remove to the `pop()` method as: `number_names.pop('eight')`. And it returns the value of the deleted element.

Now let's print the `number_names` dict once more, to be sure that the item 'eight' has removed:

```
[3
0]: 1 # print the dict again
      2 number_names
```

```
[3
0]: {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6, 'seven': 7}
```

Let's delete one more element. The one with the key of 'seven' for example:

```
[3
1]: 1 # remove the item with key 'seven'
      2 number_names.pop('seven')
      3
      4 # print number_names again
      5 number_names
```

```
[3
1]: {'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6}
```

del:

The second way of deleting elements from Dictionaries is using the **del** statement. It removes the item but does not return anything.

```
[3
2]: 1 # redefine an empty dict
      2 number_names = {}
      3
      4 # add items with keys
      5 number_names['one'] = 1
      6 number_names['two'] = 2
      7 number_names['three'] = 3
      8 number_names['four'] = 4
      9 number_names['five'] = 5
     10 number_names['six'] = 6
     11 number_names['seven'] = 7
     12 number_names['eight'] = 8
     13
     14 # print the dict
     15 number_names
```

```
[3
2]: {'one': 1,
      'two': 2,
      'three': 3,
      'four': 4,
      'five': 5,
      'six': 6,
      'seven': 7,
      'eight': 8}
```

Let's delete the item with key '**one**' from the dictionary:

```
[3
3]: 1 # delete item 'one' with del
      2 del number_names['one']
      3
      4 # print the dict
      5 number_names
```

```
[3
3]: {'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6,
      'seven': 7, 'eight': 8}
```

Now let's delete the item with key 'five':

```
[3
4]: 1 # delete item 'five' with del
      2 del number_names['five']
      3
      4 # print the dict
      5 number_names
```

```
[3
4]: {'two': 2, 'three': 3, 'four': 4, 'six': 6, 'seven': 7,
      'eight': 8}
```

What happens if you pass a key which does not exist in the dictionary? Python will throw a **KeyError** if it cannot find that item in the dictionary.

```
[3
5]: 1 # pass a key which does not exist
      2 del number_names['ten']
```

```
[3
5]: KeyError: 'ten'
```

Read Elements from a Dictionary

In this section we will see how we get dictionary items in detail. Let's start by redefining our **number_names** dict:

```
[3  
6]: 1  # redefine an empty dict  
2  number_names = {}  
3  
4  # add items with keys  
5  number_names['one'] = 1  
6  number_names['two'] = 2  
7  number_names['three'] = 3  
8  number_names['four'] = 4  
9  number_names['five'] = 5  
10  number_names['six'] = 6  
11  number_names['seven'] = 7  
12  number_names['eight'] = 8  
13  
14  # print the dict  
15  number_names
```

```
[3  
6]: {'one': 1,  
      'two': 2,  
      'three': 3,  
      'four': 4,  
      'five': 5,  
      'six': 6,  
      'seven': 7,  
      'eight': 8}
```

```
[3  
7]: 1  # get the item with key 'seven'  
2  number_names['seven']
```

```
[3  
7]: 7
```

```
[3 8]: 1 # get the item with key 'two'  
2 number_names['two']
```

```
[3 8]: 2
```

In cells 37 and 38, we get the dict elements with keys ‘seven’ and ‘two’ respectively. As you see, Python returns their values.

dict.items():

Most of the time we need to get all the items in a dictionary. Python has a special method for this which is `items()`. Its syntax is: `dict.items()`. It returns a **view object** named `dict_items`. The `dict_items` is a list of key-value pairs of the dictionary. These key-value pairs are in form of **Tuples**, which we will cover in the next chapter. The view object will reflect any changes done to the dictionary.

Let’s get all the items in our `number_names` dict:

```
[3 9]: 1 # get all items in the dict  
2 number_names.items()
```

```
[3 9]: dict_items([('one', 1), ('two', 2), ('three', 3), ('four',  
4), ('five', 5), ('six', 6), ('seven', 7), ('eight', 8)])
```

dict.keys():

If we only need the keys in a dictionary, we use `keys()` method for this. The syntax is `dict.keys()` and it returns a view object, `dict_keys`, that displays a list of all the keys in the dictionary in

order of insertion. The view object will reflect any changes done to the dictionary.

```
[4
0]: 1 # get all keys
      2 number_names.keys()

[4
0]: dict_keys(['one', 'two', 'three', 'four', 'five', 'six',
      'seven', 'eight'])
```

dict.values():

If we need only the values in a dictionary we use **values()** method. Its syntax is **dict.values()** and it returns a view object, **dict_values**. The view object contains the values of the dictionary, in the form of a list. And it will reflect any changes done to the dictionary.

```
[4
1]: 1 # get the values only
      2 number_names.values()

[4
1]: dict_values([1, 2, 3, 4, 5, 6, 7, 8])
```

Now that we know how to access the items in a dictionary let's create a **for** loop to print them as key:value pairs:

```
[4
2]: 1 # with for loop, print the items
      2 for key, value in number_names.items():
      3     print(key, ':', value)

[4
2]: one : 1
      two : 2
```

```
three : 3
four : 4
five : 5
six : 6
seven : 7
eight : 8
```

In cell 42, we loop over the items of the `number_names` dict with a `for` loop. The method call of `number_names.items()` will return a list of key-value tuples as `(key, value)`. In the `for` loop definition, we deconstruct this tuple into to two variables: `key` and `value`. Then we print them in line 3.

Let's create the same `for` loop with different loop variable names, to understand it better:

```
[4
3]: 1 # 'k' is key and 'v' is for value
      2 for k, v in number_names.items():
      3     print(k, ':', v)
```

```
[4
3]: one : 1
      two : 2
      three : 3
      four : 4
      five : 5
      six : 6
      seven : 7
      eight : 8
```

Let's say, we want to print only the keys. We can loop over `keys()` to achieve this:

```
[4  
4]: 1 # only the keys
```

```
2 for k in number_names.keys():  
3     print(k)
```

```
[4  
4]:
```

```
one  
two  
three  
four  
five  
six  
seven  
eight
```

Now let's get only the values:

```
[4  
5]: 1 # only the values
```

```
2 for p in number_names.values():  
3     print(p)
```

```
[4  
5]:
```

```
1  
2  
3  
4  
5  
6  
7  
8
```

Before ending this section, let's talk a bit more on reading the elements of a dictionary. We will redefine the `car` dict to use in our next examples.

```
[4 6]: 1 car = {  
2     'brand': 'Ford',  
3     'model': 'Mustang',  
4     'year': 1964,  
5     'color': 'red',  
6     'age': 72,  
7     'price': 100000,  
8     'motor': 1.6  
9 }
```

Let's try get the elements with keys ‘**brand**’ and ‘**BRAND**’:

```
[4 7]: 1 car['brand']
```

```
[4 7]: 'Ford'
```

```
[4 8]: 1 # KeyError: 'BRAND'  
2 car['BRAND']
```

```
[4 8]: KeyError: 'BRAND'
```

Why do we get **KeyError** when we try to access the element `car['BRAND']`? Because, Python is **case-sensitive** and there is no key as ‘**BRAND**’.

get():

In cell 48, we get an error when we try to access an element which does not exist. We mainly call errors as **Exceptions** in programming. And if you do not properly handle your errors (exceptions) your program might crash. As it did in cell 48.

Now let's say, we want to get an element from the dictionary but we don't want to get any errors if it does not exist. Hopefully, we have the `get()` method for this. It's syntax is `dict.get(element)` and it returns the element if it exists. If not, it does not throw any exceptions, simply returns `None`.

```
[4  
9]: 1 # get the item 'brand'  
2 car.get('brand')
```

```
[4  
9]: 'Ford'
```

In cell 49, we try to get items with `car.get('brand')` and it returns the value as '**Ford**'.

```
[5  
0]: 1 # get the item 'BRAND'  
2 car.get('BRAND')  
3  
4 # decide the return value, if not exist  
5 car.get('BRAND', 'No such brand')
```

```
[5  
0]: No such brand'
```

In cell 50, we call the `get()` method as: `car.get('BRAND')`. And it returns nothing (`None`) because there is no such item. The good part is we don't get any errors here. That's the beauty of the `get()` method.

You can also specify what to return, if `get()` method cannot find the key. It's syntax is: `get(key, <return if not found>)`. For

example you can call it as follows: `car.get('BRAND', 'No such brand')`, as in line 5 of cell 50.

`len()`:

The built-in `len()` function gives you the length of the dictionary. **Length** means the total number of elements which are key-value pairs. Let's see the length of our `car` dict:

```
[5  
1]: 1 # length of the car dict  
2 len(car)
```

```
[5  
1]: 7
```

`in:`

We use the '`in`' operator to see if a key exist in the dictionary. It checks for the keys by default. But we can also use it for checking an item or just a value. Let's see with examples:

```
[5  
2]: 1 # in checks for keys  
2 # is there a key as 'age'  
3 'age' in car
```

```
[5  
2]: True
```

In cell 52, we check if there is any key as '`age`' in the `car` dict, and the '`in`' operator returns `True`.

```
[5  
3]: 1 # is there a key as 'height'  
2 'height' in car
```

```
[5  
3]: False
```

In cell 53, we check if there is any key as ‘**height**’ in the `car` dict, and the ‘**in**’ operator returns `False`. We can also specify the `keys()` method explicitly, although it is not needed:

```
[5  
4]: 1 # is there a key as 'height'  
     2 'height' in car.keys()
```

```
[5  
4]: False
```

Now let’s check something in the values of the dictionary:

```
[5  
5]: 1 # check for values  
     2 6 in number_names.values()
```

```
[5  
5]: True
```

```
[5  
6]: 1 # check for value 'Mustang'  
     2 value_to_check = 'Mustang'  
     3 value_to_check in car.values()
```

```
[5  
6]: True
```

```
[5  
7]: 1 # check for value 'Mondeo'  
     2 value_to_check = 'Mondeo'  
     3 value_to_check in car.values()
```

```
[5  
7]: False
```

Loop Over a Dictionary

In this section we will loop over the items of a dictionary via **items()**, **keys()** and **values()** methods. Let's redefine the cars dictionary to work with:

```
[5] 1 cars = {  
8]: 2   'Audi': 'Germany',  
3   'Mazda': 'Japan',  
4   'Fiat': 'Italy',  
5   'Ford': 'US'  
6 }  
7  
8 cars
```

```
[5] 1 {'Audi': 'Germany', 'Mazda': 'Japan', 'Fiat': 'Italy',  
8]: 2   'Ford': 'US'}
```

```
[5] 1 # print the items  
9]: 2 for car in cars.items():  
3   print(car)
```

```
[5] 1 ('Audi', 'Germany')  
9]: 2 ('Mazda', 'Japan')  
3   ('Fiat', 'Italy')  
4   ('Ford', 'US')
```

In the previous section we learned that **dict.items()** returns a list of tuples and each tuple is key-value pair. Here in cell 59, you can see it. We print each element as a tuple of **(key, value)**.

```
[6 0]: 1 # deconstructing the items as key, value
2 for brand, country in cars.items():
3     print(brand, '-', country)
```

```
[6 0]: Audi - Germany
Mazda - Japan
Fiat - Italy
Ford - US
```

In cell 60, we deconstruct the returning tuple from `cars.items()`. It returns a tuple as of `(key, value)` and we name the key as `brand` and the value as `country`.

We can easily loop over the keys as follows:

```
[6 1]: 1 # loop over the keys
2 for k in cars.keys():
3     print(k)
```

```
[6 1]: Audi
Mazda
Fiat
Ford
```

And we here is the loop over the values in the `cars` dict:

```
[6 2]: 1 # loop over the values
2 for country in cars.values():
3     print(country)
```

```
[6 2]: Germany
```

Japan
Italy
US

Example:

Define a function named `count_letters`. It will take a text as the parameter. It will count each letter in the text and print them with their counts like: `{'a': 3, 's': 3, 'e': 1}`. It should only count alphabetical letters, not any other characters.

```
[6
3]: 1  def count_letters(text):
2      # a dict for numbers and counts
3      letters = dict()
4
5      # loop over the letters in the text
6      for char in text:
7          # check if letter is alphabetical
8          if char.isalpha():
9              letters[char] = letters.get(char, 0) + 1
10
11      return letters
```

In the `count_letters` function, we first create an empty dictionary in line 3 as `letters`. Then we loop over every character in `letters`. In the `if` statement in line 8, we check if the `char` is an alphabetic string as: `char.isalpha()`. If it is alphabetic, then `isalpha()` returns `True`, and we add this `char` into the dictionary.

Let us analyze the code line 9 which is `letters[char] = letters.get(char, 0) + 1`. The `char` is the key in the dictionary, we already know this. And we also know that the keys are unique

in a dictionary. Let's assume a character appears for the first time. And let's assume it's the letter 't'. If this is the first time that the letter 't' appears then the `letters.get(char, 0)` statement will return `0`. Because we tell the `get()` method to return `0` if it cannot find the key. And we add `1` to this value, which is for the current iteration. Next time, when the letter 't' appears then the `letters.get(char, 0)` statement will return `1`, because there is already an item with this key in the dictionary. So we add `1` to it and the total value updates as `2`. That's how we calculate the values in the dictionary for each letter.

And finally, after looping over all the letters in the `text` and filling our `letters` dict, we return it to the caller. Let's call this function with a text and see the letter counts in it:

```
[6  
4]: 1 text = 'example text!'  
     2 letters = count_letters(text)  
     3 letters  
  
[6  
4]: {'e': 3, 'x': 2, 'a': 1, 'm': 1, 'p': 1, 'l': 1, 't': 2}
```

Reverse Lookup

In dictionaries we mainly look for a key and try to find the value corresponding to that key. That is the main idea of dictionary search in Python. And it's very fast because it uses a **hash table** structure. Hash tables are special data structures in which the index

values of the data element is generated from a **hash function**. That makes accessing the data much faster because the index value behaves as a key for the data value.

Now, let's say we want to do the opposite. Let's assume, we have a value this time and we want to find the key corresponding this value.

Here are the possible consequences for this reverse lookup:

- There might be more than one key mapping to the same value
- You lose the advantages of hash tables
- It may cause a high memory cost

Reverse lookup is something you should avoid whenever possible.

Now let's define a function for reverse lookup and advance our discussion of nonexistent keys.

```
[6
5]: 1 def reverse_lookup(dictionary, value):
      2     # loop over the dictionary
      3     for key in dictionary:
      4         # check if the value with this key == value
      5         if dictionary[key] == value:
      6             # we found the key
      7             return key
```

Now let's define a dictionary and call this function:

```
[6  
6]: 1 dictionary = {  
2     'a': 2,  
3     'b': 1,  
4     'c': 4,  
5     'd': 3,  
6     'e': 2  
7 }  
8  
9 dictionary
```

```
[6  
6]: {'a': 2, 'b': 1, 'c': 4, 'd': 3, 'e': 2}
```

```
[6  
7]: 1 value = 2  
2 reverse_lookup(dictionary, value)
```

```
[6  
7]: 'a'
```

We define a dictionary variable that have two keys with the same value. Keys ‘a’ and ‘e’ both have the value 2. And we call our **reverse_lookup** function with the dictionary and the value of 2. But it only returns the first key which is ‘a’. That’s one of the drawbacks of reverse lookup.

What happens if we call our function with a value that does not exist in our **dictionary**? A value of 122 for example. We know that it is not in the **dictionary** values. Let’s call our function with it:

```
[6  
8]: 1 value = 122  
2 reverse_lookup(dictionary, value)
```

In cell 68, we call the function with a value of `122`, and our function returns nothing. That's a case which we should handle. What will our function do, when it gets a value which is not included in the dictionary? Let's fix this bug.

We already know that, Python raises the `KeyError` exception when it cannot find a given key in the dictionary. We can use the same error type. And you will see how we can raise our own exceptions.

```
[6
9]: 1  def reverse_lookup(dictionary, value):
2      # loop over the dictionary
3      for key in dictionary:
4          # check if the value with this key == value
5          if dictionary[key] == value:
6              # we found the key
7              return key
8      else:
9          # raise error
10         raise KeyError('We could not find with
this value:', value)
```

In cell 69, we loop over the keys in the `dictionary` with a `for` loop. If we find the key which corresponds to the `value` parameter, we return it in line 7. If we cannot find any keys then the `for` loops succeeds without any return statement, which means the `else` block gets executed. Inside the `else` block have a special syntax to raise an exception in Python. `raise` is the keyword and it's syntax is: `raise SomeError`. We raise a `KeyError` as: `raise`

KeyError('We could not find with this value:', value). We can pass any text we want into the **KeyError()** constructor. The error description is up to us.

Now that our function is capable of handling cases when no keys exist, let's call it one more time.

```
[7] 1 value = 333
0]: 2 reverse_lookup(dictionary, value)
[7]      KeyError: ('We could not find with this value:',
0]:      333)
```

In cell 70, we call the **reverse_lookup** function with a value **333**. This value does not exist in the **dictionary** and our function raised an exception of **KeyError**: ('We could not find with this value:', 333). That is the exception which we define.

Dictionary & List

Lists can be used as **values** in Dictionaries. For example, for the letters in a text we can keep the count as the key and a list of letters having that count as the value:

```
[7] 1 letters_list = {
1]: 2 1: ['k', 'd'],
3 3: ['e', 'g', 'l'],
4 4: ['a'],
5 5: ['i', 's']
6 }
7
```

```
8 print(letters_list)
[7
1]: {1: ['k', 'd'], 2: ['e', 'g', 'l'], 3: ['a'], 4: ['i', 's']}
```

Now let's define a function that takes a text parameter and returns a dictionary which looks like the `letters_list` in cell 71. Actually it will be just the opposite of the result we got from the `count_letters` function in cell 63.

```
[7
2]: 1 def lists_of_letters(text):
2
3     # get the dictionary of occurrences
4     # by calling the count_letters function
5     dictionary_with_letter_key =
6     count_letters(text)
7
8     dictionary_with_count_key = {}
9
10    for letter, count in
11        dictionary_with_letter_key.items():
12            # if this count is not in keys -> add it and
13            # assign a list of key
14            if count not in dictionary_with_count_key:
15                dictionary_with_count_key[count] =
16                [letter]
17            else:
18                # this value is in letters keys
19                dictionary_with_count_key[count].append
20                (letter)
21
22    return dictionary_with_count_key
```

Here are the steps for how we define the `lists_of_letters` function in cell 72:

- In line 5 we call the `count_letters` function and pass the `text` parameter as the argument. We assign the result to a local variable named `dictionary_with_letter_key`. We know that, `count_letters` gives us a dictionary with letters being the keys and counts being the values. Here we need the opposite of this. The counts will be the keys and a list of letters will be the values.
- In line 7 we define an empty dictionary as `dictionary_with_count_key`. It will be our new dictionary to return.
- In line 9, we loop over the items of `dictionary_with_letter_key`. We deconstruct each item as `letter` and `count`.
- In line 11, we check if this `count` is not in the `dictionary_with_count_key` as: `if count not in dictionary_with_count_key`. Remember that the `in` statement checks for the keys by default. So we check if the `count` does not exist in the keys of `dictionary_with_count_key`.
- If the `count` does not exist in the keys of `dictionary_with_count_key`, then we add it as: `dictionary_with_count_key[count] = [letter]`. Please

be careful, that we add this **letter** in the form of a list as: **[letter]**. That's because the values of the **dictionary_with_count_key** will be of type list.

- In line 13, in the **else** statement, we know that this **count** already exists in the **dictionary_with_count_key**. Because it is the **else** statement of line 11. And since it exists as a key, we can access its value as: **dictionary_with_count_key[count]**. And since this value is a list we can append the new letter to it as: **dictionary_with_count_key[count].append(letter)**.
- In line 17, we return the **dictionary_with_count_key**.

Let's call the **lists_of_letters** function with a text:

```
[7] 1 text = 'example text!'
3]: 2 letters_numbers = lists_of_letters(text)
3 3]: 3 letters_numbers
      {3: ['e'], 2: ['x', 't'], 1: ['a', 'm', 'p', 'l']}
```

As you see in cell 73, the **lists_of_letters** function return a dictionary which has the counts as the keys and a list of letters as the values.

Very Important Note:

Lists:

- **can** be used as **values** of Dictionaries

- but **cannot** be used as **keys** of Dictionaries

Let's first see what this note means with an example:

```
[7
4]: 1 # list
      2 a = [1, 2]
      3
      4 # dict
      5 s = dict()
      6
      7 # pass list as key for dict
      8 s[a] = 'my list'
```

```
[7
4]: TypeError: unhashable type: 'list'
```

As you see in cell 74, we tried to set the list `a`, as the key for the dictionary `s`, but Python raised an exception. **TypeError**: unhashable type: 'list'. Which states that a list is an unhashable type, and you cannot use an unhashable type as dictionary keys.

Rule of Thumb: Dictionary Keys must be **hashable** types.

A **hash** is a function which takes any value and returns an integer for that value. A Dictionary uses the hash values for the keys to store and access the items. In other words, it uses the hash value like an index for the keys.

Remember we talked about Mutable and Immutable Types. Now we are ready to see the connection between all these concepts.

Mutable vs. Immutable:

- **Hash** values for **Mutable** types may **change** (because they can be mutated)
- **Hash** values for **Immutable** types **cannot change** (they cannot be mutated)

Here is the connection:

- **Mutable** Types **cannot** be keys for a Dictionary
- **Immutable** Types **can** be keys for a Dictionary

So Immutable Types can be keys for Dictionaries, but Mutable Types cannot. Since Lists are Mutable Types they cannot be used as the keys for Dictionaries.

Let's remember the built-in data types in Python once again to see which one can be a key for a Dictionary:

Type	Desc.	Immutable?	Can be Key for Dictionary?
int	Integer	Yes	Yes
float	Float	Yes	Yes
bool	Boolean	Yes	Yes
str	String	Yes	Yes
list	List	No	No
tuple	Tuple	Yes	Yes
dict	Dictionary	No	No
set	Set	No	No
frozenset	Frozen Set	Yes	Yes

As a final example let see some valid keys that we can define for a Dictionary:

```
[7  
5]: 1 # define an empty dict  
2 s = dict()  
3  
4 # bool keys are valid  
5 s[True] = 'Correct'  
6 s[False] = 'Incorrect'  
7  
8 # int and float keys are valid  
9 s[5] = 'five'  
10 s[4.8] = 'four-dot-eight'  
11  
12 # string keys are valid  
13 s['final'] = 124545454
```

```
[7  
6]: 1 # print the final value of s  
2 s
```

```
[7  
6]: {True: 'Correct',  
      False: 'Incorrect',  
      5: 'five',  
      4.8: 'four-dot-eight',  
      'final': 124545454}
```

QUIZ - Dictionary

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_Dictionary.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *15_Dictionary*. Here are the questions for this chapter:

QUIZ - Dictionary:

Q1:

Define a function named **words_dict**.

It will read *en_words.txt* file and will create a dictionary from these words.

It will only take the words that have more than or equal to 19 letters.

The Key of the dictionary will be the word itself and the Value will be number of letters in the word.

The function will return this dictionary.

Hints:

- `open()`

Expected Output:

```
{  
'anticonservatism': 19,  
'comprehensiveness': 19,  
'counterdemonstration': 20,  
'counterdemonstrations': 21,  
...  
}
```

```
[  
1 1 # Q 1:  
]:  
2  
3 # ---- your solution here ----
```

```
4
5
6 # call the function you defined
7 words_dictionary = words_dict()
8 words_dictionary
```

```
[  
1  
]:  
'anticonservationist': 19,  
'comprehensivenesses': 19,  
'counterdemonstration': 20,  
'counterdemonstrations': 21,  
'counterdemonstrator': 19,  
'counterdemonstrators': 20,  
'counterinflationary': 19,  
'counterpropagations': 19,  
'counterretaliations': 19,  
'disinterestednesses': 19,  
'electrocardiographs': 19,  
'extraconstitutional': 19,  
'hyperaggressiveness': 19,  
'hyperaggressivenesses': 21,  
'hypersensitivenesses': 20,  
'inappropriatenesses': 19,  
'inconsideratenesses': 19,  
'interdenominational': 19,  
'irreconcilabilities': 19,  
'microminiaturization': 20,  
'microminiaturizations': 21,  
'miscellaneouses': 19,  
'multidenominational': 19,  
'representativenesses': 20}
```

Q2:

Define a function named **words_length_dict**.

It will read en_words.txt file and will create a dictionary from these words.

It will only take the words that have more than or equal to 19 letters.

The Key will be the length (number of characters in word) and the Value will be a List of words which have that length. The function will return this dictionary.

Hints:

- `open()`

Expected Output:

```
{19: ['anticonservations', 'comprehensivenesses',  
'counterdemonstrator', ...],  
20: ['counterdemonstration', 'counterdemonstrators',  
'hypersensitivenesses', ...],  
21: ['counterdemonstrations', 'hyperaggressivenesses',  
'microminiaturizations']}
```

```
[  
2 1 # Q 2:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 length_words = words_length_dict()  
8 length_words
```

```
[  
2 {19: ['anticonservations',  
]:  
'comprehensivenesses',  
'counterdemonstrator',  
'counterinflationary',  
'counterpropagations',
```

```
'counterretaliations',
'disinterestednesses',
'electrocardiographs',
'extraconstitutional',
'hyperaggressiveness',
'inappropriatenesses',
'inconsideratenesses',
'interdenominational',
'irreconcilabilities',
'miscellaneous',
'multidenominational'],
20: ['counterdemonstration',
      'counterdemonstrators',
      'hypersensitivenesses',
      'microminiaturization',
      'representativenesses'],
21: ['counterdemonstrations',
      'hyperaggressivenesses',
      'microminiaturizations']}
```

Q3:

Define 4 functions named `car_1`, `car_2`, `car_3`, `car_4`.

These functions will create dictionaries as below (name of the dictionary will be `car`):

```
{'brand': 'Ford',
'model': 'Mustang',
'year': 1964,
'color': 'Red',
'price': 30000,
'km': 89000,
'motor': 1.6}
```

Functions will create the dictionary by different ways and return the dictionary.

Hints:

- { }

- `dict()`
- `update()`

Expected Output:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964, 'color': 'Red', 'price': 30000, 'km': 89000, 'motor': 1.6}
```

```
[3]: 1  # Q 3:  
2  
3  # Function 1  
4  # ---- your solution here ----  
5  
6  # call the function you defined  
7  print(car_1())  
8  
9  
10 # Function 2  
11 # ---- your solution here ----  
12  
13 # call the function you defined  
14 print(car_2())  
15  
16  
17 # Function 3  
18 # ---- your solution here ----  
19  
20 # call the function you defined  
21 print(car_3())  
22  
23  
24 # Function 4  
25 # ---- your solution here ----  
26  
27 # call the function you defined
```

28 `print(car_4())`

[3
]:

```
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964,  
'color': 'Red', 'price': 30000, 'km': 89000, 'motor':  
1.6}  
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964,  
'color': 'Red', 'price': 30000, 'km': 89000, 'motor':  
1.6}  
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964,  
'color': 'Red', 'price': 30000, 'km': 89000, 'motor':  
1.6}  
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964,  
'color': 'Red', 'price': 30000, 'km': 89000, 'motor':  
1.6}
```

Q4:

Define a function named `create_a_new_car`.

It will call one of the functions defined in Q3 and will get the car dictionary.

Then it will copy the items of this car dictionary into another dictionary via a loop.

It will first copy all the elements in car dictionary, then create new keys via appending "`_2`" at the end of existing key, and create a new element.

Values will be the same.

It will return the new dictionary.

Hint:

- `copy()`
- `update()`
- `items()`

Expected Output:

```
{  
'brand': 'Ford',  
'model': 'Mustang',  
'year': 1964,  
'color': 'Red',
```

```
'price': 30000,  
'km': 89000,  
'motor': 1.6  
'brand_2': 'Ford',  
'model_2': 'Mustang',  
'year_2': 1964,  
'color_2': 'Red',  
'price_2': 30000,  
'km_2': 89000,  
'motor_2': 1.6  
}
```

```
[  
4 1 # Q 4:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 create_a_new_car()
```

```
[  
4 {'brand': 'Ford',  
]:  
'model': 'Mustang',  
'year': 1964,  
'color': 'Red',  
'price': 30000,  
'km': 89000,  
'motor': 1.6,  
'brand_2': 'Ford',  
'model_2': 'Mustang',  
'year_2': 1964,  
'color_2': 'Red',  
'price_2': 30000,
```

```
'km_2': 89000,  
'motor_2': 1.6}
```

Q5:

Define a function named **concat_dicts**.

It will concatenate the dictionaries below and return the resulting dict.

The function will take these dictionaries as parameters.

Hints:

- use only one for loop
- search for looping on multiple dictionaries. Here is an example for statement: **for x in (d1, d2,):**
- update()

Dictionaries to concat:

```
d1={4:120, 7:60}
```

```
d2={'A': 300, 'B':400}
```

```
d3={True: 'Correct', False: 'Incorrect'}
```

Expected Output:

```
{4: 120, 7: 60, 'A': 300, 'B': 400, True: 'Correct', False: 'Incorrect'}
```

```
[  
5 1      # Q 5:  
]:  
2  
3      # ---- your solution here ----  
4  
5  
6      # call the function you defined  
7  d1={4:120, 7:60}  
8  d2={'A': 'AAA', 'B':'BBB'}  
9  d3={True: 'Correct', False: 'Incorrect'}  
10  
11  d = concat_dicts(d1, d2, d3)  
12  print(d)
```

[
5
]:

```
{4: 120, 7: 60, 'A': 'AAA', 'B': 'BBB', True:  
'Correct', False: 'Incorrect'}
```

Q6:

Define a function named **sum_of_same_keys**.

It will take two dictionaries (d1, d2) as parameters.

And it will sum the values of items having the same key in both dicts.

It will not take distinct keys. So it will return a new dictionary with only common keys.

Hints:

- Check if both parameters are dictionary (dict)
 - if not raise an error -> 'Both parameters must be dictionary type!'
 - to check -> use **isinstance()** instead of **type()**.
 - **isinstance(,)**
- Check if both dictionaries have the same length
 - if not raise an error -> 'Dictionaries must be the same length!'

Parameters:

```
d1 = {'a': 10, 'b': 30, 'c':50}
```

```
d2 = {'a': 40, 'b': 60, 'd':90}
```

Expected Output:

```
{'a': 50, 'b': 90}
```

[
6
]:

```
1 # Q 6:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 d1 = {'a': 10, 'b': 30, 'c':50, 'd': 100}
```

```
8 d2 = {'a': 40, 'b': 60, 'd':90, 'e': 50}
9 sum_of_same_keys(d1, d2)
```

```
[  
6      {'a': 50, 'b': 90, 'd': 190}  
]:
```

Q7:

Let's improve the function in Q6.

It will work the same for the same keys.

But this time we will also include the different keys.

Function name will be

sum_of_same_keys_value_of_distinct_ones.

Hints:

- Check if both parameters are dictionary (dict)
 - if not raise an error -> 'Both parameters must be dictionary type!'
 - to check -> use **isinstance()** instead of **type()**.
 - **isinstance(,)**
- Check if both dictionaries have the same length
 - if not raise an error -> 'Dictionaries must be the same length!'

Parameters:

d1 = {'a': 10, 'b': 30, 'c':50}

d2 = {'a': 40, 'b': 60, 'd':90}

Expected Output:

{'a': 50, 'b': 90, 'c': 50, 'd': 90}

```
[  
7  1 # Q 7:  
]:  
2  
3 # ---- your solution here ----  
4  
5
```

```
6 # call the function you defined
7 d1 = {'a': 10, 'b': 30, 'c': 50}
8 d2 = {'a': 40, 'b': 60, 'd': 90}
9 sum_of_same_keys_value_of_distinct_ones(d1, d2)
```

```
[  
7      {'a': 50, 'b': 90, 'c': 50, 'd': 90}  
]:
```

Q8:

Define a function named **delete_odds**.

It will take a dictionary as parameter.

It will delete the items with odd indices from the dictionary and return a new dictionary with remaining items.

Hints:

- Do not change original dictionary (parameter)
- items() for the loop
- enumerate() for the index

Parameter Dictionary:

```
dictionary = {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D', 'e': 'E', 'f': 'F'}
```

Expected Output:

```
{'a': 'A', 'c': 'C', 'e': 'E'}
```

```
[  
8 1 # Q 8:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 dictionary = {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D', 'e': 'E',  
8 'f': 'F'}  
9 evens = delete_odds(dictionary)  
9 print(evens)
```

```
[  
8      {'a': 'A', 'c': 'C', 'e': 'E'}  
]:
```

Q9:

Define a function named **convert_lists_into_dict**.

It will take two lists as parameters.

The function will use the first list elements as Keys and second list elements as Values and it will create a dictionary. Then it will return this dictionary.

Hints:

- `enumerate()`

Parameters:

```
l_1 = ['name', 'lastname', 'age', 'gender']  
l_2 = ['John', 'Doe', 100, 'Male']
```

Expected Output:

```
{'name': 'John', 'lastname': 'Doe', 'age': 100, 'gender': 'Male'}
```

```
[  
9      1      # Q 9:  
]:  
2  
3      # ---- your solution here ----  
4  
5  
6      # call the function you defined  
7      l_1 = ['name', 'lastname', 'age', 'gender']  
8      l_2 = ['John', 'Doe', 100, 'Male']  
9      employee = convert_lists_into_dict(l_1, l_2)  
10     print(employee)
```

```
[  
9      {'name': 'John', 'lastname': 'Doe', 'age': 100,  
]:      'gender': 'Male'}
```

Q10:

Let's consider a function with keys being both numbers and letters.

Example: {'a': 'A', 'b': 'B', 2: 200, 'd': 'D', 5: 300, 'f': 'F', 1: 50}

Define a function named **alphabetical**.

It will delete the elements with keys being number.

And it will return the final dictionary which has only alphabetical keys.

Hints:

- Mutate the original dictionary that is the parameter
- use two loops
- keys() for loops
- pop() for delete
- to check if alphabetical -> isalpha()
- keep in mind isalpha() is a string (str) function

Parameter Dictionary:

dictionary = {'a': 'A', 'b': 'B', 2: 200, 'd': 'D', 5: 300, 'f': 'F', 1: 50}

Expected Output:

dictionary before calling alphabetical: {'a': 'A', 'b': 'B', 2: 200, 'd': 'D', 5: 300, 'f': 'F', 1: 50}

dictionary after calling alphabetical: {'a': 'A', 'b': 'B', 'd': 'D', 'f': 'F'}

```
[1] 1      # Q 10:  
0]: 2  
     3      # ---- your solution here ----  
     4  
     5  
     6      # call the function you defined  
     7      dictionary = {'a': 'A', 'b': 'B', 2: 200, 'd': 'D', 5:  
     8      300, 'f': 'F', 1: 50}  
           print("dictionary before calling alphabetical:",  
                  dictionary)
```

```
9     alphabetical(dictionary)
10    print("dictionary after calling alphabetical:",
      dictionary)

[1
0]:   dictionary before calling alphabetical: {'a': 'A',
      'b': 'B', 2: 200, 'd': 'D', 5: 300, 'f': 'F', 1: 50}
      dictionary after calling alphabetical: {'a': 'A', 'b':
      'B', 'd': 'D', 'f': 'F'}
```

OceanofPDF.com

SOLUTIONS - Dictionary

Here are the solutions for the quiz for Chapter 15 - Dictionary.

SOLUTIONS - Dictionary:

S1:

```
[  
1 1      # S 1:  
]:  
2  
3      # Let's define a CONSTANT  
4      # and we will use it as a global variable  
5      MIN_CHAR_LENGTH = 19  
6  
7  
8  def words_dict():  
9  
10     # an empty dict  
11     dictionary = dict()  
12  
13     # read the words  
14     file = open('en_words.txt')  
15  
16     # iterate over file -> line by line  
17     for i, line in enumerate(file):  
18  
19         # line -> \n, ''  
20         line_list = line.split()  
21  
22         word = line_list[0]  
23  
24         # if i < 20:  
25             # print(word)
```

```
26
27      # check for length
28      if len(word) >= MIN_CHAR_LENGTH:
29          # print(word)
30
31      # check if it has not been added already
32      if not word in dictionary:
33          # add the word into dict
34          dictionary[word] = len(word)
35
36  return dictionary
37
38
39  # call the function you defined
40 words_dictionary = words_dict()
41 words_dictionary
```

[
1
]:

```
{'anticonservatism': 19,
'comprehensivenesses': 19,
'counterdemonstration': 20,
'counterdemonstrations': 21,
'counterdemonstrator': 19,
'counterdemonstrators': 20,
'counterinflationary': 19,
'counterpropagations': 19,
'counterretaliations': 19,
'disinterestednesses': 19,
'electrocardiographs': 19,
'extraconstitutional': 19,
'hyperaggressiveness': 19,
'hyperaggressivenesses': 21,
'hypersensitivenesses': 20,
'inappropriatenesses': 19,
```

```
'inconsideratenesses': 19,
'interdenominational': 19,
'irreconcilabilities': 19,
'microminiaturization': 20,
'microminiaturizations': 21,
'miscellaneous': 19,
'multidenominational': 19,
'representativenesses': 20}
```

S2:

```
[ 1 # S 2:
]: 2
 3 # global constant
 4 MIN_CHAR_LENGTH = 19
 5
 6 def words_length_dict():
 7
 8     # an empty dict
 9     dictionary = dict()
10
11     # read the words
12     file = open('en_words.txt')
13
14     # iterate over file -> line by line
15     for i, line in enumerate(file):
16
17         # line -> \n, '
18         line_list = line.split()
19
20         word = line_list[0]
21
22         # get the length
23         length = len(word)
```

```
24
25      # check for length
26      if len(word) >= MIN_CHAR_LENGTH:
27
28          # first time -> [word]
29          if not length in dictionary:
30              dictionary[length] = [word]
31          else:
32              dictionary[length].append(word)
33
34      return dictionary
35
36
37      # call the function you defined
38      length_words = words_length_dict()
39      length_words
```

[
2
]:

```
{19: ['anticonservatism',
```

```
'comprehensivenesses',
'counterdemonstrator',
'counterinflationary',
'counterpropagations',
'counterretaliations',
'disinterestednesses',
'electrocardiographs',
'extraconstitutional',
'hyperaggressiveness',
'inappropriatenesses',
'inconsideratenesses',
'interdenominational',
'irreconcilabilities',
'miscellaneous',
'multidenominational'],
```

```
20: ['counterdemonstration',
      'counterdemonstrators',
      'hypersensitivenesses',
      'microminiaturization',
      'representativenesses'],
21: ['counterdemonstrations',
      'hyperaggressivenesses',
      'microminiaturizations']}
```

S3:

```
[3
]: 1 # S 3:
 2
 3 # Function 1
 4 def car_1():
 5     car = {}
 6     car['brand'] = 'Ford'
 7     car['model'] = 'Mustang'
 8     car['year'] = 1964
 9     car['color'] = 'Red'
10     car['price'] = 30000
11     car['km'] = 89000
12     car['motor'] = 1.6
13     return car
14
15 # call the function you defined
16 print(car_1())
17
18
19 # Function 2
20 def car_2():
21     car = dict()
22     car['brand'] = 'Ford'
23     car['model'] = 'Mustang'
```

```
24     car['year'] = 1964
25     car['color'] = 'Red'
26     car['price'] = 30000
27     car['km'] = 89000
28     car['motor'] = 1.6
29     return car
30
31 # call the function you defined
32 print(car_2())
33
34
35 # Function 3
36 def car_3():
37     car = dict()
38     car.update(
39         {
40             'brand': 'Ford',
41             'model': 'Mustang',
42             'year': 1964,
43             'color': 'Red',
44             'price': 30000,
45             'km': 89000,
46             'motor': 1.6
47         }
48     )
49     return car
50
51 # call the function you defined
52 print(car_3())
53
54
55 # Function 4
56 def car_4():
57     car = dict(
```

```

58     {
59         'brand': 'Ford',
60         'model': 'Mustang',
61         'year': 1964,
62         'color': 'Red',
63         'price': 30000,
64         'km': 89000,
65         'motor': 1.6
66     }
67 )
68     return car
69
70 # call the function you defined
71 print(car_4())

```

[3]:

```

{'brand': 'Ford', 'model': 'Mustang', 'year': 1964,
'color': 'Red', 'price': 30000, 'km': 89000, 'motor':
1.6}
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964,
'color': 'Red', 'price': 30000, 'km': 89000, 'motor':
1.6}
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964,
'color': 'Red', 'price': 30000, 'km': 89000, 'motor':
1.6}
{'brand': 'Ford', 'model': 'Mustang', 'year': 1964,
'color': 'Red', 'price': 30000, 'km': 89000, 'motor':
1.6}

```

S4:

[4]:

```

1     # S 4:
2
3     def create_a_new_car():
4
5         # first get the car dictionary

```

```
6     car = car_4()
7
8     # add this car dict into a new dict
9     new_car = car.copy()
10
11    # loop over items
12    for item in car.items():
13
14        # print(item)
15
16        key = item[0]
17        value = item[1]
18
19        key += '_2'
20
21        new_car[key] = value
22
23    return new_car
24
25
26    # call the function you defined
27    create_a_new_car()
```

```
[4]: {'brand': 'Ford',
      'model': 'Mustang',
      'year': 1964,
      'color': 'Red',
      'price': 30000,
      'km': 89000,
      'motor': 1.6,
      'brand_2': 'Ford',
      'model_2': 'Mustang',
      'year_2': 1964,
```

```
'color_2': 'Red',
'price_2': 30000,
'km_2': 89000,
'motor_2': 1.6}
```

S5:

```
[ 1      # S 5:
5  ]:
2
3  def concat_dicts(d1, d2, d3):
4
5      dictionary = {}
6
7      # Multiple Dictionaries -> (d1, d2, ...)
8
9      for e in (d1, d2, d3):
10          dictionary.update(e)
11
12  return dictionary
13
14
15  # call the function you defined
16  d1={4:120, 7:60}
17  d2={'A': 'AAA', 'B':'BBB'}
18  d3={True: 'Correct', False: 'Incorrect'}
19
20  d = concat_dicts(d1, d2, d3)
21  print(d)
```

```
[ 5      {4: 120, 7: 60, 'A': 'AAA', 'B': 'BBB', True:
5  ]:      'Correct', False: 'Incorrect'}
```

S6:

```
[  
6 1     # S 6:  
]:  
2  
3     def sum_of_same_keys(d1, d2):  
4  
5         # check 1 -> dict types  
6         if not isinstance(d1, dict) or not isinstance(d2,  
dict):  
7             raise Exception('Both parameters must be  
dictionary type!')  
8  
9         # check 2 -> length  
10        if len(d1) != len(d2):  
11            raise Exception('Dictionaries must be the  
same length!')  
12  
13        # passed  
14        dictionary = {}  
15  
16        for key in d1:  
17            if key in d2:  
18                dictionary[key] = d1[key] + d2[key]  
19  
20        return dictionary  
21  
22  
23        # call the function you defined  
24        d1 = {'a': 10, 'b': 30, 'c': 50, 'd': 100}  
25        d2 = {'a': 40, 'b': 60, 'd': 90, 'e': 50}  
26        sum_of_same_keys(d1, d2)
```

```
[  
6 6     {'a': 50, 'b': 90, 'd': 190}  
]:
```

S7:

```
[  
7 1    # S 7:  
]:  
2  
def  
3 sum_of_same_keys_value_of_distinct_ones(d1,  
d2):  
4  
    # check 1 -> dict types  
    if not isinstance(d1, dict) or not isinstance(d2,  
dict):  
        raise Exception('Both parameters must be  
dictionary type!')  
8  
    # check 2 -> length  
    if len(d1) != len(d2):  
        raise Exception('Dictionaries must be the  
same length!')  
12  
13    # passed  
14    dictionary = {}  
15  
16    # loop over d1  
17    for key in d1:  
18        if key in d2:  
19            dictionary[key] = d1[key] + d2[key]  
20        else:  
21            dictionary[key] = d1[key]  
22  
23    # loop over d2  
24    for key in d2:  
25        if not key in d1:  
26            dictionary[key] = d2[key]
```

```
27
28     return dictionary
29
30
31     # call the function you defined
32     d1 = {'a': 10, 'b': 30, 'c': 50}
33     d2 = {'a': 40, 'b': 60, 'd': 90}
34     sum_of_same_keys_value_of_distinct_ones(d1,
d2)
```

```
[  
7  
]:      {'a': 50, 'b': 90, 'c': 50, 'd': 90}
```

S8:

```
[  
8  1      # S 8:  
]:  
2  
3     def delete_odds(dictionary):  
4  
5         d = {}  
6  
7         for index, item in  
enumerate(dictionary.items()):  
8             key = item[0]  
9             value = item[1]  
10  
11             if index % 2 == 0:  
12                 d[key] = value  
13  
14     return d  
15  
16  
17     # call the function you defined
```

```
18     dictionary = {'a': 'A', 'b': 'B', 'c': 'C', 'd': 'D', 'e':  
19         'E', 'f': 'F'}  
20     evens = delete_odds(dictionary)  
21     print(evens)
```

```
[  
8      {'a': 'A', 'c': 'C', 'e': 'E'}  
]:
```

S9:

```
[  
9      1      # S 9:  
]:  
2  
3      def convert_lists_into_dict(list1, list2):  
4  
5          dictionary = dict()  
6  
7          for index, key in enumerate(list1):  
8              dictionary[key] = list2[index]  
9  
10         return dictionary  
11  
12  
13         # call the function you defined  
14         l_1 = ['name', 'lastname', 'age', 'gender']  
15         l_2 = ['John', 'Doe', 100, 'Male']  
16         employee = convert_lists_into_dict(l_1, l_2)  
17         print(employee)
```

```
[  
9      {'name': 'John', 'lastname': 'Doe', 'age': 100,  
]:      'gender': 'Male'}
```

S10:

```

[1
0]: 1 # S 10:
2
3 def alphabetical(dictionary):
4
5     # list to keep keys to delete
6     keys_to_delete = []
7
8     # get the keys to delete
9     for key in dictionary.keys():
10         if not str(key).isalpha():
11             keys_to_delete.append(key)
12
13     # loop over keys to delete -> delete that key
14     # from the dict
15     for key in keys_to_delete:
16         if key in dictionary.keys():
17             dictionary.pop(key)
18
19     # call the function you defined
20     dictionary = {'a': 'A', 'b': 'B', 2: 200, 'd': 'D', 5:
21     300, 'f': 'F', 1: 50}
22     print("dictionary before calling alphabetical:", dictionary)
23     alphabetical(dictionary)
24     print("dictionary after calling alphabetical:", dictionary)

```

```

[1
0]: dictionary before calling alphabetical: {'a': 'A',
'b': 'B', 2: 200, 'd': 'D', 5: 300, 'f': 'F', 1: 50}
      dictionary after calling alphabetical: {'a': 'A', 'b':
'B', 'd': 'D', 'f': 'F'}

```

16. Tuple

Tuple Creation

Tuple is a Sequence Type like String and List. The elements of a Tuple are separated by commas. Tuples can include any type of elements. And they are indexed (implicitly) by integers like Lists. The main difference between Tuples and List is; Lists are Mutable Types while Tuples are Immutable.

Let's see how we create a Tuple:

```
[  
1  1  t = 'x', 'y', 'z', 'q', 'p'  
]:  
2  print(t)
```

```
[  
1      ('x', 'y', 'z', 'q', 'p')  
]:
```

In cell 1, we create a Tuple with name `t`. As you see, we pass a sequence of items separated with commas to create a Tuple.

```
[  
2  1  type(t)  
]:
```

```
[  
2      tuple  
]:
```

The type name in Python is **tuple** with lowercase letters. Let's define another Tuple. This time the items will be enclosed with a pair of parentheses.

```
[  
3 1 t2 = (1, 2, 4, 6, 8, 20)  
]:  
2 t2
```

```
[  
3 (1, 2, 4, 6, 8, 20)  
]:
```

It is not mandatory to use parentheses in Tuple definition but by convention people use it. We will be using parentheses most of time in this book.

Let's check the type of tuple **t2**. But let's use the **isinstance()** method this time. **isinstance(var, type)** returns **True** if the first parameter **var** is of type the second parameter.

```
[  
4 1 # check if t2 is a tuple  
]:  
2 isinstance(t2, tuple)
```

```
[  
4 True  
]:
```

Question: How to create a tuple with only one element?

```
[  
5 1 one_tuple = 'x'  
]:  
2 type(one_tuple)
```

```
[  
5      str  
]:
```

In cell 5, we just assign a single string value ‘x’ to the **one_tuple** variable. But when we print its type we see that it’s a string (**str**), not a **tuple**.

```
[  
6  1 one_tuple = ('x')  
]:  
2 type(one_tuple)  
  
[  
6      str  
]:
```

In cell 6, we try with parentheses around the value ‘x’. But it’s still a string. So, how do we create a tuple with just one element?

Answer: You add a comma after the element.

```
[  
7  1 one_tuple = ('x',)  
]:  
2 print(one_tuple)  
3 type(one_tuple)  
  
[  
7      ('x',)  
]:  
tuple
```

In cell 7, we define a tuple with just one element as: ('x',). We need this extra comma to convert it to a tuple. If we don’t use the comma, then it’s a string.

```
[  
8 1 one_tuple_2 = 'y',  
]:  
2 print(one_tuple_2)  
3 type(one_tuple_2)
```

```
[  
8 ('y',)  
]:  
tuple
```

And in cell 8, we define the one element tuple as: 'y',. Since the parentheses are not mandatory we omit them. But we have to use the extra comma.

tuple():

Another way to create a tuple is the **tuple()** constructor. Let's create some tuples with the constructor.

```
[  
9 1 # empty tuple  
]:  
2 t = tuple()  
3 t
```

```
[  
9 ()  
]:
```

In cell 9 we define an empty tuple with the **tuple()** constructor. As you in the cell output, an empty tuple is nothing but a pair of empty parentheses. Let's check its type to be sure if it's a tuple:

```
[1 0]: 1 # check if t is a tuple
2 isinstance(t, tuple)

[1 0]: True

[1 1]: 1 # let's create a tuple with single element
2 single = tuple('x')
3 single

[1 1]: ('x',)
```

In cell 11, we create a tuple with just one element. We just pass the element into the `tuple()` constructor as the argument: `tuple('x')`.

Let's pass a string into the `tuple()` constructor and see what happens:

```
[1 2]: 1 lang = tuple('Python')
2 lang

[1 2]: ('P', 'y', 't', 'h', 'o', 'n')
```

In cell 12, we pass the string `'Python'` into the `tuple()` constructor. And the result is a tuple of characters in that string.

Indexing and Slicing

Indexing:

Since Tuples are of Sequence Types, you can access its elements with indices inside the square brackets as: `tuple[index]`. In cell 13 and 14 you can see the examples:

```
[1 3]: 1 # get the first element of lang tuple
2 lang[0]
```

```
[1 3]: 'P'
```

```
[1 4]: 1 # get the element at index 2
2 lang[2]
```

```
[1 4]: 't'
```

What happens if you enter a list as an argument into the `tuple()` constructor?

```
[1 5]: 1 # define a list
2 my_list = ['A', 'B', 'C', 'D']
3
4 # pass my_list into tuple()
5 list_tuple = tuple(my_list)
6
7 # print the list_tuple
8 list_tuple
```

```
[1 5]: ('A', 'B', 'C', 'D')
```

In cell 15, we pass a list into the `tuple()` constructor as: `tuple(my_list)`. And the result is a tuple with the elements of that

list.

Slicing:

If there is indexing in a type then there is also slicing. In Tuples, slicing works exactly the same as in Lists.

```
[1] 1 # define a text
6]: 2 text = 'Moon is the orbit of Earth.'
3
4 # convert the text into a tuple
5 tup = tuple(text)
6
7 # print the tuple
8 print(tup)
```



```
[1] ('M', 'o', 'o', 'n', ' ', 'i', 's', ' ', 't', 'h', 'e', ' ', 'o', 'r',
6]:   'b', 'i', 't', ' ', 'o', 'f', ' ', 'E', 'a', 'r', 't', 'h', '.'
```

In cell 16, we create a tuple out of a string. And you see the tuple in the output. Now, let's do some slicing on this tuple:

```
[1] 1 # slicing
7]: 2 # slice the word -> Moon
3 tup[0:4]
```



```
[1] ('M', 'o', 'o', 'n')
```

In cell 17, we slice the tuple from index 0 to index 4. Remember the end index, 4, is not included. So the result is a tuple which is: ('M', 'o', 'o', 'n').

```
[1 8]: 1 # negative index -> start -1, right to left
2 # Earth.
3 # [-6 -5 -4 -3 -2] -1
4 tup[-6:-1]
```

```
[1 8]: ('E', 'a', 'r', 't', 'h')
```

In cell 18, we use negative index for slicing. We start from index -6 and end at index -1. And the result is the tuple of: ('E', 'a', 'r', 't', 'h').

We can use slicing for copying the whole object as we did in Lists. Let's copy the entire tuple with slicing:

```
[1 9]: 1 # copy the whole tuple
2 new_tup = tup[::]
3 print(new_tup)
```

```
[1 9]: ('M', 'o', 'o', 'n', ' ', 'i', 's', ' ', 't', 'h', 'e', ' ', 'o', 'r',
'b', 'i', 't', ' ', 'o', 'f', ' ', 'E', 'a', 'r', 't', 'h', ' ')
```

We can also copy the tuple in the opposite order. Remember we use the negative step size for reverse copy. Let's do it:

```
[2 0]: 1 # take the inverse of tuple
2 inv_tup = tup[::-1]
3 print(inv_tup)
```

```
[2 0]: ('.', 'h', 't', 'r', 'a', 'E', ' ', 'f', 'o', ' ', 't', 'i', 'b', 'r', 'o',
' ', 'e', 'h', 't', ' ', 's', 'i', ' ', 'n', 'o', 'o', 'M')
```

Tuples are Immutable. Let's try to mutate a tuple and see what happens:

```
[2 1]: 1 t = tuple([0,1,2,34,5])  
2 t
```

```
[2 1]: (0, 1, 2, 34, 5)
```

```
[2 2]: 1 # see the item at index 3  
2 t[3]
```

```
[2 2]: 34
```

The item at index 3 is 34. Let's try to change it to a value of 3:

```
[2 3]: 1 # change the item at index 3  
2 t[3] = 3
```

```
[2 3]: TypeError: 'tuple' object does not support item  
assignment
```

As you see in cell 23, we cannot mutate the tuple `t`. Python throws `TypeError` when we try to change an item in it. As we saw earlier, for Immutable Types, if we want to change any of its parts, we have to reassign the variable. Let's reassign tuple `t` with a new value at index 3:

```
[2 4]: 1 # we will reassign it  
2 t = tuple([0,1,2,3,4,5])  
3 t
```

```
[2 4]: (0, 1, 2, 3, 4, 5)
```

Tuple Comparison

Comparing Tuples is the same as comparing Strings. Python starts from the first elements of both Tuples. If the first elements are not equal then this means either one of them is less than the other one. Python stops comparing, because the decision is made. It will not move forward. But if the first elements are equal, then it will check the second elements. And it will move on like this.

If the elements are strings, then it will be an alphabetical sort. But if the elements are numbers then it will be numeric comparison. Let's see some examples:

```
[2 5]: 1 # String Comparison
2 'abxc' < 'ated'
```

```
[2 5]: True
```

In cell 25, we compare two strings. Since the first letters are ‘a’ in both strings, Python will move on to the second elements. In the string on the left the second element is ‘b’ while it is ‘t’ in the string on the right. So string on the left is less than the one on the right. That's why the result of comparison is **True**.

Let's do the same comparison with tuples this time:

```
[2 6]: 1 # Tuple Comparison
2 ('a', 'b', 'x', 'c') < ('a', 't', 'e', 'd')
```

```
[2 6]: True
```

Now they are tuples, not strings. But the comparison is exactly the same. The first elements are equal, but the second element on the left is less than the second one on the right. So the first tuple is less than the second one.

```
[2 7]: 1 # String Comparison
2 'abc' < 'aac'
```

```
[2 7]: False
```

```
[2 8]: 1 # Tuple Comparison
2 tuple('abc') < tuple('aac')
```

```
[2 8]: False
```

In cell 27, we compare strings and we can see that `'abc'` is less than `'aac'`. So the output is `False`. That's exactly the same for tuples in cell 28. The expression of `tuple('abc') < tuple('aac')` returns `False`.

```
[2 9]: 1 # Tuple Comparison
2 (1, 2, 3) < (5, 4, 3)
```

```
[2 9]: True
```

9]:

In cell 29, since the first element of the left hand side is **1** and the first one on the right is **5**, Python decides the tuple on the left is less than the one on the right. It doesn't look other items.

[3
0]:

```
1 # Tuple Comparison  
2 (5, 40, 200) < (6, 1, 3)
```

[3
0]:

```
True
```

In cell 30, the first element on the left is **5**. The first one on the right is **6**. So the tuple on the left is less than the one on the right.

[3
1]:

```
1 # String Comparison  
2 '1243' < '1234'
```

[3
1]:

```
False
```

[3
2]:

```
1 # Tuple Comparison  
2 tuple('1243') < tuple('1234')
```

[3
2]:

```
False
```

In cells 31 and 32 we compare the same set of numbers in forms of strings and tuples respectively. The result is the same in both cases. All the elements are the same in both sides. So it is **False** to say that left hand side is less than the right hand side.

Tuple Assignment

In Python, **Tuple assignment** is a very powerful feature that allows a tuple of variables on the left to be assigned values from a tuple on the right of the assignment. We will do some examples on tuple assignments and you will see how useful it is.

Example: Let's say we have two variables, **a** and **b**, and we want to swap their values. Swapping means we will assign the value of **a** to **b** and the value of **b** to **a**.

```
[3
3]: 1 # define two variables
      2 a = 99
      3 b = 1
```

To swap the values of **a** and **b**, let's first use the traditional way, which is using a temporary (temp) variable.

```
[3
4]: 1 # assign the value of a to b
      2 # and the value of b to a
      3 # a will be 1
      4 # b will be 99
      5
      6 # Way 1 -> temporary variable
      7
      8 print("before assignment:")
      9 print('a: ', a)
     10 print('b: ', b)
     11
     12 # assign a to temp
     13 temp = a
```

```
14
15 # assign b to a
16 a = b
17
18 # assign temp to b
19 b = temp
20
21 print("after assignment")
22 print('a: ', a)
23 print('b: ', b)
```

[3
4]: before assignment:

```
a: 99
b: 1
after assignment
a: 1
b: 99
```

In cell 34, we first assign the value of `a` to a `temp` variable in line 13. Then in line 16, we assign `b` to `a`. And finally in line 19, we assign the `temp` variable to `b`. And we are able to swap their values.

The solution to swap the values in cell 34, is a bit cumbersome. Actually there is a much more elegant way in Python for this operation. Which is the tuple assignment.

[3
5]:

```
1 # Tuple Assignment
2
3 # initialize the variables
4 a = 99
5 b = 1
```

```
6
7     print("before assignment:")
8     print('a:', a)
9     print('b:', b)
10
11    # tuple assignment
12    a, b = b, a
13
14    print("after tuple assignment")
15    print('a:', a)
16    print('b:', b)
```

[3
5]: before assignment:

```
a: 99
b: 1
after tuple assignment
a: 1
b: 99
```

In cell 35, line 12, we implement tuple assignment. Its syntax is: **a, b = b, a**. Python assigns the value of **a** to **b** and the value of **b** to **a** simultaneously.

Example: Let's do another example. We want to create two variables, **num_1** and **num_2** with values **500** and **800** respectively. And we want to do this via tuple assignment.

[3
6]:

```
1 # tuple assignment to create two variables
2 num_1, num_2 = 500, 800
3
4 # print them
5 print('num_1: ', num_1)
```

```
6 print('num_2: ', num_2)  
[3 6]: num_1: 500  
       num_2: 800
```

In cell 36, we create two variables with tuple assignment. You can think tuple assignment as **unpacking** of a tuple into variables.

Important:

In Tuple Assignment, number of variables must be the same on both sides of the assignment.

```
[3 7]: 1 # number of items are different in two sides  
       2 num_1, num_2 = 500, 800, 600  
[3 7]: ValueError: too many values to unpack (expected 2)
```

As you see in cell 37, the number of items in both sides of assignment are not equal. On the left we have two variables while on the left we have three. Python cannot complete this assignment. That's why we get **ValueError**.

Example: Split the user name and domain name from an email via Tuple Assignment.

```
[3 8]: 1 # split the string at the car '@'  
       2 'johndoe@example.com'.split('@')  
[3 8]: ['johndoe', 'example.com']
```

In cell 38, we split the email of '`johndoe@example.com`' via the `split()` method. And we already know that `split()` returns a list of the parts. We want to assign these two parts to variables.

```
[3  
9]: 1 # split the string and do tuple assignment
```

```
2 user_name, domain =  
'johndoe@example.com'.split('@')
```

```
[4  
0]: 1 print(user_name)
```

```
2 print(domain)
```

```
[4  
0]: johndoe
```

```
example.com
```

In cell 39, we use tuple assignment to assign the items of the list on the right to two separate variables on the left. And in cell 40, we print their values.

For tuple assignment (or unpacking) the right hand side does not have to be a tuple. It can be a list too. Let's see an example of unpacking on the lists.

Example: Assign each element of a list to separate variables via Tuple Assignment.

```
[4  
1]: 1 # create a list
```

```
2 my_list = ['A', 'B', 'C', 'D']  
3 my_list
```

```
[4  
1]: ['A', 'B', 'C', 'D']
```

```
[4 2]: 1 # unpack list elements into separate variables
2 a, b, c, d = my_list
```

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

```
[4 3]: a: A
b: B
c: C
d: D
```

In cell 42, we unpack the list elements and assign them to separate variables: **a**, **b**, **c** and **d**. And we print their values in cell 43.

Tuples and Functions

You may notice that, functions can return only one value.

Question: What should we do if we want the function to return multiple values?

Answer: Return a Tuple that includes all the values you want to return.

Example:

The built-in **divmod()** function takes two parameters:

- dividend

- divisor

And returns a tuple of two results:

- quotient
- remainder

```
[4
4]: 1 # divide 23 by 4
      2 result = divmod(23, 4)
      3 result
```

```
[4
4]: (5, 3)
```

In cell 44, we use the `divmod()` function to divide 23 by 4. And the function returns the `result` which is a tuple of two values the quotient and the remainder as: `(5, 3)`. We can also create two different variables out of this result:

```
[4
5]: 1 # create separate variables -> quotient, remainder
      2 quotient = result[0]
      3 print('quotient:', quotient)
      4
      5 remainder = result[1]
      6 print('remainder:', remainder)
```

```
[4
5]: quotient: 5
      remainder: 3
```

In cell 45, we get the items of result tuple via indices as: `result[0]` and `result[1]`. And we assign these items to two

variables as **quotient** and **remainder**.

We can unpack the result of **divmod(23, 4)** in just one line with tuple assignment. Let's do it:

```
[4
6]: 1 # tuple assignment
      2 # unpacking
      3 quotient, remainder = divmod(23, 4)
      4
      5 print('quotient:', quotient)
      6 print('remainder:', remainder)
```

```
[4
6]: quotient: 5
      remainder: 3
```

In cell 46, line 3, we assign the items of the returning tuple of **divmod(23, 4)** to two separate variables with tuple assignment.

Example:

Define a function with unknown parameters (***args**). The function name will be **sum_and_multiply** and it will return the summation and multiplication result of these arguments.

Before writing the actual code for this function, let's do a quick debugging. We will print the ***args** parameter and see what is in.

```
[4
7]: 1 def sum_and_multiply(*args):
      2     print(args)
      3     print(type(args))
```

```
[4
8]: 1 sum_and_multiply(2, 5, 4, 3)
```

```
[4  
8]: (2, 5, 4, 3)  
<class 'tuple'>
```

In cell 47 we print the `*args` parameter and its type. And in cell 48 we call the function. The result is a tuple. This shows us that, Python keeps the unknown parameters (`*args`) in the form of a tuple.

Now that we know `*args` is a tuple, we can loop over its items to do multiplication operation. Let's define the function:

```
[4  
9]: 1 # let's define  
2 def sum_and_multiply(*args):  
3  
4     # summation  
5     summation = sum(args)  
6  
7     # multiplication -> for loop  
8     multiplication = 1  
9     for arg in args:  
10         multiplication *= arg  
11  
12     # return both results  
13     return (summation, multiplication)
```

In the function body we use the built-in `sum()` function for summation. And for multiplication we set a `for` loop. It loops over the items of the `*args` tuple. We call the current item as `arg` at each iteration. In line 10, we use shorthand multiplication to

multiple all the items. And finally in line 13, we return a tuple of summation and multiplication as: **(summation, multiplication)**.

```
[5 0]: 1 sum_and_multiply(2, 5, 4, 3)
```

```
[5 0]: (14, 120)
```

In cell 50, we call the `sum_and_multiply(2, 5, 4, 3)` function with some arguments. And it returns a tuple of summation and multiplication results of these numbers.

We can use tuple assignment to assign the summation and multiplication results to separate variables in a single line as follows:

```
[5 1]: 1 # get the result elements from the tuple
2 # via Tuple Assignment
3 summation, multiplication = sum_and_multiply(2, 5,
4, 3)
4
5 print('summation:', summation)
6 print('multiplication:', multiplication)
```

```
[5 1]: summation: 14
multiplication: 120
```

Example:

Define a function taking a list as the parameter. The function name will be `simple_stats`. And it will return three simple statistics as:

- min
- max
- mean (average)

Since we have three variables to return, we need to **pack** them in a tuple as: (**minimum**, **maximum**, **mean**). We will use the **statistics** module to be able to calculate the mean.

```
[5
2]: 1  # import the modules
      2  import statistics
      3
      4  def simple_stats(a_list):
      5
      6      # minimum
      7      minimum = min(a_list)
      8
      9      # maximum
     10     maximum = max(a_list)
     11
     12      # mean
     13     mean = statistics.mean(a_list)
     14
     15      # return -> tuple(min, max, average)
     16     return (minimum, maximum, mean)
```

In cell 52, we define our **simple_stats** function. We first import the **statistics** module as: **import statistics**. In line 7, we use the built-in **min()** function to calculate the minimum value. In line we use the built-in **max()** function for the maximum value. In line 13, we call the **statistics.mean(a_list)** function and pass the

`a_list` as the argument. It returns the average (mean) of the numbers in the list. And finally in line 16, we return a tuple of these three variables as: `return (minimum, maximum, mean)`.

Now let's call the function and unpack the returning tuple:

```
[5
3]: 1 # define a list
      2 my_list = [1, 2, 3, 4, 5]
      3
      4 # call the function and unpack the result
      5 minimum, maximum, average =
      6 simple_stats(my_list)
      7
      8 print('minimum:', minimum)
      9 print('maximum:', maximum)
      9 print('average:', average)
```

```
[5
3]: minimum: 1
      maximum: 5
      average: 3
```

zip() Function

In Python, sometimes we need to use different types together. For example, we need to iterate over the items in a List and a Tuple in the same loop. We will learn the built-in `zip()` function to see how we manage this. The `zip()` function takes two or more sequence type arguments.

Sequence Type: A type that you can access its elements via indices. The indices are integer and implicit. Here the built-in sequence types in Python:

- List
- Tuple
- String
- Range

The `zip()` function takes the corresponding elements (with the same indices) of the parameters and returns them as a Zip Object which contains Tuples.

```
[5
4]: 1 # define a string and a list
      2 text = 'xyzt'
      3 a_list = [1, 2, 3, 4]
      4
      5 # zip the string and the list
      6 zip_obj = zip(text, a_list)
      7
      8 print(zip_obj)
```

```
[5
4]: <zip object at 0x0000025BB897B880>
```

In cell 54, line 6, we see the syntax of the `zip()` function. We pass two arguments, a string and a list as: `zip(text, a_list)`. And it returns a zip object. This zip object contains tuples which have item pairs from the string and the list respectively.

Let's loop over the zip object and see its elements:

```
[5]: 1 # loop over zip_obj
      2 for z in zip_obj:
      3     print(z)
```

```
[5]: ('x', 1)
      ('y', 2)
      ('z', 3)
      ('t', 4)
```

As you see in the output of cell 55, each element in the zip object is a Tuple:

- the first element is from the String
- the second element is from the List

Let's try to access the elements of the zip object via indices and see what happens:

```
[5]: 1 # try to get the item at index 0
      2 zip_obj[0]
```

```
[5]: 6: TypeError: 'zip' object is not subscriptable
```

As you see in cell 56, we get **TypeError** when we try to access an element of the zip object with an index. Why? Because the zip object is an **iterator**. An iterator is a special object in Python with **__next__** dunder method in it. You cannot directly access the elements of an iterator. The details of iterators are out of

the scope of this book. For now, it's enough to know that you can iterate over an iterator and you can convert it to a list.

```
[5  
7]: 1 # define two lists  
2 d1 = ['A', 'B', 'C', 'D', 'E']  
3 d2 = [10, 20, 30, 40, 50]  
4  
5 # zip the lists into a zip object  
6 new_zip = zip(d1, d2)  
7  
8 # convert the zip object (iterator) to a list  
9 zip_list = list(new_zip)  
10  
11 # print the list  
12 zip_list
```

```
[5  
7]: [('A', 10), ('B', 20), ('C', 30), ('D', 40), ('E', 50)]
```

In cell 57, we zip two lists together with the `zip(d1, d2)` expression in line 6. This returns a zip object which is an iterator and we name it as `new_zip`. Then in line 9, we convert this iterator to a list with the `list()` constructor as: `zip_list = list(new_zip)`. The `zip_list` is a list of tuples. The item pairs inside these tuples are from `d1` and `d2` respectively.

Now we can access any of the elements in the `zip_list`, because it is a list now.

```
[5  
8]: 1 zip_list[2]
```

[5
8]: ('C', 30)

Question: What happens if the parameters of the `zip()` function are in different lengths?

Answer: The minimum number will decide the final length of the zip object.

[5
9]:

```
1 # two lists with different lengths
2 d1 = ['A', 'B', 'C', 'D', 'E']
3 d2 = [10, 20, 30]
4
5 # pass the lists to the zip function
6 new_zip = zip(d1, d2)
7
8 print(new_zip)
```

[5
9]:

```
<zip object at 0x0000025BB8978080>
```

[6
0]:

```
1 # assign the zip object to a list
2 zip_list = list(new_zip)
3 zip_list
```

[6
0]:

```
[('A', 10), ('B', 20), ('C', 30)]
```

In cell 59, we define two lists with different lengths. Then in line 6, we use the `zip()` function to zip them. And we get a zip object as the result.

In cell 60, we convert the zip object to a list and we print the list. As you see in the output, the length of the `zip_list` is 3. It is

the length of `d2` which is the minimum number of the lengths of two lists.

```
[6 1]: 1 # see the length of the zip_list  
2 len(zip_list)
```

```
[6 1]: 3
```

Now, let's loop over the items of the `zip_list`. We know that it's a list of tuples, so let's print these tuples:

```
[6 2]: 1 # loop over the zip_list  
2 for el in zip_list:  
3     print(el)
```

```
[6 2]: ('A', 10)  
      ('B', 20)  
      ('C', 30)
```

In cell 62, we print the items of the `zip_list`. Each item is a tuple of two elements. The first element is from `d1` and the second one is from `d2`.

Now let's unpack the elements of the tuples in a `for` loop. We will get the first and the second elements of current tuple at each iteration:

```
[6 3]: 1 # tuple unpacking  
2 for d1_el, d2_el in zip_list:  
3     print("{0} is from d1 - {1} is from  
d2".format(d1_el, d2_el))
```

[6
3]:

A is from d1 - 10 is from d2

B is from d1 - 20 is from d2

C is from d1 - 30 is from d2

In cell 63, we unpack the current tuple of the loop as: **for d1_el, d2_el in zip_list**. Then we use these variables in the **string.format()** function as: "**{0} is from d1 - {1} is from d2**".**.format(d1_el, d2_el)**.

Example:

Check the corresponding elements (at the same index) of two lists. And print them if they are equal. In other words, we want to print the common elements at the same index in both lists.

[6
4]:

```
1 # define two lists
2 l1 = ['a', 'B', 'C', 'd', 'e', 'F']
3 l2 = ['A', 'B', 'c', 'd', 'E', 'F']
4
5 # loop over the zip of them
6 for e1, e2 in zip(l1, l2):
7     # check if the items are the same
8     if e1 == e2:
9         print(e1)
```

[6
4]:

B

d

F

In cell 64, we loop over the zip of the two lists, **l1** and **l2**. We unpack the tuple items in the **for** loop as **e1** and **e2**. Then in line

8, we check if their values are the same. And we print one of them if they have the same value.

Tuple & Dictionary

We learned that, we use `dict.items()` method to get the elements of a Dictionary. And the `items()` method returns a list of Tuples.

Let's define a dictionary of letters and their ASCII codes:

```
[6
5]: 1 # ASCII codes of first 3 letters
      2 asciis = {
      3     'A': ord('A'),
      4     'B': ord('B'),
      5     'C': ord('C'),
      6     'a': ord('a'),
      7     'b': ord('b'),
      8     'c': ord('c')
      9 }
     10
     11 asciis
```

```
[6
5]: A 65
      B 66
      C 67
      a 97
      b 98
      c 99
```

Let's print the items in the `asciis` dictionary:

```
[6] 1 # items()
[6] 2 asciis.items()
[6] 3 dict_items([('A', 65), ('B', 66), ('C', 67), ('a', 97),
[6] ('b', 98), ('c', 99)])
```

Now let's loop over the items one by one, by deconstructing (unpacking) each tuple as **key** and **value**:

```
[6] 1 # loop over items
[7] 2 for key, value in asciis.items():
[7] 3     print(key, value)
```



```
[6] A 65
[7] B 66
[7] C 67
[7] a 97
[7] b 98
[7] c 99
```

Remainder:

Dictionaries are **not ordered** types. We cannot expect its items to be ordered. Because, Dictionaries do not use implicit integer indexing. We will get an error if we try to access the elements of the dictionary with an index.

```
[6] 1 # try to access the dict element with index
[8] 2 asciis[0]
```



```
[6] KeyError: 0
```

Convert a List of Tuples to a Dictionary:

We saw that `items()` method returns the elements of a Dictionary as a List of Tuples. Now let's do the opposite. This time, we have a list of tuples and we want convert it to a dictionary. Let's start by defining a list of tuples first:

```
[6 9]: 1 # define a list of tuples
2 months_days = [
3     ('January', 31),
4     ('February', 28),
5     ('March', 31),
6     ('April', 30)
7 ]
8
9 months_days
```

```
[6 9]: [('January', 31), ('February', 28), ('March', 31),
         ('April', 30)]
```

The variable `months_days` is a list of tuples. Now let's convert it into a dictionary:

```
[7 0]: 1 # convert to a dictionary => dict()
2 months = dict(months_days)
3 months
```

```
[7 0]: {'January': 31, 'February': 28, 'March': 31, 'April':
         30}
```

As you see in cell 70, when we convert a list of tuples to a dictionary, the first element becomes the key and the second one becomes the value as: `'January': 31`.

Example:

Use `zip()` and `range()` functions to create a Dictionary of days in a week. The dictionary elements will be as follows:

- 1: Monday
- 2: Tuesday
- 3: Wednesday
- 4: Thursday
- 5: Friday
- 6: Saturday
- 7: Sunday

```
[7] 1 # define a list for the day names
 2 day_names = ['Monday', 'Tuesday', 'Wednesday',
 3               'Thursday', 'Friday', 'Saturday', 'Sunday']
 4
 5 # define a range for the numbers
 6 numbers = range(1, 8)
 7
 8 # zip numbers and day_names
 9 days = zip(numbers, day_names)
10
11 # print the zip object
12 print(days)
```

```
[7] 1: <zip object at 0x000001E215A28080>
```

In cell 71, we define a list variable as `day_names`. Then we define a range of integers from 1 to 7 as: `numbers = range(1,`

8). In line 8, we use the `zip()` function to create item pairs of **numbers** and **day_names**.

We know that the result of the `zip()` function is a zip object which is an iterator. This iterator contains a list of tuples. Now let's convert it to a dictionary:

```
[7 2]: 1 # convert the zip object
        2 days_dict = dict(days)
        3
        4 print(days_dict)
```

```
[7 2]: {1: 'Monday', 2: 'Tuesday', 3: 'Wednesday', 4:
        'Thursday', 5: 'Friday', 6: 'Saturday', 7: 'Sunday'}
```

We could create a dictionary out of the `zip()` function in just one line as follows:

```
[7 3]: 1 # do the same thing in one line
        2 days_dict_2 = dict(zip(numbers, day_names))
        3
        4 print(days_dict_2)
```

```
[7 3]: {1: 'Monday', 2: 'Tuesday', 3: 'Wednesday', 4:
        'Thursday', 5: 'Friday', 6: 'Saturday', 7: 'Sunday'}
```

Tuple as Key of a Dictionary:

We have learned that the key of a Dictionary must be an Immutable type. Since Tuples are Immutable, we can use them as Dictionary keys.

Example:

We want to store the students' names and grades in a dictionary. A tuple of (**FirstName**, **LastName**) will be the key and **Grade** will be the value. Here is the dictionary we want to achieve:

```
[7
4]: 1 student_grades = {
      2     ('Musa', 'Arda'): 'AA',
      3     ('Bruce', 'Wayne'): 'DC',
      4     ('Clark', 'Kent'): 'FF',
      5     ('Peter', 'Parker'): 'FD'
      6 }
```

```
[75
]: 1 # create 3 lists
  2 names = ['Musa', 'Bruce', 'Clark', 'Peter']
  3 lastnames = ['Arda', 'Wayne', 'Kent', 'Parker']
  4 grades = ['AA', 'DC', 'FF', 'FD']
  5
  6 # create an empty dict
  7 student_grades = {}
  8
  9 # loop over 3 list simultaneously -> zip()
10 for name, lastname, grade in zip(names, lastnames,
11     grades):
12     student_grades[(name, lastname)] = grade
13
14 # print the dictionary
15 student_grades
```

[75]
]:

```
{('Musa', 'Arda'): 'AA',  
 ('Bruce', 'Wayne'): 'DC',  
 ('Clark', 'Kent'): 'FF',  
 ('Peter', 'Parker'): 'FD'}
```

In cell 75, we define three lists: **names**, **lastnames** and **grades**. And we define an empty dict as **student_grades** in line 7. In the **for** loop we pass these lists as the arguments to the **zip()** function as: **zip(names, lastnames, grades)**. And we unpack each item as: **name, lastname, grade**. Then in line 11, create a tuple of **(name, lastname)** and pass it as the dictionary key. And the value is the **grade**. This is the code for it: **student_grades[(name, lastname)] = grade**.

lambda Function as Key

Some built-in Python functions expect a declaration to execute. We have to tell them how they should process. This declaration is a function itself. In other words, we pass a function as the argument to another function. Remember, we saw the **lambda** function before. Now we will use **lambda** function as a parameter.

For example:

For sort operations Python has two built-in functions: **sort()** and **sorted()**. Here are the syntax for them:

sort()

```
list.sort(reverse=True|False, key=myFunc)
```

```
sorted()
```

```
sorted(list, key=myFunc, reverse=True|False)
```

Both of `sort()` and `sorted()` functions have a parameter called `key`. Python uses this parameter to decide how to do the sorting operation. It is the sort condition, and it expects a function. The `myFunc` that we pass to the `key` parameter is our function. First we will create a normal function and pass it to the `key` parameter. Then we will do the same thing with a `lambda` function.

To start with, let's define a function to return the length of the argument it takes. It's a very simple, one line function:

```
[7]  
6]: 1 # define a function to return the length  
     2 def myFunc(e):  
         3     return len(e)
```

The `myFunc` function takes an argument as `e`, and returns its length as: `return len(e)`. Now let's define a tuple named `cars` and call this function with the elements in this tuple. It will return the length of each element.

```
[7]  
7]: 1 # define a tuple  
     2 cars = ('Mercedes', 'Audi', 'BMW', 'Porsche', 'VW')  
  
[7]  
8]: 1 # call the function with an element  
     2 myFunc('Audi')
```

[7
8]:

4

In cell 78, we call `myFunc` with a string value of ‘Audi’ and it returns its length which 4. Let’s call with the item at index 4 of `cars` tuple:

[7
9]:

```
1 # call the function with the element
2 # at index 4 of cars tuple
3 myFunc(cars[4])
```

[7
9]:

2

The item at index 4 is ‘VW’ and `myFunc` returns 2, which is its length. Length means, number of characters for strings.

Now let’s say we want to sort the elements of `cars` tuple. But the sort condition (key) will be the length. We want to sort the `cars` tuple with respect to the length (number of chars) of the elements. It’s going to be in ascending order. Remember that, the `sorted()` function does not mutate the original tuple, it returns a sorted copy of the tuple.

[8
0]:

```
1 # sorted()
2 # the key is myFunc function
3 sorted(cars, key = myFunc)
```

[8
0]:

```
['VW', 'BMW', 'Audi', 'Porsche', 'Mercedes']
```

In cell 80, we pass `myFunc` as the argument value to the `key` parameter. This tells Python that it should use `myFunc` as the sort criteria. And we know that `myFunc` returns the length of its argument. Python will pass each elements of the `cars` tuple, to the `myFunc` one by one. And it will get the lengths of the elements. Then it will sort the tuple by these lengths in ascending order. And you see the result in the output of cell 80.

Let's try to do the same thing with the `sort()` method:

```
[8
1]: 1 # sort()
2 # sort() is a list function -> operates on lists
3 cars.sort(key = myFunc)
```

```
[8
1]: AttributeError: 'tuple' object has no attribute 'sort'
```

In cell 81, we get an `AttributeError` when we try to call the `sort()` method on a tuple. That's because `sort()` is a list method. To be able to use it, we need to convert our tuple to a list first:

```
[8
2]: 1 # sort()
2 # sort() is a list function -> operates on lists
3
4 # convert cars tuple to a list
5 cars_list = list(cars)
6
7 # print the list before sort()
8 print("---- before sort() ----")
9 print(cars_list)
10
```

```
11  # sort() -> sorts in place -> mutates the list
12  cars_list.sort(key=myFunc)
13
14  print("---- after sort() ----")
15  print(cars_list)
```

[8
2]:

```
---- before sort() ----
['Mercedes', 'Audi', 'BMW', 'Porsche', 'VW']
---- after sort() ----
['VW', 'BMW', 'Audi', 'Porsche', 'Mercedes']
```

In cell 82, we convert the `cars` tuple to a list named `cars_list`. Then we print its items before calling the `sort()` method on it. Then in line 12, we call the `sort()` method and pass `myFunc` as the key parameter. Since the `sort()` method mutates the list in-place, we see the `cars_list` as sorted in the output. And its items are sorted by their lengths because `myFunc` returns the length. Remember that, `myFunc` is the sort criteria.

In `sorted()` function and `sort()` method, we learned how we can pass a function for the `key` parameter. Now let's pass a lambda function. We saw that lambda functions are powerful one line anonymous functions.

[8
3]:

```
1  # redefine the cars tuple
2  cars = ('Mercedes', 'Audi', 'BMW', 'Porsche', 'VW')
3  cars
```

[8
3]:

```
('Mercedes', 'Audi', 'BMW', 'Porsche', 'VW')
```

```
[8 4]: 1 # sorted()
2 sorted(cars, key = lambda e: len(e))
```

```
[8 4]: ['VW', 'BMW', 'Audi', 'Porsche', 'Mercedes']
```

In cell 83, we redefine the `cars` tuple. And in cell 84, we call the `sorted()` function. This time the `key` parameter is a `lambda` function as: `key = lambda e: len(e)`. This lambda function takes one argument as `e` and returns its length as `len(e)`.

```
[8 5]: 1 # sort()
2
3 # convert the tuple to a list
4 cars_list = list(cars)
5
6 # call the sort() method
7 cars_list.sort(key=lambda e: len(e))
8
9 print(cars_list)
```

```
[8 5]: ['VW', 'BMW', 'Audi', 'Porsche', 'Mercedes']
```

In cell 85, we use the `sort()` method on the `cars_list`. And the key parameter is the same lambda function as in cell 84: `key = lambda e: len(e)`.

Question:

So far, we didn't pass any `key` parameters into `sorted()` or `sort()`. How did they do the sorting?

Answer:

They use standard sorting:

- for numbers it uses numeric sort
- for strings it uses alphabetical sort

Example:

Sort the elements of the tuple **numbers** which is **(2, 1, 3, 7, 6, 5, 4)**.

```
[8  
6]: 1 numbers = (2, 1, 3, 7, 6, 5, 4)  
     2 numbers
```

```
[8  
6]: (2, 1, 3, 7, 6, 5, 4)
```

```
[8  
7]: 1 # default -> sorts numeric  
     2 sorted(numbers)
```

```
[8  
7]: [1, 2, 3, 4, 5, 6, 7]
```

In cell 87, we call the **sorted()** function with the default value for key parameter. Since the elements are numbers, it performs numeric sort on them. We can do the same sorting with a lambda function. This lambda function will take a parameter and it will return that parameter. That's all.

```
[8  
8]: 1 # lambda  
     2 # sorted(numbers) = sorted(numbers, key=lambda  
                                e: e)
```

```
3 sorted(numbers, key=lambda e: e)
```

```
[8 8]: [1, 2, 3, 4, 5, 6, 7]
```

The lambda function for the key parameter in cell 88, has no effect on the sort. Why? Because it just returns whatever the parameter value `e` is: `lambda e: e`. And that's exactly the same condition as the default sort.

Example:

Sort the elements of the list `letters` which is `['c', 'f', 'b', 'a', 'e', 'd']`.

```
[8 9]: 1 # define the list
2 letters = ['c', 'f', 'b', 'a', 'e', 'd']
3
4 # default sort
5 letters.sort()
6
7 print(letters)
```

```
[8 9]: ['a', 'b', 'c', 'd', 'e', 'f']
```

The elements of the `letters` list are string characters, that's why the default sort is an alphabetical sort. Let's do the same sort operation with a `lambda` function now. Remember that we have to redefine the `letters` list, because the `sort()` method sorts the list in-place, which means it has mutated. So we need to reset its items.

```
[9 0]: 1 # redefine the letters
```

```
2 letters = ['c', 'f', 'b', 'a', 'e', 'd']
3
4 # letters.sort() = letters.sort(key = lambda x: x)
5 letters.sort(key = lambda x: x)
6
7 print(letters)
```

```
[9
0]: ['a', 'b', 'c', 'd', 'e', 'f']
```

In cell 90, we pass a **lambda** function for the key parameter. This function has no effect because it just returns the item it takes as the parameter: **lambda x: x**. So it's the same thing as default sort.

OceanofPDF.com

QUIZ - Tuple

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_Tuple.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *16_Tuple*. Here are the questions for this chapter:

QUIZ - Tuple:

Q1:

Define a function named **create_tuple**.

It will create a tuple of numbers from 1 to 10.

Then it will add numbers from 11 to 20 to this tuple.

And it will return the final form of the tuple.

Hints:

- Tuples are Immutable. So how do you add elements to a Tuple?
- use for loop
- range()

Expected Output:

(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20)

```
[  
1  # Q 1:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 t = create_tuple()  
8 print(t)
```

```
[1]: (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,  
]: 17, 18, 19, 20)
```

Q2:

Define a function named **tuple_to_string**.

It will take a Tuple as parameter and convert it into a String.
Then it will return this String.

Hints:

- `join()`

Parameter:

```
t = ('M', 'a', 'c', 'h', 'i', 'n', 'e', ' ', 'L', 'e', 'a', 'r', 'n', 'i', 'n', 'g')
```

Expected Output:

Machine Learning

```
[1]: # Q 2:  
[2]: 1  
[3]: 2  
[4]: 3 # ---- your solution here ----  
[5]: 4  
[6]: 5  
[7]: 6 # call the function you defined  
[8]: 7 t = ('M', 'a', 'c', 'h', 'i', 'n', 'e', ' ', 'L', 'e', 'a', 'r', 'n',  
[9]: 'i', 'n', 'g')  
[10]: 8 t = tuple_to_string(t)  
[11]: 9 print(t)
```

```
[1]:  
[2]: Machine Learning  
[3]:
```

Q3:

Define a function named **string_to_list_to_tuple**.
It will take String as parameter.

First it will convert this String into a List, then it will convert this List into a Tuple.

Finally it will return the Tuple.

Hints:

- only use constructor methods
- list()
- tuple()

Parameter:

```
pythons_father = 'Guido van Rossum'
```

Expected Result:

```
('G', 'u', 'i', 'd', 'o', ' ', 'v', 'a', 'n', ' ', 'R', 'o', 's', 's', 'u', 'm')
```

```
[  
3 1 # Q 3:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 pythons_father = 'Guido van Rossum'  
8 father_tuple =  
9 string_to_list_to_tuple(pythons_father)  
9 print(father_tuple)
```

```
[  
3 ('G', 'u', 'i', 'd', 'o', ' ', 'v', 'a', 'n', ' ', 'R', 'o', 's', 's',  
]: 'u', 'm')
```

Q4:

Define a function named **how_many_instances**.

It will take a Tuple and an index (int) as parameter.

Function will first find the element at that index.

Then it will return the number of instances of the element in the Tuple.

Hints:

- `count()`

Parameters:

`t = (5, 2, 3, 3, 4, 2, 1, 3, 4, 5, 2, 1, 2)`

`i = 1`

`how_many_instances(t, i)`

Expected Output:

`4`

```
[  
4  1      # Q 4:  
]:  
2  
3      # ---- your solution here ----  
4  
5  
6      # call the function you defined  
7  t = (5, 2, 3, 3, 4, 2, 1, 3, 4, 5, 2, 1, 2)  
8  i = 1  
9  count = how_many_instances(t, i)  
10 print(count)
```

```
[  
4      4  
]:
```

Q5:

Slicing Operations in Tuples are exactly the same as in Lists and Strings.

Our Tuple is: `tup = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')`

Find the items in `tup`, based on slicing and indexing:

1. items from position 4 (inc.) to position 7 (inc.)
2. the first 5 items
3. items from position 6 (inc.)
4. all items
5. 2nd item from the last
6. last 4 items

7. items from position 2 to 8, with step size 2
8. all items with step size 3
9. items from position 9 to position 3 (exc.) but in reverse order
10. print the tup in reverse order
11. print the tup in reverse order with step size 2
12. all the elements excluding the last one, with negative index

Expected Output for each sub question:

1. ('d', 'e', 'f', 'g')
2. ('a', 'b', 'c', 'd', 'e')
3. ('f', 'g', 'h', 'i', 'j')
4. ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')
5. i
6. ('g', 'h', 'i', 'j')
7. ('b', 'd', 'f', 'h')
8. ('a', 'd', 'g', 'j')
9. ('i', 'h', 'g', 'f', 'e', 'd')
10. ('j', 'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a')
11. ('j', 'h', 'f', 'd', 'b')
12. ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i')

```
[5]: 1 # Q 5:
      2
      3 tup = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')
      4
      5 # 1. items from position 4 (inc) to position 7 (inc)
      6 # position 4 -> index 3 (index starts 0)
      7 # position 7 -> index 6 (since it is included the
      8 # end index will be 7)
      9 # ---- your solution here ----
     10 # 2. the first 5 items
```

```
11  # ---- your solution here ----
12
13  # 3. items from position 6 (inc)
14  # ---- your solution here ----
15
16  # 4. all items
17  # ---- your solution here ----
18
19  # 5. 2nd item from the last
20  # ---- your solution here ----
21
22  # 6. last 4 items
23  # -4 , -3 , -2, -1
24  # ---- your solution here ----
25
26  # 7. items from position 2 to 8, with step size 2
27  # ---- your solution here ----
28
29  # 8. all items with step size 3
30  # ---- your solution here ----
31
32  # 9. items from position 9 to position 3 (exc) but
33  # in reverse order
34  # reverse order -> -step size
35  # ---- your solution here ----
36
37  # 10. print the tup in reverse order
38  # ---- your solution here ----
39
40  # 11. print the tup in reverse order with step size
41  # 2
42  # ---- your solution here ----
```

```
42  # 12. all the elements excluding the last one, with
     negative index
43  # ---- your solution here ----
```

[
5
]:

```
('d', 'e', 'f', 'g')
('a', 'b', 'c', 'd', 'e')
('f', 'g', 'h', 'i', 'j')
('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')
i
('g', 'h', 'i', 'j')
('b', 'd', 'f', 'h')
('a', 'd', 'g', 'j')
('i', 'h', 'g', 'f', 'e', 'd')
('j', 'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a')
('j', 'h', 'f', 'd', 'b')
('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i')
```

Q6:

Define a function named **change_tuple_ending**.

It will take a List as parameter.

The elements of this list will be Tuples.

Each Tuple may be in different length, so the size of the elements in the list is not fixed.

The function will replace the last item in each Tuple element in the list with its square.

Hints:

- Mutate the original list (the parameter list will change in-place)
- How can you get last element in a Tuple?
- How to mutate the list, while looping over it?
- You need to find a way to mutate the list (enumerate())
- How to create a Tuple with single element?

Parameter:

```
tuple_list = [(2,5,8), (4,3), (1,7,9,6), (5,)]
```

Expected Output:

```
tuple_list before passing to function: [(2, 5, 8), (4, 3), (1, 7, 9, 6), (5,)]
```

```
tuple_list after passing to function: [(2, 5, 64), (4, 9), (1, 7, 9, 36), (25,)]
```

```
[  
6  1      # Q 6:  
]:  
2  
3      # ---- your solution here ----  
4  
5  
6      # call the function you defined  
7  tuple_list = [(2,5,8), (4,3), (1,7,9,6), (5,)]  
8  print('tuple_list before passing to function:',  
       tuple_list)  
9  
10 change_tuple_ending(tuple_list)  
11 print('tuple_list after passing to function:',  
       tuple_list)
```

```
[  
6  tuple_list before passing to function: [(2, 5, 8),  
]:  
   (4, 3), (1, 7, 9, 6), (5,)]  
   tuple_list after passing to function: [(2, 5, 64), (4,  
9), (1, 7, 9, 36), (25,)]
```

Q7:

Define a function named **replace_tuple_with_sqare**.

It will take a List as parameter.

The items in this List are Tuples.

Each Tuple may be in different length, so the size of the elements in the list is not fixed.

The function will replace each element of Tuples with its square.

And it will return the new list of squared Tuples.

Hint:

- Do not change the parameter
- How to mutate the list while looping over it?

Parameter:

```
tuple_liste = [(2,5,8), (4,3), (1,7,9,6), (5,)]
```

Expected Output:

```
tuple_list: [(2, 5, 8), (4, 3), (1, 7, 9, 6), (5,)]
```

```
new_tuple_list: [(4, 25, 64), (16, 9), (1, 49, 81, 36), (25,)]
```

```
[  
7 1 # Q 7:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 tuple_list = [(2,5,8), (4,3), (1,7,9,6), (5,)]  
8  
9 new_tuple_list =  
10 replace_tuple_with_sqaure(tuple_list)  
11 print('tuple_list:', tuple_list)  
12 print('new_tuple_list:', new_tuple_list)
```

```
[  
7 tuple_list: [(2, 5, 8), (4, 3), (1, 7, 9, 6), (5,)]  
]:  
new_tuple_list: [(4, 25, 64), (16, 9), (1, 49, 81,  
36), (25,)]
```

Q8:

Define a function named **movie_characters**.

It will take 4 lists as parameters.

4 lists are:

- List 1 -> Actor/Actress Name
- List 2 -> Movie Title
- List 3 -> Release Year
- List 4 -> Character Name

Function will take these 4 lists and create 2 Tuples out of them.

Each Tuple will have 2 elements:

- Tuple 1 -> (Actor/Actress Name, Character Name)
- Tuple 2 -> (Movie Title, Release Year)

It will use the Tuple 1 as the Key and Tuple 2 as the Value and create a Dictionary.

And it will return this dictionary.

Hints:

- `zip()`

Parameters:

```
actors = ['Marlon Brando', 'Heath Ledger', 'Natalie Portman',  
'Emma Stone']
```

```
characters = ['Don Vito Corleone', 'Joker', 'The Swan Queen',  
'Mia']
```

```
movies = ['The Godfather', 'The Dark Knight', 'Black Swan',  
'La La Land']
```

```
years = [1972, 2008, 2010, 2016]
```

Expected Output:

```
{('Marlon Brando', 'Don Vito Corleone'): ('The Godfather',  
1972),  
('Heath Ledger', 'Joker'): ('The Dark Knight', 2008),  
('Natalie Portman', 'The Swan Queen'): ('Black Swan', 2010),  
('Emma Stone', 'Mia'): ('La La Land', 2016)}
```

```
[  
8 1  # Q 8:  
]:  
2
```

```

3   # ---- your solution here ----
4
5
6   # call the function you defined
7   # lists:
8   actors = ['Marlon Brando', 'Heath Ledger',
9   'Natalie Portman', 'Emma Stone']
10  characters = ['Don Vito Corleone', 'Joker', 'The
11  Swan Queen', 'Mia']
12  movies = ['The Godfather', 'The Dark Knight',
13  'Black Swan', 'La La Land']
14  years = [1972, 2008, 2010, 2016]
15

13  # call the functions
14  movie_dictionary = movie_characters(actors,
15  characters, movies, years)
16  movie_dictionary

```

[
8
]:

```

{('Marlon Brando', 'Don Vito Corleone'): ('The
Godfather', 1972),
('Heath Ledger', 'Joker'): ('The Dark Knight',
2008),
('Natalie Portman', 'The Swan Queen'): ('Black
Swan', 2010),
('Emma Stone', 'Mia'): ('La La Land', 2016)}

```

Q9:

Define a function named **sort_tuple_of_tuples**.

It will take a Tuple as the parameter.

The elements inside this Tuple are also Tuples.

Namely, the parameter is a Tuple of Tuples.

Each Tuple element contains 2 items.

The function will sort the main Tuple and return a sorted Tuple.

Sort Rule is going to be the second element in the inner Tuples.

Hints:

- lambda

Parameter:

```
tuple_of_tuples = (('a', 12), ('e', 8), ('b', 16), ('c', 22))
```

Expected Output:

```
[('e', 8), ('a', 12), ('b', 16), ('c', 22)]
```

```
[ 1     # Q 9:  
9  ]:  
2  
3     # ---- your solution here ----  
4  
5  
6     # call the function you defined  
7     tuple_of_tuples = (('a', 12), ('e', 8), ('b', 16), ('c',  
8     22))  
9     sorted_tuple =  
10    sort_tuple_of_tuples(tuple_of_tuples)  
     sorted_tuple
```

```
[ 1     [('e', 8), ('a', 12), ('b', 16), ('c', 22)]  
9  ]:  
     ]:
```

Q10:

Define a function named **number_of_occurrences**.

It will take Tuple of integers and a number as parameters.

It will return the number of occurrences of that number in the Tuple.

It will use the function defined in Q4 (how_many_instances) to find the count. (It will not do the counting itself.)

If the number does not exist in the Tuple, it will return:

'Number not in Tuple!'.

Hints:

- function returning a function
- raise
- index()

Parameters:

tup = (5, 2, 3, 3, 4, 2, 1, 3, 4, 5, 2, 1, 2)

num = 2

Expected Output:

4

```
[10] 1 # Q 10:  
:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 tup = (5, 2, 3, 3, 4, 2, 1, 3, 4, 5, 2, 1, 2)  
8 num = 2  
9 number_of_occurrences(tup, num)
```

```
[10] 4
```

SOLUTIONS - Tuple

Here are the solutions for the quiz for Chapter 16 - Tuple.

SOLUTIONS - Tuple:

S1:

```
[  
1  1      # S 1:  
]:  
2  
3  def create_tuple():  
4      # create the tuple  
5      tup = (1,2,3,4,5,6,7,8,9,10)  
6  
7      # add items  
8      for i in range(11, 21):  
9          # create a new temp tuple via addition  
10         # (i,) -> tuple with one element  
11         new_tup = tup + (i,)  
12  
13         # reassing the new tuple  
14         tup = new_tup  
15  
16      return tup  
17  
18  
19      # call the function you defined  
20      t = create_tuple()  
21      print(t)
```

```
[  
1  (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,  
]: 17, 18, 19, 20)
```

S2:

```
[  
2 1 # S 2:  
]:  
2  
3 def tuple_to_string(tup):  
4     # join all the items of tup  
5     text = ''.join(tup)  
6  
7     return text  
8  
9  
10    # call the function you defined  
11    t = ('M', 'a', 'c', 'h', 'i', 'n', 'e', ' ', 'L', 'e', 'a', 'r',  
12    'n', 'i', 'n', 'g')  
13    t = tuple_to_string(t)  
14    print(t)
```

```
[  
2 Machine Learning  
]:
```

S3:

```
[  
3 1 # S 3:  
]:  
2  
3 def string_to_list_to_tuple(text):  
4     # first String -> List  
5     a_list = list(text)  
6  
7     # now List -> Tuple  
8     tup = tuple(a_list)  
9  
10    return tup
```

```
11
12
13 # call the function you defined
14 pythons_father = 'Guido van Rossum'
15 father_tuple =
16 string_to_list_to_tuple(pythons_father)
17 print(father_tuple)
```

```
[3]: ('G', 'u', 'i', 'd', 'o', ' ', 'v', 'a', 'n', ' ', 'R', 'o', 's',
's', 'u', 'm')
```

S4:

```
[4]: 1 # S 4:
2
3 def how_many_instances(tup, index):
4     # get the element at index
5     el = tup[index]
6
7     # count()
8     return tup.count(el)
9
10
11 # call the function you defined
12 t = (5, 2, 3, 3, 4, 2, 1, 3, 4, 5, 2, 1, 2)
13 i = 1
14 count = how_many_instances(t, i)
15 print(count)
```

```
[4]: 4
```

S5:

```
[5]: 1 # S 5:
      2
      3 tup = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')
      4
      5 # 1. items from position 4 (inc) to position 7 (inc)
      6 # position 4 -> index 3 (index starts 0)
      7 # position 7 -> index 6 (since it is included the
      8 # end index will be 7)
      9 print(tup[3:7])
     10
     11 # 2. the first 5 items
     12 print(tup[:5])
     13
     14 # 3. items from position 6 (inc)
     15 print(tup[5:])
     16
     17 # 4. all items
     18 print(tup[:])
     19
     20 # 5. 2nd item from the last
     21 print(tup[-2])
     22
     23 # 6. last 4 items
     24 # -4 , -3 , -2, -1
     25 print(tup[-4:])
     26
     27 # 7. items from position 2 to 8, with step size 2
     28 print(tup[1:8:2])
     29
     30 # 8. all items with step size 3
     31 print(tup[::-3])
```

```

32  # 9. items from position 9 to position 3 (exc) but
33  in reverse order
34  # reverse order -> -step size
35  print(tup[8:2:-1])
36
37  # 10. print the tup in reverse order
38  print(tup[::-1])
39
40  # 11. print the tup in reverse order with step size
41  2
42  print(tup[::-2])
43  # 12. all the elements excluding the last one, with
44  negative index
45  print(tup[:-1])

```

[
5
]:

```

('d', 'e', 'f', 'g')
('a', 'b', 'c', 'd', 'e')
('f', 'g', 'h', 'i', 'j')
('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')
i
('g', 'h', 'i', 'j')
('b', 'd', 'f', 'h')
('a', 'd', 'g', 'j')
('i', 'h', 'g', 'f', 'e', 'd')
('j', 'i', 'h', 'g', 'f', 'e', 'd', 'c', 'b', 'a')
('j', 'h', 'f', 'd', 'b')
('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i')

```

S6:

[
6
]:

1 # S 6:

```

2
3  def change_tuple_ending(a_list):
4
5      for index, tup in enumerate(a_list):
6          # get the tuple items up to the last one
7          items_up_to_last = tup[:-1]
8
9          # last element -> square it
10         last_item = tup[-1]**2
11
12         # concat them -> override tup
13         tup = items_up_to_last + (last_item,)
14
15         # mutate the list -> change the current
16         # element
17         a_list[index] = tup
18
19
20     # call the function you defined
21     tuple_list = [(2,5,8), (4,3), (1,7,9,6), (5,)]
22     print('tuple_list before passing to function:', tuple_list)
23
24     change_tuple_ending(tuple_list)
25     print('tuple_list after passing to function:', tuple_list)

```

[
6
]:

tuple_list before passing to function: [(2, 5, 8),
(4, 3), (1, 7, 9, 6), (5,)]

tuple_list after passing to function: [(2, 5, 64), (4,
9), (1, 7, 9, 36), (25,)]

S7:

[
7

1 # S 7:

]:

```
2
3 def replace_tuple_with_sqaure(a_list):
4     # new_list
5     new_list = a_list.copy()
6
7     for index, tup in enumerate(new_list):
8         new_tup = tuple()
9
10        # loop over the tup items
11        for t in tup:
12            new_tup += (t**2,)
13
14        # mutate the list
15        new_list[index] = new_tup
16
17    return new_list
18
19
20 # call the function you defined
21 tuple_list = [(2,5,8), (4,3), (1,7,9,6), (5,)]
22
23 new_tuple_list =
24 replace_tuple_with_sqaure(tuple_list)
25
26 print('tuple_list:', tuple_list)
27 print('new_tuple_list:', new_tuple_list)
```

[
7
]:

```
tuple_list: [(2, 5, 8), (4, 3), (1, 7, 9, 6), (5,)]
new_tuple_list: [(4, 25, 64), (16, 9), (1, 49, 81, 36), (25,)]
```

S8:

```
[8]: 1 # S 8:
      2
      3 def movie_characters(actors, characters, movies,
      4   years):
      5     # create an empty dict
      6     dictionary = {}
      7
      8     # loop over the 4 lists -> zip()
      9     for actor, character, movie, year in zip(actors,
      10   characters, movies, years):
      11       # create and add tuples into dictionary
      12       dictionary[(actor, character)] = (movie,
      13   year)
      14
      15     # call the function you defined
      16     # lists:
      17     actors = ['Marlon Brando', 'Heath Ledger',
      18   'Natalie Portman', 'Emma Stone']
      19     characters = ['Don Vito Corleone', 'Joker', 'The
      20   Swan Queen', 'Mia']
      21     movies = ['The Godfather', 'The Dark Knight',
      22   'Black Swan', 'La La Land']
      23     years = [1972, 2008, 2010, 2016]
      24
      25     # call the functions
      26     movie_dictionary = movie_characters(actors,
      27   characters, movies, years)
      28     movie_dictionary
```

```
[  
8  
]:
```

```
{('Marlon Brando', 'Don Vito Corleone'): ('The  
Godfather', 1972),  
('Heath Ledger', 'Joker'): ('The Dark Knight',  
2008),  
('Natalie Portman', 'The Swan Queen'): ('Black  
Swan', 2010),  
('Emma Stone', 'Mia'): ('La La Land', 2016)}
```

S9:

```
[  
9  
]:
```

```
1      # S 9:  
2  
3  def sort_tuple_of_tuples(tuple_of_tuples):  
4      # let's say we have a tuple x  
5      # x -> ('c', 22)  
6      # x[1] -> 22  
7      return sorted(tuple_of_tuples, key = lambda  
8          x: x[1])  
9  
10     # call the function you defined  
11     tuple_of_tuples = (('a', 12), ('e', 8), ('b', 16), ('c',  
12     22))  
13  
14     sorted_tuple =  
15     sort_tuple_of_tuples(tuple_of_tuples)  
16     sorted_tuple
```

```
[  
9  
]:
```

```
[('e', 8), ('a', 12), ('b', 16), ('c', 22)]
```

S10:

```
[1
0]: 1 # S 10:
2
3 def number_of_occurrences(tup, num):
4
5     # check if the num is in tup
6     if not num in tup:
7         raise Exception('Number not in Tuple!')
8     else:
9
10    # call the function in Q4
11    # how_many_instances(tup, index)
12
13    # tuple.index(el) -> first found index
14    index = tup.index(num)
15
16    # return the function
17    return how_many_instances(tup, index)
18
19
20    # call the function you defined
21    tup = (5, 2, 3, 3, 4, 2, 1, 3, 4, 5, 2, 1, 2)
22    num = 2
23    number_of_occurrences(tup, num)
```

```
[1
0]: 4
```

17. Set

Set Creation

So far, we covered Lists, Dictionaries and Tuples. Now we will learn another built-in data type in Python which are the **Sets**.

A Set:

- is a datatype that is used to store collections of data
- is **unordered** and **unindexed**
- has elements which are **unique**
- is created by:
 - { } (but with elements in it)
 - **set()** constructor

Create a Set with curly brackets { }:

```
[  
1  1 # Try to Create Set with {}  
]:  
2  my_set = {}  
3  my_set
```

```
[  
1  {}  
]:
```

```
[  
2  1 type(my_set)  
]:
```

```
[  
2  dict  
]:
```

In cell 1, we try to create an empty set. But we see that it's a dict not a set when we print its type in cell 2. Why? Because a pair of empty curly brackets `{ }` creates a dict not a set.

To be able to create a set with curly brackets `{ }`, we have to pass elements in it. Let's do it now:

```
[  
3  1 # empty {} -> dict  
]:  
2  
3 # for set  
4 # we have to pass items in {}  
5 my_set = {'dog', 'cat', 'horse'}  
6 my_set
```

```
[  
3  {'cat', 'dog', 'horse'}  
]:
```

In cell 3, we pass elements that are separated by commas, into the curly brackets `{ }` as: `{'dog', 'cat', 'horse'}`. And this creates a set. Let's print its type now:

```
[  
4  1 type(my_set)  
]:
```

```
[  
4  set  
]:
```

As you see in cell 4, the type of `my_set` is set now. That's how we create sets with curly brackets.

Create a Set with the `set()` constructor:

Another way of creating a set is using the `set()` constructor. To create an empty set we pass an empty pair of curly brackets into the `set()` function as: `set({})`. To create a set with elements we pass the elements enclosed by a set of curly brackets as: `set({item_1, item_2, ...})`.

```
[  
5  1 # set({}) -> empty set  
]:  
2 an_empty_set = set({})  
3 type(an_empty_set)
```

```
[  
5  set  
]:
```

```
[  
6  1 # a set with elements  
]:  
2 letters = set({'A', 'B', 'C', 'D'})  
3 letters
```

```
[  
6  {'A', 'B', 'C', 'D'}  
]:
```

Sets cannot have two items with the same value. Even if you pass duplicate items, Python will only use one of them and ignore the rest.

```
[7]: 1 # Same element in a set -> only once
]: 2 grades = ['A', 'A', 'B', 'C', 'B', 'C']
3 print(grades)
```

```
[7]: ['A', 'A', 'B', 'C', 'B', 'C']
]:
```

In cell 7, we have a list which includes duplicate items. Let's convert this list to a set and see the resulting set variable:

```
[8]: 1 # convert the List -> Set
]: 2 # the same elements -> removed
3 grades_set = set(grades)
4 grades_set
```

```
[8]: {'A', 'B', 'C'}
]:
```

In cell 8, the **grades** list have duplicate items in it. But after we convert it to a set, the duplicates items are ignored. Every element occurs only once in the set.

```
[9]: 1 # String -> Set
]: 2 sentence = 'sentence'
3 set(sentence)
```

```
[9]: {'c', 'e', 'n', 's', 't'}
]:
```

In cell 9, we convert a string to a set. And in the output, you can see a set with the unique items.

Since set object does not have index structure, you cannot access its elements with indices. Let's try and see what happens:

```
[1 0]: 1 # try to access the elements of a Set with index
2 grades_set[0]
```

```
[1 0]: TypeError: 'set' object is not subscriptable
```

In cell 10, we try to access the elements in the `grades_set` with an index. But Python throws a **TypeError** stating that **'set' object is not subscriptable**. Which means you cannot use indices with sets.

Set Methods

In Python, we have a set of built-in methods that you can use on Sets. Let's see some of them.

`intersection()`:

Returns a set which contains the items that exist in both set **A**, and set **B**. It's syntax is: **A.intersection(B)**.

```
[1 1]: 1 grades = ['A', 'A', 'B', 'C', 'B', 'C']
2 grades = set(grades)
3 grades
```

```
[1 1]: {'A', 'B', 'C'}
```

```
[1] 1 letters = ['A', 'L', 'T', 'B', 'F']
[2]: 2 letters = set(letters)
      3 letters
```

```
[1] 1 {'A', 'B', 'F', 'L', 'T'}
[2]:
```

```
[1] 1 # A.intersection(B)
[3]: 2 grades.intersection(letters)
```

```
[1] 1 {'A', 'B'}
[3]:
```

In cell 13, we get the common items which exist in both sets, **grades** and **letters**. The syntax is: **grades.intersection(letters)**. And the intersection is the set of **{'A', 'B'}**.

union():

Returns a set containing the union of sets, **A** and **B**. It's syntax is: **A.union(B)**.

```
[1] 1 # get the union of grades and letters
[4]: 2 grades.union(letters)
```

```
[1] 1 {'A', 'B', 'C', 'F', 'L', 'T'}
[4]:
```

In cell 14, we get the union of the sets **grades** and **letters**. And the result is the set of **{'A', 'B', 'C', 'F', 'L', 'T'}**.

difference():

Returns a set containing the difference between two sets, **A** and **B**. It's syntax is: **A.difference(B)**.

```
[1  
5]: 1 # get the difference of grades from letters  
2 grades.difference(letters)
```

```
[1  
5]: {'C'}
```

```
[1  
6]: 1 # get the difference of letters from grades  
2 letters.difference(grades)
```

```
[1  
6]: {'F', 'L', 'T'}
```

In cell 15, we get the difference of **grades** from **letters** as: **grades.difference(letters)**. And the result is the set of **{'C'}**.

In cell 16, we get the difference of **letters** from **grades** as: **letters.difference(grades)**. And the result is the set of **{'F', 'L', 'T'}**.

issubset():

Checks if a set (**A**) is the subset of the other one (**B**). It's syntax is: **A.issubset(B)**. If all elements of **A** are in **B**, then we say '**A is a subset of B**'.

```
[1  
7]: 1 # if grades is a subset of letters  
2 grades.issubset(letters)
```

```
[1  
7]: False
```

In cell 17, we check if `grades` is a subset of `letters` as: `grades.issubset(letters)`. And it returns `False`. Because the letter ‘C’ exists in `grades` but it does not exist in `letters`. So `grades` is not a subset of `letters`.

```
[1] 1 # define a new set
[8]: 2 grades_small = {'A', 'B'}
```



```
[1] 1 # if grades_small is a subset of letters
[9]: 2 grades_small.issubset(letters)
```



```
[1] 9: True
```

In cell 16, we check whether `grades_small` is a subset of `letters` or not. And actually it is a subset of `letters`. That’s why `issubset()` returns `True`. Let’s check if it is a subset of `grades` too:

```
[2] 0: 1 # if grades_small is a subset of grades
[0]: 2 grades_small.issubset(grades)
```



```
[2] 0: True
```

`issuperset()`:

Checks if a set (A) is the superset of the other one (B). Its syntax is: `A.issuperset(B)`. If all elements of B are in A then we say ‘A is a superset of B’.

```
[2] 1 # if letters is a superset of grades
```

```
[1]:  
2 letters.issuperset(grades)
```

```
[2]  
1:  
False
```

In cell 21, we check whether `letters` is a superset of `grades`. And obviously it is not, so the `issuperset()` method returns **False**.

```
[2]  
2:  
1 # if letters is a superset of grades_small  
2 letters.issuperset(grades_small)
```

```
[2]  
2:  
True
```

Since all the elements of the `grades_small` set exist in the `letters` set, the statement of `letters.issuperset(grades_small)` returns **True**.

symmetric_difference():

Returns a set with the symmetric differences of two sets, `A` and `B`. Symmetric difference means, all elements which are only in `A` plus all elements which are only in `B`. In other words, it is the union of two differences: the difference of `A` from `B` and the difference of `B` from `A`.

```
[2]  
3:  
1 # symmetric difference of letters and grades  
2 letters.symmetric_difference(grades)
```

```
[2]  
3:  
{'C', 'F', 'L', 'T'}
```

Since the difference of `grades` from `letters` is `{'C'}` and the difference of `letters` from `grades` is `{'F', 'L', 'T'}`, the union of these two differences is the set of `{'C', 'F', 'L', 'T'}`.

Set Operations

`add()`:

Adds an element to the set. It's syntax is: `set.add(element)`.

Let's add some elements to the `grades` set:

```
[2  
4]: 1 # add 'F' to grades  
      2 grades.add('F')  
      3 grades
```

```
[2  
4]: {'A', 'B', 'C', 'F'}
```

```
[2  
5]: 1 # add 'T' to grades  
      2 grades.add('T')  
      3 grades
```

```
[2  
5]: {'A', 'B', 'C', 'F', 'T'}
```

In cell 24, we add the element of `'F'` to the `grades` set. And in cell 25, we add the letter of `'T'`. Let's see what happens if we try to add the same element:

```
[2  
6]: 1 # Python will ignore the same element  
      2 grades.add('F')  
      3 grades
```

```
[2 6]: {'A', 'B', 'C', 'F', 'T'}
```

As you see in cell 26, Python ignores the letter ‘F’ which we try to add. Because this element already exists in the set and sets cannot include duplicate elements.

remove():

Removes the specified element from the set. It’s syntax is:
set.remove(element).

```
[2 7]: 1 # set.remove(<element>)
2 grades.remove('C')
3 grades
```

```
[2 7]: {'A', 'B', 'F', 'T'}
```

In cell 27, we remove the element ‘C’ from the **grades** set. Let’s remove one more element:

```
[2 8]: 1 # remove 'F'
2 grades.remove('F')
3 grades
```

```
[2 8]: {'A', 'B', 'T'}
```

Loop over a Set:

Now let’s see how we loop over a set in Python:

```
[2 9]: 1 for letter in letters:
2     print(letter)
```

[2
9]:

```
B  
F  
L  
T  
A
```

In cell 29, we loop over the set named `letters`. We call the current element as `letter` at each iteration and print it in line 2. The loop structure is the same as we did with other types. Let's loop over the `grades` set too:

[3
0]:

```
1 for g in grades:  
2     print(g)
```

[3
0]:

```
B  
T  
A
```

Assignment in Sets:

In sets, assignment is **aliasing**. Remember, aliasing is nothing but giving another name to the same object.

[3
1]:

```
1 # create a set  
2 a = {2, 5, 7}  
3  
4 # assignment -> aliasing  
5 b = a  
6  
7 print('a:', a)  
8 print('b:', b)
```

```
[3  
1]: a: {2, 5, 7}  
     b: {2, 5, 7}
```

In cell 31, we create a set with the name `a` in line 2. And in line 5, we assign the variable `a` to `b`. Since assignment is an aliasing operation, now `a` and `b` refer to the same object in the memory. In other words they are identical. They are just two separate names for the same object.

Now let's add an element to `b` and see if `a` is also mutated:

```
[3  
2]: 1 # add element into b  
     2 b.add('NEW')  
  
[3  
3]: 1 # print both a and b to see they are identical  
     2 print('a:', a)  
     3 print('b:', b)  
  
[3  
3]: a: {2, 'NEW', 5, 7}  
     b: {2, 'NEW', 5, 7}
```

We only add the item ‘NEW’ to `b` but we see that the variable `a` has this new item too. Why? Because they are actually the same object.

Since they refer the same object;

- they are identical
- their values are equal
- a is a subset of b

- `b` is a subset of `a`
- `a` is a superset of `b`
- `b` is a superset of `a`

[3]
4]: 1 *# identity check*
2 `a is b`

[3]
4]: `True`

[3]
5]: 1 *# equality check*
2 `a == b`

[3]
5]: `True`

[3]
6]: 1 `a.issubset(b)`

[3]
6]: `True`

[3]
7]: 1 `a.issuperset(b)`

[3]
7]: `True`

Now let's add a new element to `a` and see that it has been also added to the variable `b`:

[3]
8]: 1 *# add a new element to a*
2 `a.add('YYYY')`

[3]
9]: 1 `print('a:', a)`

```
2 print('b:', b)
```

```
[3 9]: a: {2, 'NEW', 5, 'YYYY', 7}
      b: {2, 'NEW', 5, 'YYYY', 7}
```

update():

Updates the set with another set or any other type that can be converted to a set. With `update()` method, you can add either a single element or multiple elements.

```
[4 0]: 1 # define a set of fruits
      2 fruits = {'Apple', 'Orange', 'Melon'}
      3 fruits
```

```
[4 0]: {'Apple', 'Melon', 'Orange'}
```

```
[4 1]: 1 # define a list
      2 more_fruits = ['Cherry', 'Banana']
      3 more_fruits
```

```
[4 1]: ['Cherry', 'Banana']
```

In cell 40 we define a set named `fruits`. And in cell 41, we define a list named `more_fruits`. Now let's update our set with elements in this list:

```
[4 2]: 1 # update fruits with more_fruits
      2 fruits.update(more_fruits)
      3 fruits
```

[4
2]:

```
{'Apple', 'Banana', 'Cherry', 'Melon', 'Orange'}
```

In cell 42, we add the items of the **more_fruits** list to the **fruits** set with the help of **update()** function. The statement is: **fruits.update(more_fruits)**.

OceanofPDF.com

QUIZ - Set

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_Set.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *17_Set*. Here are the questions for this chapter:

QUIZ - Set:

Q1:

Define a function named **create_set_and_add**.

It will take a list as parameter.

First it will create a set containing the elements below:

"Apple", "Banana", "Cherry", "Orange"

Then it will add the items of the parameter list into this set.

And finally it will return the final set.

Hints:

- `{}`
- `add()`

Parameter:

`list_to_add = ['Cranberry', 'Grape', 'Pineapple', 'Mango']`

Expected Output:

(the order of items may change, because sets are unordered collections)

`{'Mango', 'Apple', 'Cranberry', 'Banana', 'Cherry', 'Grape', 'Pineapple', 'Orange'}`

```
[  
1  1  # Q 1:  
]:  
2  
3  # ---- your solution here ----  
4
```

```
5
6 # call the function you defined
7 list_to_add = ['Cranberry', 'Grape', 'Pineapple',
8 'Mango']
9 all_fruits = create_set_and_add(list_to_add)
10 print(all_fruits)
```

```
[1]: {'Pineapple', 'Apple', 'Mango', 'Orange', 'Grape',
'Cherry', 'Cranberry', 'Banana'}
```

Q2:

Define a function named **create_set_and_add_all_at_once**. It will take a list as parameter.

First it will create a set containing the elements below:

"Apple", "Banana", "Cherry", "Orange"

Then it will add the items of the parameter list into this set (all at once).

And finally it will return the final set.

Hints:

- `set()`
- `update()`

Parameter:

```
list_to_add = ['Cranberry', 'Grape', 'Pineapple', 'Mango']
```

Expected Output:

(the order of items may change, because sets are unordered collections)

```
{'Mango', 'Apple', 'Cranberry', 'Banana', 'Cherry', 'Grape',
'Pineapple', 'Orange'}
```

```
[2]: 1 # Q 2:
2
3 # ---- your solution here ----
4
5
```

```
6 # call the function you defined
7 list_to_add = ['Cranberry', 'Grape', 'Pineapple',
8 'Mango', 'Another Fruit']
9 all_fruits =
10 create_set_and_add_all_at_once(list_to_add)
11 print(all_fruits)
```

```
[2]   {'Pineapple', 'Apple', 'Mango', 'Another Fruit',
]:   'Orange', 'Grape', 'Cherry', 'Cranberry', 'Banana'}
```

Q3:

Define a function named **same_elements**.

It will take two Sets as parameters.

The function will return the same elements (intersection) of both sets in a List.

And this list is going to be sorted in ascending order.

Hints:

- no loops
- intersection()
- sorted()

Parameters:

```
set_1 = {10, 20, 30, 40, 50, 60}
set_2 = {20, 40, 60, 80, 90, 100}
```

Expected Output:

```
[20, 40, 60]
```

```
[1]   # Q 3:
2
3   # ---- your solution here ----
4
5
6   # call the function you defined
7   set_1 = {10, 20, 30, 40, 50, 60}
8   set_2 = {20, 40, 60, 80, 90, 100}
```

```
9     intersection = same_elements(set_1, set_2)
10    print(intersection)
```

```
[  
3      [20, 40, 60]  
]:
```

Q4:

Define a function named **all_elements**.

It will take two Sets as parameters.

The function will return all the elements (union) of both sets in a List.

And this list is going to be sorted in ascending order.

Hints:

- no loops
- union()
- sort()

Parameters:

```
set_1 = {10, 20, 30, 40, 50, 60}
```

```
set_2 = {20, 40, 60, 80, 90, 100}
```

Expected Output:

```
[100, 90, 80, 60, 50, 40, 30, 20, 10]
```

```
[  
4      1      # Q 4:  
]:  
2  
3      # ---- your solution here ----  
4  
5  
6      # call the function you defined  
7      set_1 = {10, 20, 30, 40, 50, 60}  
8      set_2 = {20, 40, 60, 80, 90, 100}  
9      union = all_elements(set_1, set_2)  
10     print(union)
```

```
[  
4  
]:
```

```
[10, 20, 30, 40, 50, 60, 80, 90, 100]
```

Q5:

Define a function named **get_difference**.

It will take two lists (list_1, list_2) as parameters.

The function will return the Set of elements which are in list_1 but not in list_2.

Hints:

- no loops
- set()
- difference()

Parameters:

```
l_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
l_2 = [2, 4, 6, 8]
```

Expected Output:

```
{1, 3, 5, 7, 9}
```

```
[  
5  
]:
```

```
1      # Q 5:  
2  
3      # ---- your solution here ----  
4  
5  
6      # call the function you defined  
7      l_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]  
8      l_2 = [2, 4, 6, 8]  
9      diff = get_difference(l_1, l_2)  
10     print(diff)
```

```
[  
5  
]:
```

```
{1, 3, 5, 7, 9}
```

Q6:

Define a function named **is_completely_different**.

It will check if two sets (parameters) are completely different or not.

Completely different means they have no elements in common.

- If they are completely different it will return -> "They are completely different."
- If they have any elements in common it will return -> "They are not completely different."

The function will also check if the two parameters are Set or not.

If any of them is not a Set it will raise an Exception as "Parameters must be of Set type."

Hints:

- no loops
- completely different: `isdisjoint()`
- `isinstance()`
- raise `Exception()`

Parameters:

```
set_1 = {20, 10, 40, 30, 50}
```

```
set_2 = {60, 80, 70, 100, 90}
```

Expected Output:

'They are completely different.'

Parameters:

```
set_1 = {20, 10, 40, 30, 50, 60}
```

```
set_2 = {60, 80, 70, 90, 40, 10}
```

Expected Output:

'They are not completely different.'

```
[  
6 1 # Q 6:  
]:
```

```

2
3 # ---- your solution here ----
4
5
6 # call the function you defined
7 set_1 = {20, 10, 40, 30, 50}
8 set_2 = {60, 80, 70, 100, 90}
9 print(is_completely_different(set_1, set_2))
10
11 set_1 = {20, 10, 40, 30, 50, 60}
12 set_2 = {60, 80, 70, 90, 40, 10}
13 print(is_completely_different(set_1, set_2))

```

[
6
]:

They are completely different.

They are not completely different.

Q7:

Define a function named **is_completely_different_2**.

It will check if two sets (parameters) are completely different or not.

Completely different means they have no elements in common.

- If they are completely different it will return -> "They are completely different."
- If they have any elements in common it will return -> "They are not completely different:"

And it will give the common elements in a set

The function will also check if the two parameters are Set or not.

If any of them is not a Set it will raise an Exception as "Parameters must be of Set type."

Hints:

- no loops
- completely different: `isdisjoint()`
- `isinstance()`
- raise `Exception()`

Parameters:

```
set_1 = {20, 10, 40, 30, 50}
set_2 = {60, 80, 70, 100, 90}
```

Expected Output:

'They are completely different.'

Parameters:

```
set_1 = {20, 10, 40, 30, 50, 60}
set_2 = {60, 80, 70, 90, 40, 10}
```

Expected Output:

'They are not completely different: {40, 10, 60}'

```
[ 1  # Q 7:
]: 2
 3  # ---- your solution here ----
 4
 5
 6  # call the function you defined
 7  set_1 = {20, 10, 40, 30, 50}
 8  set_2 = {60, 80, 70, 100, 90}
 9  print(is_completely_different_2(set_1, set_2))
10
11 set_1 = {20, 10, 40, 30, 50, 60}
12 set_2 = {60, 80, 70, 90, 40, 10}
13 print(is_completely_different_2(set_1, set_2))
```

```
[ 7  They are completely different.
]: 8
  They are not completely different: {40, 10, 60}
```

Q8:

Define a function named **copy_and_clear**.

It will take a Set as parameter.

And it will clear (remove all elements) this Set but copy its elements into another Set.

Finally it will return this new Set.

Hints:

- no loops
- copy()
- clear()
- remember Pass by Reference

Parameter:

```
set1 = {'A', 'B', 'C', 'D', 'E'}
```

Expected Output:

(the order of items may change, because sets are unordered collections)

before the function call -> set1: {'E', 'C', 'D', 'B', 'A'}

after the function call -> set1: set()

set1_copy: {'D', 'B', 'E', 'A', 'C'}

```
[  
8 1 # Q 8:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 set1 = {'A', 'B', 'C', 'D', 'E'}  
8 print('before the function call -> set1:', set1)  
9 set1_copy = copy_and_clear(set1)  
10 print('after the function call -> set1:', set1)  
11 print('set1_copy:', set1_copy)
```

[
8
]:

before the function call -> set1: {'E', 'C', 'D', 'A', 'B'}

after the function call -> set1: set()

set1_copy: {'E', 'C', 'A', 'B', 'D'}

Q9:

Define a function named **remove_common_elements**.

It will take two Sets (set_1, set_2) as parameters.

The function will remove the elements of set_1 which are also in set_2.

And this operation will change the original sets.

In other words, it will mutate the parameter.

Hints:

- no loops
- difference_update()

Parameters:

set_1 = {'a', 'b', 'c', 'd', 'e', 'f'}

set_2 = {'d', 'b', 'e', 'f', 'h', 'g'}

Expected Output:

(the order of items may change, because sets are unordered collections)

before the function -> set_1: {'c', 'a', 'b', 'f', 'd', 'e'}

after the function -> set_1: {'c', 'a'}

[
9
]:

```
1  # Q 9:  
2  
3  # ---- your solution here ----  
4  
5  
6  # call the function you defined  
7  set_1 = {'a', 'b', 'c', 'd', 'e', 'f'}  
8  set_2 = {'d', 'b', 'e', 'f', 'h', 'g'}
```

```
9  print('before the function -> set_1:', set_1)
10 remove_common_elements(set_1, set_2)
11 print('after the function -> set_1:', set_1)
```

[
9
]:

```
before the function -> set_1: {'d', 'b', 'c', 'a', 'e',  
'f'}
```

after the function -> set_1: {'c', 'a'}

Q10:

Define a function named **which_one_is_superset**.

It will take two Sets (set_1, set_2) as parameters.

If set_1 is the superset of set_2 it will return: "set_1: {....} is superset of set_2: {....}"

If the opposite is true it will return: "set_2: {....} is superset of set_1: {....}"

Here "{....}" are set elements.

Hints:

- no loop
- issuperset()

(the order of items may change, because sets are unordered collections)

Parameters:

```
set_1 = {'a', 'b', 'c', 'd'}
set_2 = {'d', 'b', 'e', 'f', 'a', 'c'}
```

Expected Output:

"set_2: {'c', 'a', 'b', 'f', 'd', 'e'} is superset of set_1: {'b', 'c', 'd', 'a'}."

Parameters:

```
set_1 = {'d', 'b', 'e', 'f', 'a', 'c'}
set_2 = {'a', 'b', 'c', 'd'}
```

Expected Output:

"set_1: {'c', 'a', 'b', 'f', 'd', 'e'} is superset of set_2: {'b', 'c', 'd', 'a'}"

```
[1 0]: 1 # Q 10:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 set_1 = {'a', 'b', 'c', 'd'}  
8 set_2 = {'d', 'b', 'e', 'f', 'a', 'c'}  
9 print(which_one_is_superset(set_1, set_2))  
10  
11 set_1 = {'d', 'b', 'e', 'f', 'a', 'c'}  
12 set_2 = {'a', 'b', 'c', 'd'}  
13 print(which_one_is_superset(set_1, set_2))
```

```
[1 0]: set_2: {'d', 'b', 'c', 'a', 'e', 'f'} is superset of set_1:  
{'b', 'd', 'a', 'c'}.  
set_1: {'d', 'b', 'c', 'a', 'e', 'f'} is superset of set_2:  
{'b', 'd', 'a', 'c'}
```

SOLUTIONS - Set

Here are the solutions for the quiz for Chapter 17 - Set.

SOLUTIONS - Set:

S1:

```
[  
1  1      # S 1:  
]:  
2  
3  def create_set_and_add(a_list):  
4      # first create the set  
5      fruits = {"Apple", "Banana", "Cherry",  
6      "Orange"}  
7      # loop over the list -> add elements  
8      for e in a_list:  
9          fruits.add(e)  
10  
11     return fruits  
12  
13  
14     # call the function you defined  
15     list_to_add = ['Cranberry', 'Grape', 'Pineapple',  
16     'Mango']  
17     all_fruits = create_set_and_add(list_to_add)  
18     print(all_fruits)
```

```
[  
1  ['Pineapple', 'Apple', 'Mango', 'Orange', 'Grape',  
]:  'Cherry', 'Cranberry', 'Banana']
```

S2:

```
[2] 1  # S 2:  
2  
3  def create_set_and_add_all_at_once(a_list):  
4      # first create the set  
5      fruits = {"Apple", "Banana", "Cherry",  
6      "Orange"}  
7      # add all elements -> update()  
8      fruits.update(a_list)  
9  
10     return fruits  
11  
12  
13     # call the function you defined  
14     list_to_add = ['Cranberry', 'Grape', 'Pineapple',  
15     'Mango', 'Another Fruit']  
16     all_fruits =  
17     create_set_and_add_all_at_once(list_to_add)  
18     print(all_fruits)
```

```
[2] 1  {'Pineapple', 'Apple', 'Mango', 'Another Fruit',  
2 2  'Orange', 'Grape', 'Cherry', 'Cranberry', 'Banana'}
```

S3:

```
[  
3 1  # S 3:  
4 2  
5 3  def same_elements(set_1, set_2):  
6 4      # get intersection -> common  
7 5      # set  
8 6      intersection = set_1.intersection(set_2)  
9 7  
10 8      # pass the set -> sorted() -> return a list
```

```
9     intersection = sorted(intersection)
10
11     return intersection
12
13
14     # call the function you defined
15     set_1 = {10, 20, 30, 40, 50, 60}
16     set_2 = {20, 40, 60, 80, 90, 100}
17     intersection = same_elements(set_1, set_2)
18     print(intersection)
```

```
[  
3      [20, 40, 60]  
]:
```

S4:

```
[  
4  1      # S 4:
]:  
2
3  def all_elements(set_1, set_2):
4      # get the union -> all elements
5      # set
6      union = set_1.union(set_2)
7
8      # sort() is a list function
9      union_list = list(union)
10
11     # pass the set -> sort() -> return a list
12     # sort() -> mutates
13     union_list.sort()
14
15     return union_list
16
17
```

```
18  # call the function you defined
19  set_1 = {10, 20, 30, 40, 50, 60}
20  set_2 = {20, 40, 60, 80, 90, 100}
21  union = all_elements(set_1, set_2)
22  print(union)
```

```
[5
4
]: [10, 20, 30, 40, 50, 60, 80, 90, 100]
```

S5:

```
[5
]: 1  # S 5:
2
3  def get_difference(list_1, list_2):
4      # first -> convert Lists into Sets
5      set_1 = set(list_1)
6      set_2 = set(list_2)
7
8      # get difference
9      diff = set_1.difference(set_2)
10
11     return diff
12
13
14  # call the function you defined
15  l_1 = [1, 2, 3, 4, 5, 6, 7, 8, 9]
16  l_2 = [2, 4, 6, 8]
17  diff = get_difference(l_1, l_2)
18  print(diff)
```

```
[5
]: {1, 3, 5, 7, 9}
```

S6:

```
[  
6 1 # S 6:  
]:  
2  
3 def is_completely_different(set_1, set_2):  
4     # check if both are Set type  
5     if not isinstance(set_1, set) or not  
6         isinstance(set_2, set):  
7             raise Exception("Parameters must be of Set  
8             type.")  
9             # they are both Set type  
10            else:  
11                # check if completely different  
12                if set_1.isdisjoint(set_2):  
13                    return 'They are completely different.'  
14                else:  
15                    return 'They are not completely  
16                    different.'  
17  
18            # call the function you defined  
19            set_1 = {20, 10, 40, 30, 50}  
20            set_2 = {60, 80, 70, 100, 90}  
21            print(is_completely_different(set_1, set_2))  
22  
23            set_1 = {20, 10, 40, 30, 50, 60}  
24            set_2 = {60, 80, 70, 90, 40, 10}  
25            print(is_completely_different(set_1, set_2))
```

```
[  
6  
]:
```

They are completely different.

They are not completely different.

S7:

```
[  
7 1 # S 7:  
]:  
2  
3 def is_completely_different_2(set_1, set_2):  
4     # check if both are Set type  
5     if not isinstance(set_1, set) or not  
6     isinstance(set_2, set):  
7         raise Exception("Parameters must be of Set  
8             type.")  
9         # they are both Set type  
10    else:  
11        # check if completely different  
12        if set_1.isdisjoint(set_2):  
13            return 'They are completely different.'  
14        else:  
15            # set  
16            common_elements =  
17            set_1.intersection(set_2)  
18  
19            return 'They are not completely different:  
20            ' + str(common_elements)  
21  
22            # call the function you defined  
23            set_1 = {20, 10, 40, 30, 50}  
24            set_2 = {60, 80, 70, 100, 90}  
25            print(is_completely_different_2(set_1, set_2))  
26            set_1 = {20, 10, 40, 30, 50, 60}  
27            set_2 = {60, 80, 70, 90, 40, 10}  
28            print(is_completely_different_2(set_1, set_2))
```

[
7
]:

They are completely different.

They are not completely different: {40, 10, 60}

S8:

[
8
]:

```
1      # S 8:  
2  
3      # Sets pass by reference  
4  
5      def copy_and_clear(a_set):  
6          # copy the set  
7          a_set_copy = a_set.copy()  
8  
9          # clear the original set  
10         a_set.clear()  
11  
12         return a_set_copy  
13  
14  
15         # call the function you defined  
16         set1 = {'A', 'B', 'C', 'D', 'E'}  
17         print('before the function call -> set1:', set1)  
18         set1_copy = copy_and_clear(set1)  
19         print('after the function call -> set1:', set1)  
20         print('set1_copy:', set1_copy)
```

[
8
]:

before the function call -> set1: {'E', 'C', 'D', 'A', 'B'}

after the function call -> set1: set()
set1_copy: {'E', 'C', 'A', 'B', 'D'}

S9:

```
[9]: 1 # S 9:
      2
      3 def remove_common_elements(set_1, set_2):
      4     # update the parameter in place
      5     # difference_update() removes the common
      6     elements
      7         set_1.difference_update(set_2)
      8
      9     # call the function you defined
     10    set_1 = {'a', 'b', 'c', 'd', 'e', 'f'}
     11    set_2 = {'d', 'b', 'e', 'f', 'h', 'g'}
     12    print('before the function -> set_1:', set_1)
     13    remove_common_elements(set_1, set_2)
     14    print('after the function -> set_1:', set_1)
```

```
[9]: before the function -> set_1: {'d', 'b', 'c', 'a', 'e', 'f'}
      after the function -> set_1: {'c', 'a'}
```

S10:

```
[1
0]: 1 # S 10:
      2
      3 def which_one_is_superset(set_1, set_2):
      4     if set_1.issuperset(set_2):
      5         return "set_1: {0} is superset of set_2:
      6             {1}.".format(set_1, set_2)
      7     else:
      8         return "set_2: {0} is superset of set_1:
      9             {1}.".format(set_2, set_1)
```

```
10  # call the function you defined
11  set_1 = {'a', 'b', 'c', 'd'}
12  set_2 = {'d', 'b', 'e', 'f', 'a', 'c'}
13  print(which_one_is_superset(set_1, set_2))
14
15  set_1 = {'d', 'b', 'e', 'f', 'a', 'c'}
16  set_2 = {'a', 'b', 'c', 'd'}
17  print(which_one_is_superset(set_1, set_2))
```

[1
0]:

```
set_2: {'d', 'b', 'c', 'a', 'e', 'f'} is superset of
set_1: {'b', 'd', 'a', 'c'}.
set_1: {'d', 'b', 'c', 'a', 'e', 'f'} is superset of
set_2: {'b', 'd', 'a', 'c'}
```

18. Comprehension

What is Comprehension?

Comprehension is an easy way to loop over sequences. It offers a shorter syntax for almost any operation you do with the loops. This chapter is devoted to Comprehensions. You will learn List, Dictionary and Set Comprehensions, how to set Nested Comprehensions and how to use Conditional Statements in Comprehensions.

List Comprehension:

Let's assume, you want to create a new list based on the values of an existing list. You can achieve this with loops and slicing in some cases. But there is a better way in Python: List Comprehension. In the image below you can see an example for the syntax of a list comprehension:

Output	Collection	Condition
<code>i**2</code>	<code>for i in range(1, 11)</code>	<code>if i % 2 == 1</code>

`[do this on this collection in this situation]`

Figure 18-1: An example syntax for List Comprehension

Example:

Create a list that contains the squares of numbers from 1 to 10. First, we will solve this question with the classical method, a **for** loop. Then we will do the same operation with a comprehension.

```
[  
1  1      # classical method -> loop  
]:  
2  
3  # define an empty list  
4  squares = []  
5  
6  # loop over the range  
7  for i in range(1, 11):  
8      squares.append(i**2)  
9  
10 print(squares)
```

```
[  
1  [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]  
]:
```

In cell 1, we first define an empty list to store the squares of numbers. Then we set a **for** loop on the **range(1, 11)**. At each iteration we calculate the square of the loop variable **i**, and we append it to the **squares** list. This is actually the long way. Let's do the same operation with a comprehension:

```
[  
2  1  # comprehension  
]:  
2  squares_comp = [i**2 for i in range(1, 11)]  
3  
4  print(squares_comp)
```

```
[2]: [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
]:
```

In cell 2, we set a list comprehension. We create list comprehensions with a pair of square brackets []. The expression of `i**2` is the output and `for i in range(1, 11)` is the collection. It tells Python that, we want a list of squares of numbers in that range. Here is the comprehension: `[i**2 for i in range(1, 11)]`.

Example:

Create a list that contains the cubes of numbers from 1 to 7.

```
[3]: 1 # classical method - loops
2
3 # define an empty list
4 cubes = list()
5
6 # loop over the range
7 for k in range(1, 7):
8     cubes.append(k**3)
9
10 cubes
```

```
[3]: [1, 8, 27, 64, 125, 216]
]:
```

In cell 3, we solve the question with loops. We loop over the `range(1, 7)` and append the cube of the current number to the `cubes` list. Now let's do same thing with a comprehension:

```
[4]: 1 # comprehension
      2 cubes_comp = [k**3 for k in range(1, 7)]
      3
      4 cubes_comp
      5 cubes
```

```
[4]: [1, 8, 27, 64, 125, 216]
[5]:
```

You see the comprehension in cell 4 as: `[k**3 for k in range(1, 7)]`. Here `k**3` is the output and `for k in range(1, 7)` is the collection. The comprehension returns a list of cubes in this range.

Example:

Convert the letters of '`lorem ipsum`' to uppcases and append them to a list.

```
[5]: 1 # define the string variable
      2 text = 'lorem ipsum'
[6]: 1 # classical method -> loops
      2
      3 # define an empty list
      4 capitals = []
      5
      6 # loop over the letters
      7 for letter in text:
```

```
8     capitals.append(letter.upper())
9
10    capitals
```

```
[ 6
]:      ['L', 'O', 'R', 'E', 'M', ' ', 'I', 'P', 'S', 'U', 'M']
```

In cell 6, we loop over the letters in the text of **'lorem ipsum'**. We append the uppercase form of each **letter** to the **capitals** list inside the **for** loop. Let's see the comprehension form for this question:

```
[ 7
]: 1 # Comprehension
  2 capitals_comp = [letter.upper()
  3             for letter in text]
  4
  5 capitals_comp
```

```
[ 7
]:      ['L', 'O', 'R', 'E', 'M', ' ', 'I', 'P', 'S', 'U', 'M']
```

You see the comprehension in cell 7. This time we write it in 2 lines. The expression of **letter.upper()** is the output and **for letter in text** is the collection. Here is the comprehension : **[letter.upper() for letter in text]**. It returns a list of uppercase letters in the **'lorem ipsum'** text.

Dictionary Comprehension:

Dictionary Comprehension is very similar to List Comprehension. The main difference is that, it returns a **Dictionary** instead of a List. Dictionary Comprehensions are defined with a pair curly brackets {}.

Example:

We have two lists: Programming Languages and Release Years:

- languages = ['Python', 'Java', 'JavaScript', 'C#']
- years = [1989, 1995, 1995, 2000]

We will combine these two lists in a dictionary. The language will be the key, and the release year will be the value. First we will do this with a **for** loop then we will set a Dictionary Comprehension.

```
[  
8 1 # define the lists  
]:  
2 languages = ['Python', 'Java', 'JavaScript', 'C#']  
3 years = [1989, 1995, 1995, 2000]
```

```
[  
9 1 # classical method -> loop  
]:  
2  
3 # define an empty dict  
4 lang_year = {}  
5  
6 # loop over both lists with zip()  
7 for lang, year in zip(languages, years):  
8     lang_year[lang] = year
```

```
9  
10 lang_year
```

```
[9]: {'Python': 1989, 'Java': 1995, 'JavaScript': 1995,  
]: 'C#': 2000}
```

In cell 9, we loop over both lists with the help of the `zip()` function as: `zip(languages, years)`. The items that `zip()` function returns are tuples. And we deconstruct these tuples into two variables: `lang` and `year`. Then we add them into the `lang_year` dict as: `lang_year[lang] = year`.

```
[1  
0]: 1 # Comprehension  
2 lang_year_comp = {lang: year  
3           for lang, year in zip(languages, years)}  
4  
5 lang_year_comp
```

```
[1  
0]: {'Python': 1989, 'Java': 1995, 'JavaScript': 1995,  
]: 'C#': 2000}
```

We do the same operation with a dictionary comprehension in cell 10. Here is the comprehension:

{lang: year for lang, year in zip(languages, years)}

The expression of `lang: year` is the output. Since this is a dictionary comprehension, the output is in the form of `key:value` pairs. And `for lang, year in zip(languages, years)` is the collection. Comprehension returns a dictionary.

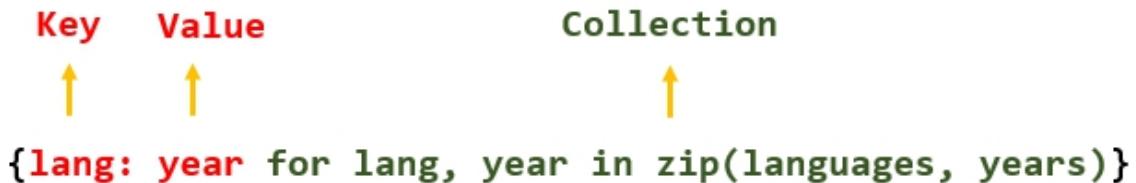


Figure 18-2: An example syntax for Dictionary Comprehension

Set Comprehension:

The last type of comprehension which we cover is the Set Comprehension. Its syntax is very similar to the List Comprehension. The difference is that, it returns a **Set** instead of a List. Set Comprehensions are defined with a pair curly brackets {}.

Example:

Let's create a set containing the letters of 'pepper'.

```
[1] 1 # define the variable
1]: 2 pepper = 'pepper'
```

```
[1] 1      # classical method -> loop
2
3      # define an empty set
4 letters = set()
5
6      # loop over the items
7 for l in pepper:
8     letters.add(l)
9
10 letters
```

```
[1] 1
2]: 2      {'e', 'p', 'r'}
```

In cell 12, we loop over the elements of the `pepper` string and we add them to the `letters` set. Let's do it with a Set Comprehension:

```
[1 3]: 1 # comprehension
        2 letters_comp = {l for l in pepper}
        3 letters_comp
        4 letters
```

```
[1 3]: {'e', 'p', 'r'}
```

We create a Set Comprehension in cell 13 as : `{l for l in pepper}`. Here, `l` is the output and `for l in pepper` is the collection. The comprehension returns a set of unique letters in the `pepper` text.

Nested Comprehensions

We use nested comprehensions like we use nested loops. The idea is almost the same. Let's see with examples:

Example:

We have two lists and we want to create a new list of Tuples. The Tuple elements will be the pairs of the items from these lists. Here are the lists and the expected output:

```
[1 4]: 1 # lists
        2 letters = ['A', 'B']
        3 numbers = [1, 2, 3]
```

```
[  
('A', 1),  
('A', 2),  
('A', 3),  
...  
]
```

Let's start with the classical method, the loops. We will set nested loops over the lists:

```
[1 5]: 1 # classical method -> loops  
2  
3 # define an empty list  
4 result = []  
5  
6 # nested loops on the lists  
7 for letter in letters:  
8     for number in numbers:  
9         tup = (letter, number)  
10        result.append(tup)  
11  
12 print(result)
```

```
[1 5]: [ ('A', 1), ('A', 2), ('A', 3), ('B', 1), ('B', 2), ('B', 3)]
```

In cell 15, the outer **for** loop is on **letters** and the inner one is on **numbers**. We create a tuple of a **letter** and a **number** in line 9. Then we append this tuple to the **result** list.

```
[1 6]: 1 # nested comprehension  
2 result_comp = [(letter, number)  
3                 for letter in letters]
```

```
4                 for number in numbers]
5 result_comp
[1
6]:      [('A', 1), ('A', 2), ('A', 3), ('B', 1), ('B', 2), ('B', 3)]
```

You see a nested comprehension in cell 16. The output is the tuple of **(letter, number)**. And the collection is a pair of nested **for** loops as: **for letter in letters for number in numbers**. We write it in two separate lines to make it easy to read.

Example:

We want to create a dictionary for numbers from 1 to 10. The number will be the key and a list of all the numbers which are less than or equal to that number will be the value. Here is the dictionary:

```
smallers = {
    1: [1],
    2: [1, 2],
    3: [1, 2, 3],
    4: [1, 2, 3, 4],
    5: [1, 2, 3, 4, 5],
    6: [1, 2, 3, 4, 5, 6],
    7: [1, 2, 3, 4, 5, 6, 7],
    8: [1, 2, 3, 4, 5, 6, 7, 8],
    9: [1, 2, 3, 4, 5, 6, 7, 8, 9],
    10: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}
```

Let's create this dictionary with nested loops first:

```
[1
7]: 1      # classical method -> loops
```

```

2
3     # empty dict
4     smallers = dict()
5
6     for i in range(1, 11):
7         for j in range(1, i+1):
8             # if already not in dict
9             if not i in smallers:
10                 smallers[i] = [j]
11             else:
12                 smallers[i].append(j)
13
14 smallers

```

[1
7]:

```

{1: [1],
2: [1, 2],
3: [1, 2, 3],
4: [1, 2, 3, 4],
5: [1, 2, 3, 4, 5],
6: [1, 2, 3, 4, 5, 6],
7: [1, 2, 3, 4, 5, 6, 7],
8: [1, 2, 3, 4, 5, 6, 7, 8],
9: [1, 2, 3, 4, 5, 6, 7, 8, 9],
10: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}

```

In cell 17, we create a nested loop structure. The first one iterates on `range(1, 11)` and the second one iterates on `range(1, i+1)`. We add the number `i` as the key to the `smallers` dict. In line 9, we check whether this key exists or not. If it doesn't exist, then in line 10, we add it as: `smallers[i] = [j]`. Be careful, here the value is a list: `[j]`. And if the number `i` already exists in the dict,

then we update the value list in line 12 as:
smallers[i].append(j).

Now we will set a nested comprehension for this question:

```
[1] 1 # nested comprehension
[8]: 2 smallers_comp = {i: [j for j in range(1, i+1)]
[3]           for i in range(1, 11)}
[4]
[5] smallers_comp
```

```
[1] {1: [1],
[8]: 2: [1, 2],
3: [1, 2, 3],
4: [1, 2, 3, 4],
5: [1, 2, 3, 4, 5],
6: [1, 2, 3, 4, 5, 6],
7: [1, 2, 3, 4, 5, 6, 7],
8: [1, 2, 3, 4, 5, 6, 7, 8],
9: [1, 2, 3, 4, 5, 6, 7, 8, 9],
10: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]}
```

In cell 18, you see a nested comprehension as:

```
{i: [j for j in range(1, i+1)] for i in range(1, 11)}
```

The outer comprehension is dictionary comprehension. And for this comprehension, the output is: **i: [j for j in range(1, i+1)]**. And the collection is: **for i in range(1, 11)**.

We have another comprehension inside the output of the first one which is: **[j for j in range(1, i+1)]**. This one is a List

Comprehension. And inside this inner comprehension, `j` is the output and `for j in range(1, i+1)` is the collection.

Conditional Statements in Comprehensions

Let's see how we use conditional statements in Comprehensions.

Example:

Let's find the odd numbers from 1 to 20.

```
[1
9]: 1      # classical method -> loop
      2
      3      # empty list
      4      odds = []
      5
      6      for i in range(1, 21):
      7          if i % 2 == 1:
      8              odds.append(i)
      9
     10     odds
```

```
[1
9]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

We print the list of odd number from 1 to 20 in cell 19. We use a `for` loop for this. And inside the loop we check if the number is an odd number. Let's use a comprehension for the same task:

Rule of Thumb:

if-else structure in Comprehension is exactly the same as in loops.

```
[2 0]: 1 # comprehension
        2 odds_comp = [i
                      for i in range(1, 21)
                      if i % 2 == 1]
        5
        6 odds_comp
```

```
[2 0]: [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
```

In cell 20, you see the structure of a comprehension with conditional statements. We write it in multiple lines but it can also be in the same line. Here is the syntax:

[i for i in range(1, 21) if i % 2 == 1]

The idea is the same as in loops. We first have the collection which is: **for i in range(1, 21)**. Then we add our condition as: **if i % 2 == 1**. As we already know, **i** is the output.

Example:

We want to create a dictionary for numbers from 2 to 20. The number will be the key and the value will be a list. This list is going to be the positive factors of that number. Here is the expected output:

```
factors = {
        2: [2],
        3: [3],
        4: [2, 4],
```

```
5: [5],  
6: [2, 3, 6]  
7: [7],  
8: [2, 4, 8],  
9: [3, 9],  
10: [2, 5, 10],  
...  
}
```

Let's start with the loops:

```
[2  
1]: 1 # classical method -> loops  
2  
3 # empty dict  
4 factors = {}  
5  
6 for i in range(2, 21):  
7     for j in range(2, i+1):  
8         # check if its a factor  
9         if i % j == 0:  
10             # check if not already in dictionary  
11             if not i in factors:  
12                 factors[i] = [j]  
13             else:  
14                 factors[i].append(j)  
15  
16 factors
```

```
[2  
1]: {2: [2],  
3: [3],  
4: [2, 4],  
5: [5],  
6: [2, 3, 6],
```

```
7: [7],  
8: [2, 4, 8],  
9: [3, 9],  
10: [2, 5, 10],  
...  
}
```

In cell 21, we set two loops. The first one iterates over the `range(2, 21)` and the inner one iterates over `range(2, i+1)`. The loop variable of the outer one is `i`, and the loop variable of the inner one is `j`. In line 9, we check if `i` is divisible by `j`. If it is divisible, then we check if `i` exists in the factors dictionary. If it's the first time that `i` occurs in the dictionary we add it as: `factors[i] = [j]`. Be careful, here the value is a list: `[j]`. And if the number `i` already exists in the dict, then we update the value list in line 14 as: `factors[i].append(j)`.

```
[2]  
[2]: 1 # comprehension  
2 factors_comp = {  
3     i: [j for j in range(2, i+1) if i % j == 0]  
4     for i in range(2, 21)  
5 }  
6  
7 factors_comp
```

```
[2]  
[2]: {2: [2],  
3: [3],  
4: [2, 4],  
5: [5],  
6: [2, 3, 6],
```

```
7: [7],  
8: [2, 4, 8],  
9: [3, 9],  
10: [2, 5, 10],  
...  
}
```

In cell 22, you see the dictionary comprehension for the same question. We write it in separate lines to make it easy to read. Here is the comprehension:

```
{  
    i: [j for j in range(2, i+1) if i % j == 0]  
    for i in range(2, 21)  
}
```

Here the output of the outer comprehension is: `i: [j for j in range(2, i+1) if i % j == 0]`. And the collection is: `for i in range(2, 21)`.

The inner comprehension is a List Comprehension which is: `[j for j in range(2, i+1) if i % j == 0]`. Here, `j` is the output, `for j in range(2, i+1)` is the collection and `if i % j == 0` is the condition.

Example:

We want to create a dictionary for **even** numbers from 2 to 20. The number will be the key and the value will be a list. This list is going to be the positive factors of that number. Here is the expected output:

```
factors_of_evens = {  
    2: [2],
```

```

4: [2, 4],
6: [2, 3, 6],
8: [2, 4, 8],
10: [2, 5, 10],
12: [2, 3, 4, 6, 12],
14: [2, 7, 14],
16: [2, 4, 8, 16],
18: [2, 3, 6, 9, 18],
20: [2, 4, 5, 10, 20]
}

```

Let's start with the loops:

```

[2
3]: 1      # classical method -> loops
      2
      3      # empty dict
      4      factors_of_evens = {}
      5
      6      for i in range(2, 21):
              # check if i is even
              if i % 2 == 0:
                  for j in range(2, i+1):
                      # check if its a factor
                      if i % j == 0:
                          # check if not already in dictionary
                          if not i in factors_of_evens:
                              factors_of_evens[i] = [j]
                          else:
                              factors_of_evens[i].append(j)
      17
      18      factors_of_evens

```

```

[2
3]: {2: [2],
      
```

```
4: [2, 4],  
6: [2, 3, 6],  
8: [2, 4, 8],  
10: [2, 5, 10],  
12: [2, 3, 4, 6, 12],  
14: [2, 7, 14],  
16: [2, 4, 8, 16],  
18: [2, 3, 6, 9, 18],  
20: [2, 4, 5, 10, 20]}
```

The difference of this example from the previous one is that, we are looking for the even numbers in the same range. That's why we add the condition of `if i % 2 == 0` in line 8. The rest is the same as the previous example.

```
[2  
4]: 1 # comprehension  
2 factors_of_evens_comp = { i: [j for j in range(2,  
i+1) if i % j == 0]  
3  
4  
5 factors_of_evens_comp
```

```
[2  
4]: {2: [2],  
4: [2, 4],  
6: [2, 3, 6],  
8: [2, 4, 8],  
10: [2, 5, 10],  
12: [2, 3, 4, 6, 12],  
14: [2, 7, 14],  
16: [2, 4, 8, 16],  
18: [2, 3, 6, 9, 18],
```

```
20: [2, 4, 5, 10, 20]}
```

In cell 24, we create a nested comprehension for the same task.

Here it is:

```
{ i: [j for j in range(2, i+1) if i % j == 0]  
  for i in range(2, 21) if i % 2 == 0 }
```

Here the output of the outer comprehension is: `i: [j for j in range(2, i+1) if i % j == 0]`. The collection is : `for i in range(2, 21)` and the condition is: `if i % 2 == 0`.

The inner comprehension is a List Comprehension which is: `[j for j in range(2, i+1) if i % j == 0]`. Here, `j` is the output, `for j in range(2, i+1)` is the collection and `if i % j == 0` is the condition.

Important Note:

Try to use Comprehensions instead of loops whenever possible, because they provide better performance. And once you get used to them, they are cleaner and easier to read and maintain.

QUIZ - Comprehension

Now it's time to solve the quiz for this chapter. You can download the quiz file, **QUIZ_Comprehension.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *18_Comprehension*. Here are the questions for this chapter:

QUIZ - Comprehension:

Q1:

Define two functions named **get_squares** and **get_squares_comp**.

They will give the list of squares of numbers from 1 to 10 (exc.).

The first function will use for loops and the second one will use Comprehension.

Expected Output:

[1, 4, 9, 16, 25, 36, 49, 64, 81]

```
[  
1 1 # Q 1:  
]:  
2  
3 # for loop  
4 # ---- your solution here ----  
5  
6  
7 # call the function you defined  
8 squares = get_squares()  
9 print(squares)  
10  
11  
12 # comprehension  
13 # ---- your solution here ----
```

```
14
15
16 # call the function you defined
17 sqrs = get_squares_comp()
18 print(sqrs)
```

```
[1]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
[2]: [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Q2:

Define two functions named **get_sentences** and **get_sentences_comp**.

They will take a paragraph string as parameter.
And they will return the sentences as list.

The first function will use for loops and second one will use Comprehension.

paragraph = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut enim ad minim veniam, quis nostrud exercitation ullamco. Duis aute irure dolor in reprehenderit in voluptate velit esse. Excepteur sint occaecat cupidatat non proident."

Expected Output:

['Lorem ipsum dolor sit amet, consectetur adipiscing elit', 'Ut enim ad minim veniam, quis nostrud exercitation ullamco', 'Duis aute irure dolor in reprehenderit in voluptate velit esse', 'Excepteur sint occaecat cupidatat non proident.']}

```
[1]: # Q 2:
[2]: 2
[3]: 3 paragraph = "Lorem ipsum dolor sit amet,
consectetur adipiscing elit. Ut enim ad minim
veniam, quis nostrud exercitation ullamco. Duis
aute irure dolor in reprehenderit in voluptate velit
```

esse. Excepteur sint occaecat cupidatat non
proident."

```
4  
5  
6 # for loop  
7 # ---- your solution here ----  
8  
9  
10 # call the function you defined  
11 sentences = get_sentences(paragraph)  
12 print(sentences)  
13  
14  
15  
16 # comprehension  
17 # ---- your solution here ----  
18  
19  
20 # call the function you defined  
21 sentences_comp =  
22 get_sentences_comp(paragraph)  
print(sentences_comp)
```

[
2
]:

```
['Lorem ipsum dolor sit amet, consectetur  
adipiscing elit', 'Ut enim ad minim veniam, quis  
nostrud exercitation ullamco', 'Duis aute irure  
dolor in reprehenderit in voluptate velit esse',  
'Excepteur sint occaecat cupidatat non proident.'][  
'Lorem ipsum dolor sit amet, consectetur  
adipiscing elit', 'Ut enim ad minim veniam, quis  
nostrud exercitation ullamco', 'Duis aute irure  
dolor in reprehenderit in voluptate velit esse',  
'Excepteur sint occaecat cupidatat non proident.'][
```

Q3:

Use the functions in Q2 and this time define two functions named **get_words** and **get_words_comp**.

These functions will return the words in the paragraph.

The first one will use for loop and the second one will use Comprehension.

paragraph = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut enim ad minim veniam, quis nostrud exercitation ullamco. Duis aute irure dolor in reprehenderit in voluptate velit esse. Excepteur sint occaecat cupidatat non proident."

Expected Output:

```
['Lorem', 'ipsum', 'dolor', 'sit', 'amet,', 'consectetur',  
 'adipiscing', 'elit.', 'Ut', 'enim', 'ad', 'minim', 'veniam,', 'quis',  
 'nostrud', 'exercitation', 'ullamco.', 'Duis', 'aute', 'irure',  
 'dolor', 'in', 'reprehenderit', 'in', 'voluptate', 'velit', 'esse.',  
 'Excepteur', 'sint', 'occaecat', 'cupidatat', 'non', 'proident.']}
```

```
[  
3 1      # Q 3:  
]:  
2  
3      paragraph = "Lorem ipsum dolor sit amet,  
4      consectetur adipiscing elit. Ut enim ad minim  
5      veniam, quis nostrud exercitation ullamco. Duis  
6      aute irure dolor in reprehenderit in voluptate velit  
7      esse. Excepteur sint occaecat cupidatat non  
8      proident."  
9  
10     # for loop  
11     # ---- your solution here ----  
12  
13     # call the function you defined  
14     words = get_words(paragraph)  
15     print(words)
```

```

14
15
16 # comprehension
17 # ---- your solution here ----
18
19
20 # call the function you defined
21 words_comp = get_words_comp(paragraph)
22 print(words_comp)

```

[
3
]:

```

['Lorem', 'ipsum', 'dolor', 'sit', 'amet,',  

 'consectetur', 'adipiscing', 'elit', 'Ut', 'enim', 'ad',  

 'minim', 'veniam,', 'quis', 'nostrud', 'exercitation',  

 'ullamco', 'Duis', 'aute', 'irure', 'dolor', 'in',  

 'reprehenderit', 'in', 'voluptate', 'velit', 'esse',  

 'Excepteur', 'sint', 'occaecat', 'cupidatat', 'non',  

 'proident.']
['Lorem', 'ipsum', 'dolor', 'sit', 'amet,',  

 'consectetur', 'adipiscing', 'elit', 'Ut', 'enim', 'ad',  

 'minim', 'veniam,', 'quis', 'nostrud', 'exercitation',  

 'ullamco', 'Duis', 'aute', 'irure', 'dolor', 'in',  

 'reprehenderit', 'in', 'voluptate', 'velit', 'esse',  

 'Excepteur', 'sint', 'occaecat', 'cupidatat', 'non',  

 'proident.']

```

Q4:

Use the functions in Q2 and this time define two functions named **get_words_starting_with_vowels** and **get_words_starting_with_vowels_comp**.

These functions will return the words in the paragraph. The first one will use for loop and the second one will use Comprehension.

paragraph = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut enim ad minim veniam, quis nostrud exercitation ullamco. Duis aute irure dolor in reprehenderit in voluptate velit esse. Excepteur sint occaecat cupidatat non proident."

Expected Output:

```
['ipsum', 'amet,', 'adipiscing', 'elit.', 'Ut', 'enim', 'ad', 'exercitation', 'ullamco.', 'aute', 'irure', 'in', 'in', 'esse.', 'Excepteur', 'occaecat']
```

```
[  
4 1    # Q 4:  
]:  
2  
3    paragraph = "Lorem ipsum dolor sit amet,  
        consectetur adipiscing elit. Ut enim ad minim  
        veniam, quis nostrud exercitation ullamco. Duis  
        aute irure dolor in reprehenderit in voluptate velit  
        esse. Excepteur sint occaecat cupidatat non  
        proident."  
4  
5  
6    # for loop  
7    # ---- your solution here ----  
8  
9  
10   # call the function you defined  
11   vowel_words =  
12   get_words_starting_with_vowels(paragraph)  
13   print(vowel_words)  
14  
15  
16   # comprehension  
17   # ---- your solution here ----  
18  
19  
20   # call the function you defined  
21   vowels_comp =  
        get_words_starting_with_vowels_comp(paragraph  
    )
```

```
22  print(vowels_comp)
```

```
[  
4  ['ipsum', 'amet,', 'adipiscing', 'elit', 'Ut', 'enim',  
]:  'ad', 'exercitation', 'ullamco', 'aute', 'irure', 'in',  
  'in', 'esse', 'Excepteur', 'occaecat']  
  ['ipsum', 'amet,', 'adipiscing', 'elit', 'Ut', 'enim',  
  'ad', 'exercitation', 'ullamco', 'aute', 'irure', 'in',  
  'in', 'esse', 'Excepteur', 'occaecat']
```

Q5:

Define a function named **dict_of_squares**.

It will return a dictionary of squares of numbers from 0 to 20 (inc.).

Key will be the number and the value will be the square of this number:

{number: square}.

Hints:

- no loops
- use comprehension only

Expected Output:

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81,  
10: 100, 11: 121, 12: 144, 13: 169, 14: 196, 15: 225, 16: 256,  
17: 289, 18: 324, 19: 361, 20: 400}
```

```
[  
5  1  # Q 5:  
]:  
2  
3  # ---- your solution here ----  
4  
5  
6  
7  # call the function you defined  
8  sqr_dict = dict_of_squares()  
9  print(sqr_dict)
```

```
[  
5  
]: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8:  
64, 9: 81, 10: 100, 11: 121, 12: 144, 13: 169, 14:  
196, 15: 225, 16: 256, 17: 289, 18: 324, 19: 361, 20:  
400}
```

Q6:

Define two functions named **numbers_and_names** and **numbers_and_names_comp**.

They will take a dictionary as parameter.

The dictionary is:

```
{1: ['Spiderman', 24],  
2: ['Wonder Woman', 100],  
3: ['Batman', 45],  
4: ['Superman', 70]}
```

They will return a new dictionary containing only the numbers as the key and the name as value.

Expected Output:

```
{1: 'Spiderman', 2: 'Wonder Woman', 3: 'Batman', 4:  
'Superman'}
```

```
[  
6 1      # Q 6:  
]:  
2  
3      heroes = {1: ['Spiderman', 24],  
4          2: ['Wonder Woman', 100],  
5          3: ['Batman', 45],  
6          4: ['Superman', 70]}  
7  
8  
9      # for loop  
10     # ---- your solution here ----  
11  
12  
13     # call the function you defined  
14     print(numbers_and_names(heroes))  
15
```

```
16
17
18 # comprehension
19 # ---- your solution here ----
20
21
22 # call the function you defined
23 print(numbers_and_names_comp(heroes))
```

```
[6]: {1: 'Spiderman', 2: 'Wonder Woman', 3: 'Batman',
      4: 'Superman'}
      {1: 'Spiderman', 2: 'Wonder Woman', 3: 'Batman',
      4: 'Superman'}
```

Q7:

Define a function named **odd_numbered_names**.

It will take a dictionary as parameter.

The dictionary is:

```
{1: ['Spiderman', 24],
2: ['Wonder Woman', 100],
3: ['Batman', 45],
4: ['Superman', 70]}
```

It will return a new dictionary containing only the odd numbers as the key and the name as value.

Hints:

- no loops
- only with comprehension

Expected Output:

```
{1: 'Spiderman', 3: 'Batman'}
```

```
[7]: 1      # Q 7:
      2
      3      # ---- your solution here ----
```

```

4
5
6  # call the function you defined
7  heroes = {1: ['Spiderman', 24],
8      2: ['Wonder Woman', 100],
9      3: ['Batman', 45],
10     4: ['Superman', 70]}
11
12 names_dict = odd_numbered_names(heroes)
13 print(names_dict)

```

```

[7]: {1: 'Spiderman', 3: 'Batman'}
]:
```

Q8:

Define a function named **word_lengths**.

It will take a text as parameter.

The function will return two lists.

The first list is going to be the words in the text.

And the second list will contain the lengths of these words.

If the word is one ('the', 'in', 'as', 'at') it will ignore the word.

Hints:

- how a function returns multiple values -> Tuple
- no loops
- comprehensions only

Parameter:

"It takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

Expected Output:

```
['It', 'takes', 'all', 'running', 'you', 'can', 'do', 'to', 'keep',
'same', 'place.', 'If', 'you', 'want', 'to', 'get', 'somewhere',
'else', 'you', 'must', 'run', 'least', 'twice', 'fast', 'that!']
```

```
[2, 5, 3, 7, 3, 3, 3, 2, 4, 4, 6, 2, 3, 4, 2, 3, 9, 5, 3, 4, 3, 5, 5, 4, 5]
```

```
[8] : 1 # Q 8:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 # Alice's Adventures in Wonderland  
8 alice_in_wonderland = """It takes all the  
running you can do, to keep in the same place.  
If you want to get somewhere else, you must run  
at least twice as fast as that!"""  
9  
10  
11 # call the function and unpack the returning  
12 Tuple  
13 words, lengths =  
14 word_lengths(alice_in_wonderland)  
15  
16 # now print them  
17 print(words)  
18 print(lengths)
```

```
[8] : ['It', 'takes', 'all', 'running', 'you', 'can', 'do,',  
'to', 'keep', 'same', 'place.', 'If', 'you', 'want',  
'to', 'get', 'somewhere', 'else', 'you', 'must',  
'run', 'least', 'twice', 'fast', 'that!']  
[2, 5, 3, 7, 3, 3, 3, 2, 4, 4, 6, 2, 3, 4, 2, 3, 9, 5,  
3, 4, 3, 5, 5, 4, 5]
```

Q9:

Define a function named **only_positives**.

It will take a list of numbers (integers and floats) as parameter.

It will take only the positive numbers and cast them into integers and return the final list.

Hints:

- no loops
- use comprehension

Parameter:

```
[12.8, -27.2, -34.5, 58.4, -82.0, 66.6, 14.9]
```

Expected Output:

```
[12, 58, 66, 14]
```

```
[  
9  1 # Q 9:  
]:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 numbers = [12.8, -27.2, -34.5, 58.4, -82.0, 66.6,  
8 14.9]  
9 positives = only_positives(numbers)  
9 print(positives)
```

```
[  
9  [12, 58, 66, 14]  
]:
```

Q10:

Define a function named **flatten**.

It will take a list as parameter.

This list will be containing lists as elements.

Namely it is a nested list.

The function will flatten the list and return a final list which contains only the individual elements. The final list not a nested list anymore.

Hints:

- no loops
- comprehension only

Parameter:

```
list_of_lists = [[8,6], [3,1,4], [2,0,9], [5]]
```

Expected Output:

```
[8, 6, 3, 1, 4, 2, 0, 9, 5]
```

```
[1 0]: 1 # Q 10:  
2  
3 # ---- your solution here ----  
4  
5  
6 # call the function you defined  
7 list_of_lists = [[8,6], [3,1,4], [2,0,9], [5]]  
8 print(flatten(list_of_lists))
```

```
[1 0]: [8, 6, 3, 1, 4, 2, 0, 9, 5]
```

OceanofPDF.com

SOLUTIONS - Comprehension

Here are the solutions for the quiz for Chapter 18 - Comprehension.

SOLUTIONS - Comprehension:

S1:

```
[  
1 1      # S 1:  
]:  
2  
3 # for loop  
4 def get_squares():  
5     squares = []  
6     for i in range(1, 10):  
7         squares.append(i**2)  
8     return squares  
9  
10  
11 # call the function you defined  
12 squares = get_squares()  
13 print(squares)  
14  
15  
16 # comprehension  
17 def get_squares_comp():  
18     squares = [x**2 for x in range(1, 10)]  
19     return squares  
20  
21  
22 # call the function you defined  
23 sqrs = get_squares_comp()  
24 print(sqrs)
```

```
[  
1  
]: [1, 4, 9, 16, 25, 36, 49, 64, 81]  
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

S2:

```
[  
2 1 # S 2:  
]:  
2  
3 paragraph = "Lorem ipsum dolor sit amet,  
consectetur adipiscing elit. Ut enim ad minim  
veniam, quis nostrud exercitation ullamco. Duis  
aute irure dolor in reprehenderit in voluptate  
velit esse. Excepteur sint occaecat cupidatat non  
proident."  
4  
5  
6 # for loop  
7 def get_sentences(paragraph):  
8     sentences = []  
9     for sentence in paragraph.split('. '):  
10         sentences.append(sentence)  
11     return sentences  
12  
13  
14 # call the function you defined  
15 sentences = get_sentences(paragraph)  
16 print(sentences)  
17  
18  
19  
20 # comprehension  
21 def get_sentences_comp(paragraph):
```

```

22     # sentences = [sentence for sentence in
23     #return sentences
24     return [sentence for sentence in
25     paragraph.split('. ')]
26
27     # call the function you defined
28     sentences_comp =
29     get_sentences_comp(paragraph)
30     print(sentences_comp)

```

```

[2]: ['Lorem ipsum dolor sit amet, consectetur
adipiscing elit', 'Ut enim ad minim veniam, quis
nostrud exercitation ullamco', 'Duis aute irure
dolor in reprehenderit in voluptate velit esse',
'Excepteur sint occaecat cupidatat non proident.']
[2]: ['Lorem ipsum dolor sit amet, consectetur
adipiscing elit', 'Ut enim ad minim veniam, quis
nostrud exercitation ullamco', 'Duis aute irure
dolor in reprehenderit in voluptate velit esse',
'Excepteur sint occaecat cupidatat non proident.']

```

S3:

```

[3]: 1     # S 3:
2
3     paragraph = "Lorem ipsum dolor sit amet,
4     consectetur adipiscing elit. Ut enim ad minim
5     veniam, quis nostrud exercitation ullamco. Duis
6     aute irure dolor in reprehenderit in voluptate
7     velit esse. Excepteur sint occaecat cupidatat non
8     proident."
9
10

```

```
6  # for loop
7  def get_words(paragraph):
8      words = []
9
10     # get sentences of paragraph
11     sentences = get_sentences_comp(paragraph)
12
13     for sentence in sentences:
14         for word in sentence.split():
15             words.append(word)
16
17     return words
18
19
20     # call the function you defined
21     words = get_words(paragraph)
22     print(words)
23
24
25
26     # comprehension
27     def get_words_comp(paragraph):
28
29         # get sentences of paragraph
30         sentences = get_sentences_comp(paragraph)
31
32         words = [word
33             for sentence in sentences
34                 for word in sentence.split()]
35
36     return words
37
38
39     # call the function you defined
```

```
40     words_comp = get_words_comp(paragraph)
41     print(words_comp)

[3]:
```

```
['Lorem', 'ipsum', 'dolor', 'sit', 'amet,', 'consectetur', 'adipiscing', 'elit', 'Ut', 'enim', 'ad', 'minim', 'veniam,', 'quis', 'nostrud', 'exercitation', 'ullamco', 'Duis', 'aute', 'irure', 'dolor', 'in', 'reprehenderit', 'in', 'voluptate', 'velit', 'esse', 'Excepteur', 'sint', 'occaecat', 'cupidatat', 'non', 'proident.']

[4]:
```

```
['Lorem', 'ipsum', 'dolor', 'sit', 'amet,', 'consectetur', 'adipiscing', 'elit', 'Ut', 'enim', 'ad', 'minim', 'veniam,', 'quis', 'nostrud', 'exercitation', 'ullamco', 'Duis', 'aute', 'irure', 'dolor', 'in', 'reprehenderit', 'in', 'voluptate', 'velit', 'esse', 'Excepteur', 'sint', 'occaecat', 'cupidatat', 'non', 'proident.']
```

S4:

```
[4]:
```

```
1     # S 4:
2
3     paragraph = "Lorem ipsum dolor sit amet, consectetur adipiscing elit. Ut enim ad minim veniam, quis nostrud exercitation ullamco. Duis aute irure dolor in reprehenderit in voluptate velit esse. Excepteur sint occaecat cupidatat non proident."
4
5
6     # for loop
7     def get_words_starting_with_vowels(paragraph):
8
9         vowels = ['a', 'e', 'i', 'o', 'u']
```

```
11     words_of_vowels = []
12
13     # get sentences of paragraph
14     sentences = get_sentences_comp(paragraph)
15
16     for sentence in sentences:
17         for word in sentence.split():
18             if word[0].lower() in vowels:
19                 words_of_vowels.append(word)
20
21     return words_of_vowels
22
23
24     # call the function you defined
25     vowel_words =
26     get_words_starting_with_vowels(paragraph)
27     print(vowel_words)
28
29
30     # comprehension
31     def
32         get_words_starting_with_vowels_comp(paragrap
33         h):
34
35             vowels = ['a', 'e', 'i', 'o', 'u']
36
37             # get sentences of paragraph
38             sentences = get_sentences_comp(paragraph)
39
40             words_of_vowels = [word
41                             for sentence in sentences
42                             for word in sentence.split()
43                             if word[0].lower() in vowels]
```

```
42
43     return words_of_vowels
44
45
46     # call the function you defined
47     vowels_comp =
48     get_words_starting_with_vowels_comp(paragraph)
49     print(vowels_comp)
```

```
[  
4
]:      ['ipsum', 'amet', 'adipiscing', 'elit', 'Ut', 'enim',
         'ad', 'exercitation', 'ullamco', 'aute', 'irure', 'in',
         'in', 'esse', 'Excepteur', 'occaecat']  
[  
4
]:      ['ipsum', 'amet', 'adipiscing', 'elit', 'Ut', 'enim',
         'ad', 'exercitation', 'ullamco', 'aute', 'irure', 'in',
         'in', 'esse', 'Excepteur', 'occaecat']
```

S5:

```
[  
5
]:      1     # S 5:
[  
5
]:      2
[  
5
]:      3     def dict_of_squares():
[  
5
]:      4
[  
5
]:      5         squares = {num: num**2 for num in range(0,
[  
5
]:      6             21)}
[  
5
]:      7         return squares
[  
5
]:      8
[  
5
]:      9
[  
5
]:      10
[  
5
]:      11     # call the function you defined
[  
5
]:      12     sqr_dict = dict_of_squares()
[  
5
]:      13     print(sqr_dict)
```

[
5
]:

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49,  
8: 64, 9: 81, 10: 100, 11: 121, 12: 144, 13: 169,  
14: 196, 15: 225, 16: 256, 17: 289, 18: 324, 19:  
361, 20: 400}
```

S6:

```
[  
6 1      # S 6:  
]:  
2  
3 heroes = {1: ['Spiderman', 24],  
4      2: ['Wonder Woman', 100],  
5      3: ['Batman', 45],  
6      4: ['Superman', 70]}  
7  
8  
9      # for loop  
10     def numbers_and_names(heroes):  
11  
12     heroedict = {}  
13  
14     for num, name_and_age in heroes.items():  
15         heroedict[num] = name_and_age[0]  
16  
17     return heroedict  
18  
19  
20     # call the function you defined  
21     print(numbers_and_names(heroes))  
22  
23  
24  
25     # comprehension  
26     def numbers_and_names_comp(heroes):  
27
```

```
28     heroedict = {num: name_and_age[0]
29             for num, name_and_age in
30             heroes.items()}
31
32
33
34     # call the function you defined
35     print(numbers_and_names_comp(heroes))
```

```
[6]:
```

```
{1: 'Spiderman', 2: 'Wonder Woman', 3: 'Batman',
4: 'Superman'}
```

```
{1: 'Spiderman', 2: 'Wonder Woman', 3: 'Batman',
4: 'Superman'}
```

S7:

```
[7]:
```

```
1     # S 7:
2
3     def odd_numbered_names(heroes):
4
5         heroedict_odds = {key: value[0]
6             for key, value in heroes.items()
7             if key % 2 == 1}
8
9     return heroedict_odds
10
11
12
13     # call the function you defined
14     heroes = {1: ['Spiderman', 24],
15               2: ['Wonder Woman', 100],
16               3: ['Batman', 45],
```

```
17         4: ['Superman', 70]}

18
19     names_dict = odd_numbered_names(heroes)
20     print(names_dict)
```

```
[  
7
]:      {1: 'Spiderman', 3: 'Batman'}
```

S8:

```
[8
]:      1      # S 8:

2
3      def word_lengths(text):
4
5          # get the words first
6          words = text.split()
7
8          # forbidden words
9          forbiddens = ('the', 'in', 'as', 'at')
10
11         # comprehensions
12         word_list = [word
13             for word in words
14             if not word in forbiddens]

15
16         lengths = [len(word)
17             for word in words
18             if not word in forbiddens]

19
20         return (word_list, lengths)
21
22
23
24         # call the function you defined
```

```

25  # Alice's Adventures in Wonderland
26  alice_in_wonderland = """It takes all the running
27  you can do, to keep in the same place.
28  If you want to get somewhere else, you must run
29  at least twice as fast as that!"""
30
31
32  # call the function and unpack the returning
33  # Tuple
34  words, lengths =
35  word_lengths(alice_in_wonderland)
36
37
38  # now print them
39  print(words)
40  print(lengths)

```

[8]:

```

['It', 'takes', 'all', 'running', 'you', 'can', 'do', 'to',
'keep', 'same', 'place.', 'If', 'you', 'want', 'to',
'get', 'somewhere', 'else', 'you', 'must', 'run',
'least', 'twice', 'fast', 'that!']
[2, 5, 3, 7, 3, 3, 3, 2, 4, 4, 6, 2, 3, 4, 2, 3, 9, 5,
3, 4, 3, 5, 5, 4, 5]

```

S9:

[
9
]:

```

1  # S 9:
2
3  def only_positives(a_list):
4
5      return [int(x)
6              for x in a_list
7              if x > 0]
8
9
10
11  # call the function you defined

```

```
12     numbers = [12.8, -27.2, -34.5, 58.4, -82.0, 66.6,  
13         14.9]  
14     positives = only_positives(numbers)  
     print(positives)
```

```
[  
9  
]:  
[12, 58, 66, 14]
```

S10:

```
[1  
0]:  
1     # S 10:  
2  
3     def flatten(a_list):  
4  
5         return [j  
6             for i in a_list  
7                 for j in i]  
8  
9  
10  
11    # call the function you defined  
12    list_of_lists = [[8,6], [3,1,4], [2,0,9], [5]]  
13    print(flatten(list_of_lists))
```

```
[1  
0]:  
[8, 6, 3, 1, 4, 2, 0, 9, 5]
```

19. Project 3 - Snake Game

Preparation

This is our final project in this book. It will cover all the topics what we have seen so far. We will build our own snake game with Python. We will use built-in [`turtle`](#), [`time`](#) and [`random`](#) modules in this project. Here is an image of our snake game.

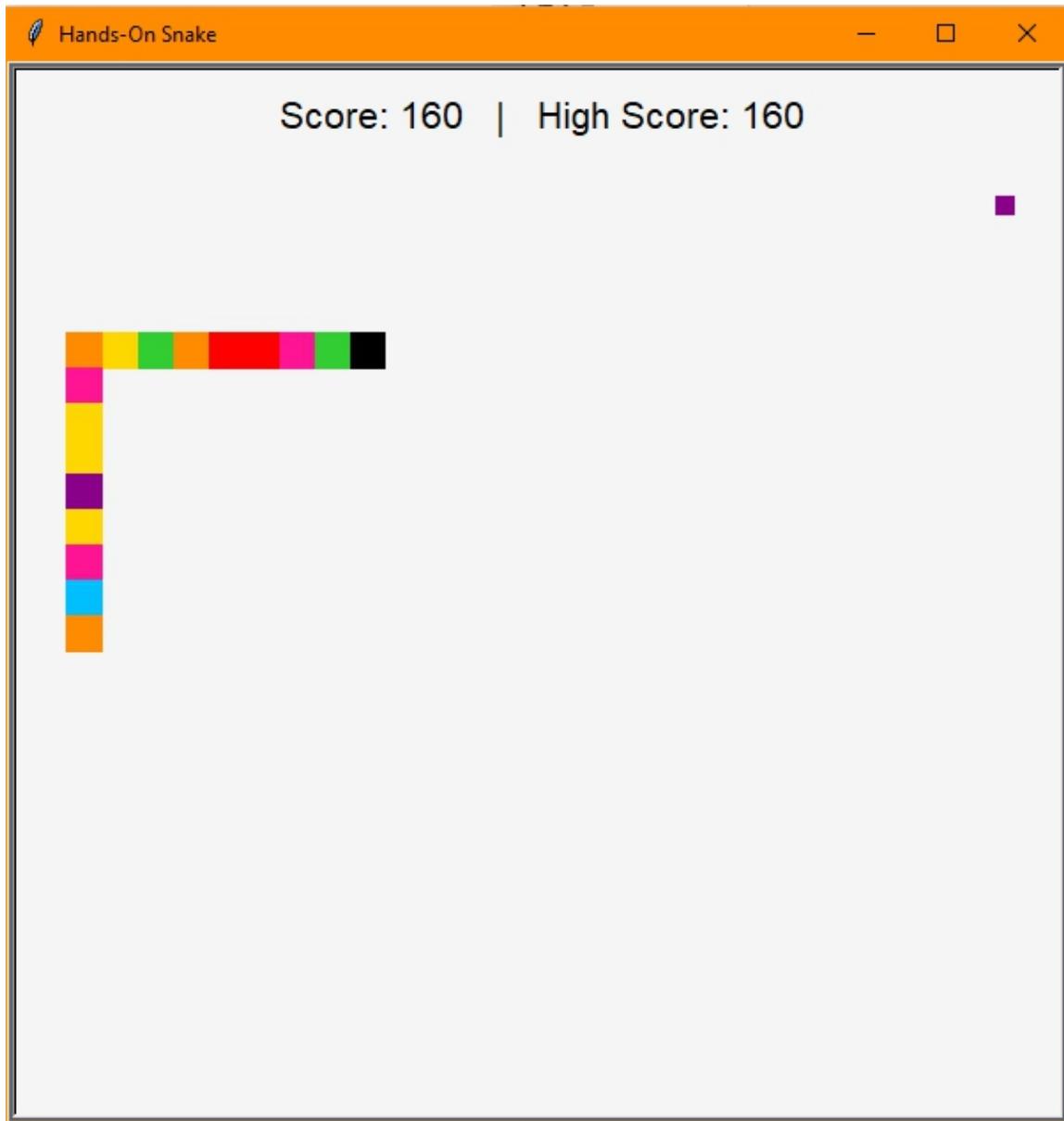


Figure 19-1: The snake game

Let's start by importing the modules:

```
[  
1  1 import turtle  
]:  
2  2 import time  
3  3 import random
```

Global Variables:

Next, we will define the global variables, which we will be using throughout the project:

```
[1] 1  # global variables
[2] 2
[3] 3  window = None
[4] 4  snake = []
[5] 5  head = None
[6]
[7] 7  direction = 'down'
[8] 8  delay_time = 0.1
[9]
[10] 10 food = None
[11] 11 food_color = ''
[12]
[13] 13 TURTLE_SIZE = 20
[14] 14 WIDTH = 600
[15] 15 HEIGHT = 600
[16]
[17] 17 X_RANGE = (WIDTH - TURTLE_SIZE) / 2
[18] 18 Y_RANGE = (HEIGHT - TURTLE_SIZE) / 2
[19]
[20] 20 pen = None
[21] 21 score = 0
[22] 22 high_score = 0
[23]
[24] 24 shapes = {
[25] 25     0: 'circle',
[26] 26     1: 'square',
[27] 27     2: 'triangle'
[28] 28 }
```

```
29
30 colors = {
31     0: 'gold',
32     1: 'lime green',
33     2: 'dark magenta',
34     3: 'red',
35     4: 'dark orange',
36     5: 'deep sky blue',
37     6: 'deep pink',
38     7: 'light sea green'
39 }
```

Here are the explanations for the global variables:

window: The main window object of the project. We instantiate it as **None**. We will assign **turtle.Screen** object to it later on.

snake: It is a list to store the head and the body parts of the snake.

head: It is an object to store the head of the snake.

direction: For keeping the direction of the snake. The default direction is down.

delay_time: The time interval in seconds to wait for delay.

food: The food object.

food_color: Color of the food.

TURTLE_SIZE: The size of our turtle in pixels.

WIDTH: Width of the screen.

HEIGHT: Height of the screen.

X_RANGE: Roughly the half of the screen in x direction.

Y_RANGE: Roughly the half of the screen in y direction.

pen: A turtle object to be able to write the score on the screen.

score: Current score.

high_score: Highest score.

shapes: the dictionary for the shapes of the foods.

colors: the dictionary for the colors we use.

Functions

Main Screen:

Now let's define a function, **set_screen**, to set up the main screen. It will use the global variable **window**, and assign **turtle.Screen** object to it. It will also set the title, background color, with and height of the screen.

```
[  
3 1     # function to set up the main screen  
]:  
2  
3 def set_screen():  
4     """Sets the main screen."""  
5  
6     global window  
7  
8     window = turtle.Screen()  
9     window.title('Hands-On Snake')  
10    window.bgcolor('white smoke')  
11    window.setup(width=WIDTH, height=HEIGHT)  
12    window.tracer(0) # False
```

The `window.tracer(0)` method in line 12 is used to turn turtle animation on or off and set a delay for update drawings. We don't want to have automatic screen updates, so we turn them off by setting `tracer()` to 0. 0 means False. We will update the screen manually.

Events:

We will play our snake game with the keyboard. So we need to listen the keyboard events. The keyboard event which we are interested in is: `onkeypress(fun, key)`. This function takes two arguments:

- **fun**: the name of the function to execute when this event occurs
- **key**: the string name of the key (e.g. "Down")

```
[  
4 1 # function to listen screen events  
]:  
2  
3 def listen_events():  
4     window.listen()  
5     window.onkeypress(set_up_direction, 'Up')  
6     window.onkeypress(set_down_direction, 'Down')  
7     window.onkeypress(set_left_direction, 'Left')  
8     window.onkeypress(set_right_direction, 'Right')
```

In the `listen_events` function we first call the `window.listen()` function to tell Python that we are going to listen the window events.

Now let's define the functions which we pass as arguments to the `window.onkeypress()` function:

```
[  
5  1  # keyboard functions  
]:  
2  
3  def set_up_direction():  
4      global direction  
5      if direction != 'down':  
6          direction = 'up'  
7  
8  def set_down_direction():  
9      global direction  
10     if direction != 'up':  
11         direction = 'down'  
12  
13 def set_left_direction():  
14     global direction  
15     if direction != 'right':  
16         direction = 'left'  
17  
18 def set_right_direction():  
19     global direction  
20     if direction != 'left':  
21         direction = 'right'
```

In cell 5, we define our window **event handlers**. Event handler means the function which gets called when an event occurs. We have four event handlers for four keyboard directions. Each of them modifies the global `direction` variable.

Let's say the user presses the 'up' key in the keyboard. As soon as the user presses the keyboard, `window.onkeypress(set_up_direction, 'Up')` event will get triggered. And this event will call our `set_up_direction` function.

Inside `set_up_direction` function we first tell Python that we will be using the `global direction` variable in line 4. Then we check if the current direction is 'not down'. Why? Because the snake will only move up, if it's currently **not moving down**. We don't want to the snake to get the opposite direction instantaneously. That's the main idea for all of the keyboard functions.

Head:

Now let's define a function to create the head of the snake.

```
[  
6 1     # create the head  
]:  
2  
3 def create_head(is_initial=True):  
4     """Creates the snake head."""  
5  
6     global head, snake  
7  
8     # create the head  
9     head = turtle.Turtle()  
10    head.shape(shapes[1]) # 20 x 20  
11    head.speed(0)  
12    head.penup()  
13
```

```
14      # start at a higher position
15      if is_initial:
16          head.goto(0, 200)
17
18      snake.append(head)
```

The `create_head` function takes one argument as `is_initial`. It's a bool variable to decide if the head should move to the starting position or not.

Inside the `create_head` function we modify the global `head` and `snake` variables. In line 9, it creates a Turtle object as `turtle.Turtle()` and assigns it to the global `head` variable. Then we modify the shape and speed of the head. We don't want the `head` to draw any shapes when moving, so we call `head.penup()` function to pull the pen up.

If the value of the parameter `is_initial` is `True`, Python will execute the line 16: `head.goto(0, 200)`. The `turtle.goto(x, y)` function moves the turtle to the absolute position of `(x, y)` on the screen.

Finally in line 18, we append the `head` to the `snake` list. Actually it is the first part of our snake.

Score:

To be able to write the score on the screen we need a new turtle object with pen. We will call this turtle as the `pen`. We will create two separate function for score: `create_score` and `update_score`.

- The `create_score` function will instantiate the `pen` turtle and set its color.
- The `update_score` function will update the score text on the screen.

```
[  
7 1 # create the score  
]:  
2  
3 def create_score():  
4  
5     global pen  
6  
7     # create the pen turtle  
8     pen = turtle.Turtle()  
9     pen.penup()  
10    pen.hideturtle()  
11    pen.goto(0, Y_RANGE - 2 * TURTLE_SIZE)  
12    pen.color('black')  
13  
14    # initialize the score  
15    update_score(0)
```

In cell 7, we define the `create_score` function. It has no parameters. It instantiates the global `pen` object as: `pen = turtle.Turtle()`. And hides the turtle object, because we don't want to see the turtle for the score. All we need is its drawings. Then in line 15, we call the `update_score(0)` function to set the score to 0.

```
[  
8 1 # update the score
```

```
[]:  
2  def update_score(score_increment,  
3  is_reset=False):  
4  
5      global score, high_score  
6  
7      if is_reset:  
8          score = 0  
9      else:  
10         score += score_increment  
11  
12      if score > high_score:  
13          high_score = score  
14  
15      pen.clear()  
16  
17      pen.write("Score: {} | High Score:  
18          {}".format(score, high_score),  
19          align='center',  
          font=('Arial', 16, 'normal'))
```

In cell 8 we define the `update_score` function. It takes the `score_increment` and `is_reset` parameters. And it mutates the global `score` and `high_score` variables. If `is_reset` parameter value is `True` then it sets the score to `0` in line 8. If not, it increments the current `score` by `score_increment` in line 10. In line 12, it checks if the current `score` is greater than the `high_score` and it changes the `high_score` if it's `True`.

In line 15, it clears the old text as: `pen.clear()`. And in line 17, it writes the new values of `score` and `high_score`.

Reset:

When the game is over we need to reset all the screen elements to able to start a fresh game. To this, we define a function named **reset**.

```
[ 1  # reset screen fn
]: 2
 3  def reset():
 4
 5  # hide the segments of snake
 6  for t in snake:
 7      t.goto(40000, 4000)
 8
 9  # clear the snake list
10  snake.clear()
11
12  # create a new head
13  create_head(is_initial=False)
14
15  # reset the score
16  update_score(0, is_reset=True)
```

In the **reset** function, we loop over all the parts in the **snake** list in line 6. And we hide all the parts by sending them to a very far **(x, y)** position on the screen. The reason for hiding is that, unfortunately we cannot destroy the turtles we create in the current kernel session. That's why we hide them.

In line 10, we clear the **snake** list. In Python you can remove all the items of a List with the **clear()** function.

In line 13, we create a new head by calling the `create_head()` function. And in line 16, we update the score to 0, which means a new game will start.

Delay:

To be able to see the snake moving, we need some delay. We use the `time` module for this. The `time` module has a method called `sleep()`. The `time.sleep(duration)` function suspends execution of the current program for a given number of seconds. In other words, Python waits for that amount of seconds before executing the next line.

```
[1] 1 # delay function
0]: 2
3 3 def delay(duration):
4     time.sleep(duration)
```

Collisions:

The game will be over if the snake collides either to its own body or to any of the four borders of the screen. Let's define the functions for collisions.

```
[1] 1     # function for border collisions
1]: 2
3 3 def check_border_collisions():
4
5      # if the head position (x, y) is out the ranges
6      # (X_RANGE, Y_RANGE) -> we collide
```

```

7      x = head.xcor()
8      y = head.ycor()
9
10     if x <= -X_RANGE or x >= X_RANGE or y
11     <= -Y_RANGE or y >= Y_RANGE:
12         # set direction
13         global direction
14         direction = 'stop'
15
16         # reset screen after 1 second
17         delay(1)
18         reset()

```

In the **check_border_collisions** function, we get the current **x** and **y** coordinates of the **head** in lines 7 and 8.

In line 10, we check for the borders. If the **x** coordinate is less than the **-X_RANGE** or greater than the **X_RANGE** than this means we have crossed the vertical borders. The same logic applies to the **y** coordinate and the **Y_RANGE** value. So if we collide to any of the borders we set the global **direction** to '**stop**' and delay the screen for 1 second and call the **reset** function.

Now let's see how we handle the body collisions:

```

[1] 1      # body collisions
2
3  def check_body_collisions():
4
5      # if the distance between the head and any of
6      # the segments is less than the TURTLE_SIZE
7      # then this means we collide

```

```

8     for i, t in enumerate(snake):
9         # exclude head
10        if i > 0:
11            if head.distance(t) < TURTLE_SIZE -
12                # set direction
13                global direction
14                direction = 'stop'
15
16                # reset screen after 1 second
17                delay(1)
18                reset()

```

In the `check_body_collisions` function, we check if the distance between the `head` and any of the segments is less than the `TURTLE_SIZE`. If that's `True`, we conclude that the head has been collide to that segment.

Remember, our snake is a list of turtle objects. We loop over every object in it and we exclude the head as: `i > 0`. The `if` statement in line 10, make sure that we pass the first segment, which is the `head`. In line 11, we check the distance between the `head` and the current segment `t`. And if it's less than the `TURTLE_SIZE` then we set the global `direction` to 'stop' and delay the screen for 1 second and call the `reset` function. The segments are also turtle objects which we see later.

Move:

To be able to move the snake, we have to move all of its segments. We will define three separate move functions. The first

one is the main `move()` function, the second one is the `move_head()` and the third one is the `move_segments()`. Let's define them:

```
[1  
3]: 1 # move function  
2  
3 def move():  
4     if window._RUNNING:  
5         # move only if the direction is not stop  
6         if direction != 'stop':  
7             # move the segments  
8             move_segments()  
9  
10        # move the head  
11        move_head()
```

The move function in cell 13, checks if the window is still running as: `window._RUNNING`. If it's running then we can move the snake. The second check is, if the direction is `not 'stop'`. Because if the direction is already '`stop`' we cannot move the snake. To move it, we call the `move_segments()` and the `move_head()` functions respectively. The order of function calls is very important here. You will see why in a minute.

```
[1  
4]: 1 # fn to move segments  
2  
3 def move_segments():  
4  
5     # move each segment in reverse order -> from  
last segment
```

```

6      # move each segment into the position of the
7      # previous one
8      # ignore the head
9      # start from the last one -> len(snake)-1
10     # up to head -> 0
11     # backwards -> -1
12
13     for i in range(len(snake)-1, 0, -1):
14         x = snake[i-1].xcor()
15         y = snake[i-1].ycor()
16         snake[i].goto(x, y)

```

To be able to move the snake we have to start from the very last segment (from the tail). The last one should move to the position of the previous one. And the second one in the last, will move to the position of the previous one. And it goes on like this, up to the first body segment. This is the trick here. You have to move the segments in reverse order. Why? If you move them in the normal order then the next segment will never know where to move, because the position of the previous one will be changed. That's why we move the segments from tail to the head.

Inside the for loop, the current segment is `snake[i]` and the previous one is `snake[i-1]`. We get the `x` coordinate of the previous one as: `x = snake[i-1].xcor()`. And we get the `y` coordinate too. Then in line 15, we move the current segment to `(x, y)` coordinates as: `snake[i].goto(x, y)`.

After we move all the body segments in reverse order, now we can move the `head`. That's why the order of the function call is so

crucial inside the `move()` function.

```
[1] 1  # fn to move the head
[5]: 2
3  def move_head():
4
5      # get current coordinate
6      x = head.xcor()
7      y = head.ycor()
8
9      if direction == 'up':
10         head.sety(y + TURTLE_SIZE)
11     elif direction == 'down':
12         head.sety(y - TURTLE_SIZE)
13     elif direction == 'left':
14         head.setx(x - TURTLE_SIZE)
15     elif direction == 'right':
16         head.setx(x + TURTLE_SIZE)
```

To move the `head`, we first check the current `direction`. Let's say the direction is '`up`'. Then we only increase the `y` coordinate of the `head` by `TURTLE_SIZE` as: `head.sety(y + TURTLE_SIZE)`. Which means the `head` will move up by `TURTLE_SIZE`.

Food:

We need to feed the snake. Each food it eats will become a body segment for it. We will have different functions for the food.

The first one will be `add_food()` which will create a food on the screen. A food is nothing but a turtle object.

The second function will be `move_food()`. It will move the food turtle to a random position on the screen.

The third one will be `eat_food()` function. We will define what ‘**eating**’ means and the actions after eating the food.

Let’s define these functions:

```
[1] 1  # create the food object
[6]: 2
3  def add_food():
4
5  if window._RUNNING:
6      global food
7
8      # create a turtle -> single -> Singleton
9      Pattern
10     if food == None:
11         food = turtle.Turtle()
12         food.shape(get_shape())
13         food.shapesize(0.5, 0.5)
14         food.speed(0)
15         food.penup()
16
17         # color
18         food.color(get_color())
19
20         # move the food
21         move_food(food)
```

In the `add_food` function, we first check if the `window` is still `running`. Then in line 9, we check if the global `food` variable is `None`. We create a new food, only if there is no food on the

screen. Then we create a new Turtle object and assign it to the global **food** variable.

In line 11, we give a random shape to the **food**. Instead of defining the shape here, we call another function **get_shape()**. The **get_shape()** function returns a random shape. We will define it later.

In lines 12, 13 and 14, we set the size, speed and pen of this new **food** object. Then in line 17, we call another function: **get_color()**. The **get_color()** function returns a random color and we set this color to the **food**. We will define the **get_color()** function later.

Finally in line 20, we call the **move_food()** function and pass this new **food** as the argument. The **move_food()** function is responsible for moving the food.

```
[1] 1  # function to move the food
[7]: 2
3  def move_food(food):
4      # x coordinate
5      x = random.randint(-X_RANGE, X_RANGE)
6
7      # y coordinate
8      y = random.randint(-Y_RANGE, Y_RANGE - 2
9      * TURTLE_SIZE)
10     # replace the food
11     food.goto(x, y)
```

The **move_food()** function, first generates a set of random x and y coordinates. It uses the **random.randint()** function for this. The **randint(a, b)** function returns a random integer **N** such that **a <= N <= b**. And we pass the limits to this function.

For the **x** coordinate it is just the left and the right borders of the screen : **x = random.randint(-X_RANGE, X_RANGE)**.

For the **y** coordinate, since we will have the score area at the top of the screen we pass the arguments as: **random.randint(-Y_RANGE, Y_RANGE - 2 * TURTLE_SIZE)**.

After getting a random pair of **(x, y)** coordinates on the screen we move the **food** to that position as: **food.goto(x, y)**.

```
[1
8]: 1      # function to eat the food
      2
      3  def eat_food():
      4
      5      # check the distance between the head and the
      6      # food
      7      if head.distance(food) < TURTLE_SIZE - 1:
      8          # move the food
      9          move_food(food)
     10
     11          # change the food shape
     12          food.shape(get_shape())
     13
     14          # create a segment for the snake
     15          create_segment()
     16
     17          # change the food color
```

```
18     food.color(get_color())
19
20     # update score
21     update_score(10)
```

In cell 18, we define the `eat_food()` function. The snake eats the food, when the distance between the head and the food is less than the `TURTLE_SIZE`. That's the condition for eating and you see it in line 6. When the snake eats the food, here are the actions we take:

- in line 9, move the food to a new position:
`move_food(food)`
- in line 12, change the shape of the food:
`food.shape(get_shape())`
- in line 15, create a new segment for the snake:
`create_segment()`
- in line 18, change the food color:
`food.color(get_color())`
- in line 21, update the score: `update_score(10)`

Body Segments:

Now that the snake eats the food, let's define functions to add body segments to it. We will have two separate functions: `create_segment` and `get_last_segment_position`.

```
[1
9]: 1     # function to create segment
      2
```

```
3  def create_segment():
4      """Creates a new segment for snake."""
5
6      global snake
7
8      # create a segment
9      segment = turtle.Turtle()
10     segment.shape(shapes[1])
11     segment.speed(0)
12     segment.color(food_color)
13     segment.penup()
14
15     # position the segment
16     x, y = get_last_segment_position()
17     segment.goto(x, y)
18
19     # add this segment into snake list
20     snake.append(segment)
```

In the `create_segment` function, we create a new turtle object and name it as segment. We set its shape and speed. In line 12, we set its color as the same color with the food. Because the segment comes from the food.

In line 16, we get the position of the last segment. We call the `get_last_segment_position` function for this. And it returns the `x` and `y` coordinates of the last segment of the snake. We need these coordinates because, we will place the new segment at that position. And that's what we do in line 17 as: `segment.goto(x, y)`.

In line 20, we append this new segment to the snake list as:
snake.append(segment).

```
[2
0]: 1 # last segment position
2
3 def get_last_segment_position():
4
5     # last element -> snake[-1]
6     x = snake[-1].xcor()
7     y = snake[-1].ycor()
8
9     # direction
10    # if direction is up -> same x, y is
11    # TURTLE_SIZE less
12    if direction == 'up':
13        y = y - TURTLE_SIZE
14
15    # if direction is up -> same x, y is
16    # TURTLE_SIZE more
17    elif direction == 'down':
18        y = y + TURTLE_SIZE
19
20    # if direction is right -> same y, x is
21    # TURTLE_SIZE less
22    elif direction == 'right':
23        x = x - TURTLE_SIZE
24
25    # if direction is left -> same y, x is
26    # TURTLE_SIZE more
27    elif direction == 'left':
28        x = x + TURTLE_SIZE
29
30    return (x, y)
```

In cell 20, we define the `get_last_segment_position` function. The last segment of the snake is the last element in the list. We get it as: `snake[-1]`. We get its `x` and `y` coordinates in lines 6 and 7. Then we check the current direction. We need to know, in which direction the snake is currently moving.

Let's say, it's moving to the '`right`'. This is the condition in line 19. Then the `y` coordinate of the new segment will be the same as the last one. That's why we do not touch the `y` coordinate. But the `x` coordinate will be `x - TURTLE_SIZE`. Because the new segment will come after the last one. That's why we have a minus sign for the '`right`' direction.

In line 26, the function returns a tuple of the new values of `(x, y)`.

Shape:

Every time the snake eats the food, we want the food to be of a new random shape. Remember, we have the global variable `shapes`. It is a dictionary and it keeps the numbers and corresponding shapes. We define a function to return a random shape each time we call it.

```
[2
1]: 1 # get a random shape
      2
      3 def get_shape():
      4     # get a random index from the shapes
      5     index = random.randint(0, len(shapes)-1)
      6     # return the item at that index
```

```
7     return shapes[index]
```

The function creates a random **index** at line 5 and returns the element at that index from the **shapes** dictionary.

Color:

```
[2
2]: 1     # get a random color
      2
      3     def get_color():
      4
      5         global food_color
      6
      7         index = random.randint(0, len(colors)-1)
      8         color = colors[index]
      9
     10        food_color = color
     11
     12        return color
```

Every time the snake eats the food, we want the food to be of a new random color. And we define a function for it. The **get_color()** function creates a random **index** by using the length of the **colors** dictionary as: **index = random.randint(0, len(colors)-1)**. And it gets the color at this index as: **color = colors[index]**.

In line 10, it assigns this **color** to the global **food_color** because the new segment will be using this global variable. Finally it returns the **color**.

Screen:

We need to update the screen, because we turned off automatic screen updates. Let's define the `update_screen()` function:

```
[2
3]: 1  # function to update screen
2
3  def update_screen():
4
5      while window._RUNNING:
6
7          # side collisions
8          check_border_collisions()
9
10         # body collisions
11         check_body_collisions()
12
13         # move the head
14         move()
15
16         # delay
17         delay(delay_time)
18
19         # create the food
20         add_food()
21
22         # eat the food
23         eat_food()
24
25         # get rid of update error
26         window.update()
```

The `update_screen()` function first checks if the window is still running. Then it starts to call other functions. Here are these

functions:

- in line 8, it checks border collisions: `check_border_collisions()`
- in line 11, it checks border collisions: `check_body_collisions()`
- in line 14, it moves the snake: `move()`
- in line 17, it calls the delay function: `delay(delay_time)`
- in line 20, it adds the food: `add_food()`
- in line 23, it calls eat_food: `eat_food()`
- in line 26, it calls the built in `update()` method on the `window` object: `window.update()`

Since each function knows exactly what to do, we only need to call them. That's all. Be careful that, the `update_screen()` function will call these functions as far as the screen is running. Because this is the loop condition for the `while` loop in line 5. You can think of it as an infinite loop, if the screen keeps running. And it will wait until `delay_time` seconds at each iteration (line 17).

Main:

Now that we have all the functions defined, we need one final function to call them. This is going to be the `main()` function. In general, `main()` functions are used to start the program execution. That's what we will do here. We will call the necessary functions

inside our `main()` function. And as soon as we call the `main()` function, our snake game will start.

```
[2  
4]: 1 # main function to call other functions  
2  
3 def main():  
4  
5     # set the screen  
6     set_screen()  
7  
8     # listen keyboard events  
9     listen_events()  
10  
11    # create head  
12    create_head()  
13  
14    # create score  
15    create_score()  
16  
17    # update screen  
18    update_screen()
```

In cell 24, we define the main function. Its purpose is to call the necessary function to set and start the game. It calls `set_screen()`, `listen_events()`, `create_head()`, `create_score()` and `update_screen()` functions in this order. Now let's call the `main()` function and play the game.

```
[2  
5]: 1 # call the main function to start the game  
2 main()
```

As soon as we run the cell 25, and call the `main()` function Python will start the game. You can start playing it now. To start a new game just press any of the direction buttons on the keyboard.

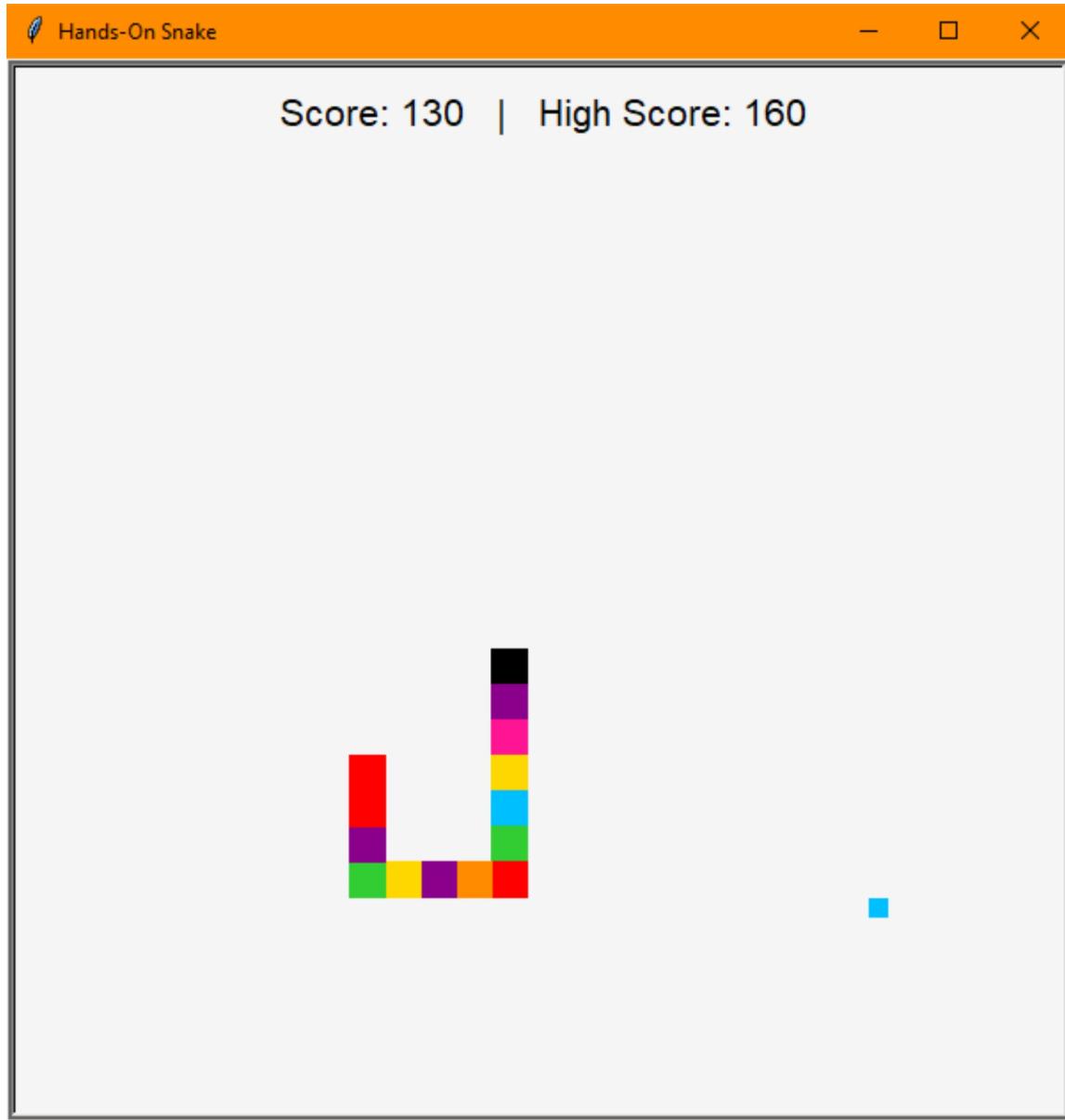


Figure 19-2: The snake game completed and ready to play

Final Warning:

Since our game is built on the turtle module, the last line should always be `turtle.mainloop()`. You will start playing the game by calling the `main()` function. When you decide to close the game, run this line of code. You should only run it before you close the game. You shouldn't run it while playing the game. It's the terminator line. That's why, it is the last cell in our project:

```
[2
6]: 1 # the last line
      2 turtle.mainloop()
```

OceanofPDF.com

20. Assignment 3 - Snake Game

We finished our third project, Project 3 - Snake Game. Now it's your turn to recreate the same project. This chapter is the assignment on the Project 3 - Snake Game.

To complete the assignment, follow the instructions where you see a **#TODO** statement. You can also download the assignment Jupyter file in the [Github repository](#) of the book. You can find the solutions in the previous chapter.

The Assignment

You will create your own Snake Game with Python. You can play it once you finish it. It will be a fun way of using what you learned in this book. You will use built-in [turtle](#), [time](#) and [random](#) modules in this assignment. Here is an image of the snake game that you will build:

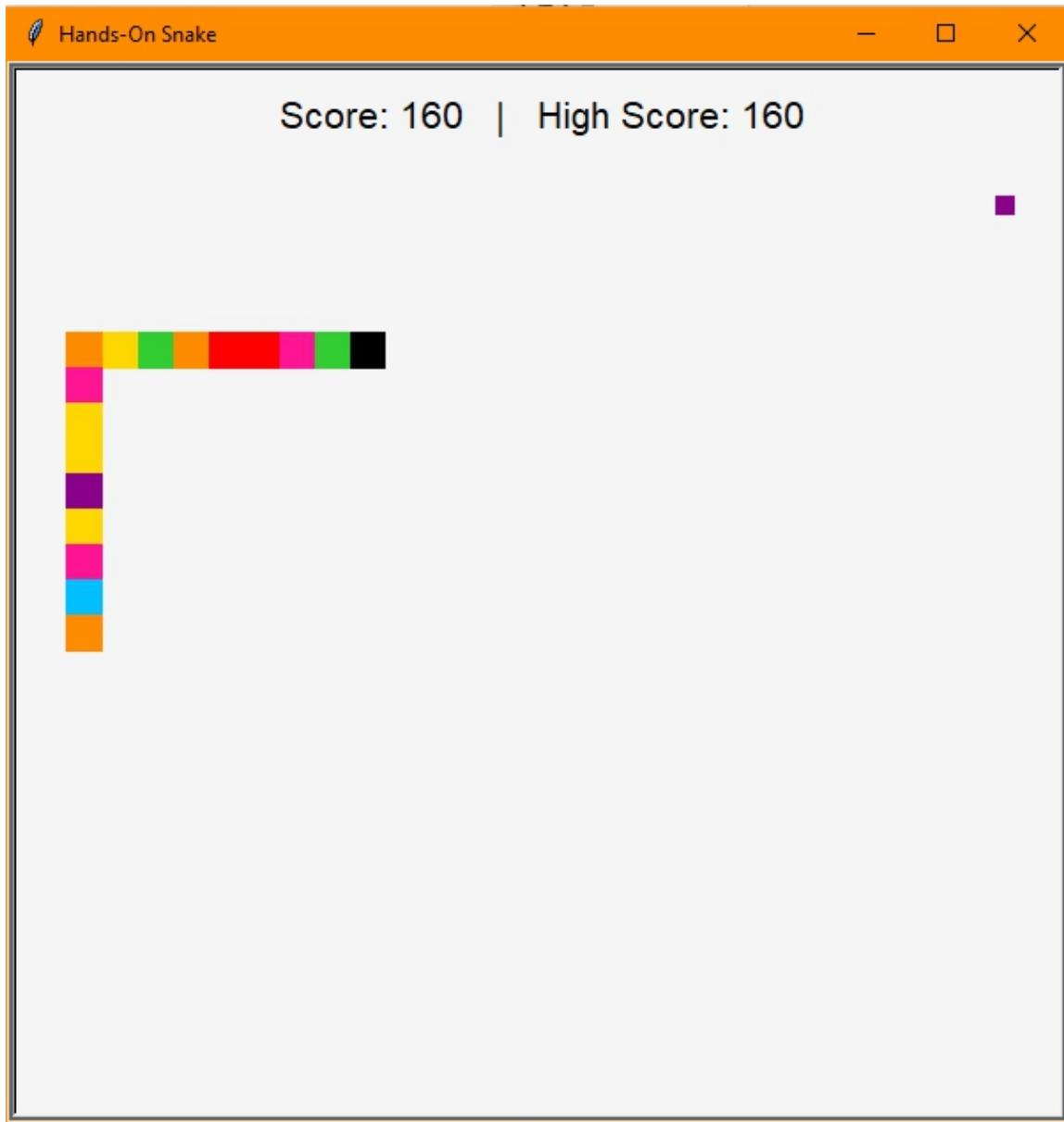


Figure 20-1: The snake game

Let's start by importing the modules:

```
[1]: #TODO - import the packages: turtle, time and
     random
```

Global Variables:

Next, we will define the global variables, which we will be using throughout the project:

```
[  
2  1  # global variable  
]:  
2  
3  window = None  
4  snake = []  
5  head = None  
6  
7  direction = 'down'  
8  delay_time = 0.1  
9  
10 food = None  
11 food_color = ''  
12  
13 TURTLE_SIZE = 20  
14 WIDTH = 600  
15 HEIGHT = 600  
16  
17 X_RANGE = (WIDTH - TURTLE_SIZE) / 2  
18 Y_RANGE = (HEIGHT - TURTLE_SIZE) / 2  
19  
20 pen = None  
21 score = 0  
22 high_score = 0  
23  
24 #TODO - define shapes dictionary  
25  
26 #TODO - define colors dictionary
```

Here are the explanations for the global variables:

window: The main window object of the project. We instantiate it as **None**. We will assign **turtle.Screen** object to it later on.

snake: It is a list to store the head and the body parts of the snake.

head: It is an object to store the head of the snake.

direction: For keeping the direction of the snake. The default direction is down.

delay_time: The time interval in seconds to wait for delay.

food: The food object.

food_color: Color of the food.

TURTLE_SIZE: The size of our turtle in pixels.

WIDTH: Width of the screen.

HEIGHT: Height of the screen.

X_RANGE: Roughly the half of the screen in x direction.

Y_RANGE: Roughly the half of the screen in y direction.

pen: A turtle object to be able to write the score on the screen.

score: Current score.

high_score: Highest score.

shapes: the dictionary for the shapes of the foods.

colors: the dictionary for the colors we use.

Functions

Main Screen:

Now let's define a function, `set_screen`, to set up the main screen. It will use the global variable `window`, and assign `turtle.Screen` object to it. It will also set the title, background color, with and height of the screen.

```
[  
3 1  # function to set up the main screen  
]:  
2  
3 def set_screen():  
4     """Sets the main screen."""  
5  
6     global window  
7  
8     #TODO - create the window (Screen)  
9     # and give it;  
10    # title, background, width-height,  
11    # and set the tracer.
```

The `window.tracer(0)` method in line 12 is used to turn turtle animation on or off and set a delay for update drawings. We don't want to have automatic screen updates, so we turn them off by setting `tracer()` to 0. 0 means False. We will update the screen manually.

Events:

We will play our snake game with the keyboard. So we need to listen the keyboard events. The keyboard event which we are interested in is: `onkeypress(fun, key)`. This function takes two arguments:

- **fun**: the name of the function to execute when this event occurs
- **key**: the string name of the key (e.g. “Down”)

```
[  
4 1 # function to listen screen events  
]:  
2  
3 def listen_events():  
4     window.listen()  
5     #TODO - listen Up, Down, Left and Right key  
press events
```

In the `listen_events` function we first call the `window.listen()` function to tell Python that we are going to listen the window events.

Now let's define the functions which we pass as arguments to the `window.onkeypress()` function:

```
[  
5 1 # keyboard functions  
]:  
2  
3 def set_up_direction():  
4     #TODO - set the global direction to up  
5  
6 def set_down_direction():  
7     #TODO - set the global direction to down  
8  
9 def set_left_direction():  
10    #TODO - set the global direction to left  
11  
12 def set_right_direction():
```

```
13     global direction
14     if direction != 'left':
15         #TODO - set the global direction to right
```

In cell 5, we define our window **event handlers**. Event handler means the function which gets called when an event occurs. We have four event handlers for four keyboard directions. Each of them modifies the global **direction** variable.

Let's say the user presses the '**up**' key in the keyboard. As soon as the user presses the keyboard, **window.onkeypress(set_up_direction, 'Up')** event will get triggered. And this event will call our **set_up_direction** function.

Inside **set_up_direction** function we first tell Python that we will be using the **global direction** variable in line 4. Then we check if the current direction is '**not down**'. Why? Because the snake will only move up, if it's currently **not moving down**. We don't want to the snake to get the opposite direction instantaneously. That's the main idea for all of the keyboard functions.

Head:

Now let's define a function to create the head of the snake.

```
[ 6
]: 1     # create the head
     2
     3     def create_head(is_initial=True):
           """Creates the snake head."""
```

```
5
6     global head, snake
7
8     # create the head
9
10    #TODO - create the head turtle, give it a shape
11    and a color.
12    #TODO - set the position for the head
13    #TODO - append the head into snake list
```

The `create_head` function takes one argument as `is_initial`. It's a bool variable to decide if the head should move to the starting position or not.

Inside the `create_head` function we modify the global `head` and `snake` variables. In line 9, it creates a Turtle object as `turtle.Turtle()` and assigns it to the global `head` variable. Then we modify the shape and speed of the head. We don't want the `head` to draw any shapes when moving, so we call `head.penup()` function to pull the pen up.

If the value of the parameter `is_initial` is `True`, Python will execute the line 16: `head.goto(0, 200)`. The `turtle.goto(x, y)` function moves the turtle to the absolute position of `(x, y)` on the screen.

Finally in line 18, we append the `head` to the `snake` list. Actually it is the first part of our snake.

Score:

To be able to write the score on the screen we need a new turtle object with pen. We will call this turtle as the `pen`. We will create two separate function for score: `create_score` and `update_score`.

- The `create_score` function will instantiate the `pen` turtle and set its color.
- The `update_score` function will update the score text on the screen.

```
[  
7 1 # create the score  
]:  
2  
3 def create_score():  
4  
5     global pen  
6  
7     # create the pen turtle  
8     #TODO - create the pen turtle and place it on  
9     screen  
10    # initialize the score  
11    #TODO - call the update_score function to  
12    initialize the pen
```

In cell 7, we define the `create_score` function. It has no parameters. It instantiates the global `pen` object as: `pen = turtle.Turtle()`. And hides the turtle object, because we don't want to see the turtle for the score. All we need is its drawings.

Then in line 15, we call the `update_score(0)` function to set the score to 0.

```
[  
8  1      # update the score  
]:  
2  
3  def update_score(score_increment, is_reset=False):  
4  
5      #TODO - update the global score and  
6      # high_score variables, based on is_reset  
7  
8      #TODO - check if the score is greater than the  
9      # high_score  
10  
11     pen.write("Score: {0} | High Score:  
12     {1}".format(score, high_score),  
13     align='center',  
14     font=('Arial', 16, 'normal'))
```

In cell 8 we define the `update_score` function. It takes the `score_increment` and `is_reset` parameters. And it mutates the global `score` and `high_score` variables. If `is_reset` parameter value is `True` then it sets the score to `0` in line 8. If not, it increments the current `score` by `score_increment` in line 10. In line 12, it checks if the current `score` is greater than the `high_score` and it changes the `high_score` if it's `True`.

In line 15, it clears the old text as: `pen.clear()`. And in line 17, it writes the new values of `score` and `high_score`.

Reset:

When the game is over we need to reset all the screen elements to able to start a fresh game. To this, we define a function named **reset**.

```
[9] : 1 # reset screen fn
      2
      3 def reset():
      4
      5     # hide the segments of snake
      6     for t in snake:
      7         t.goto(40000, 4000)
      8
      9     # clear the snake list
     10    snake.clear()
     11
     12    # create a new head
     13    #TODO - call create_head function with
     14    # is_initial parameter being False
     15
     16    # reset the score
     17    #TODO - call update_score function with
     18    # score_increment as 0 and is_reset as True
```

In the **reset** function, we loop over all the parts in the **snake** list in line 6. And we hide all the parts by sending them to a very far (x, y) position on the screen. The reason for hiding is that, unfortunately we cannot destroy the turtles we create in the current kernel session. That's why we hide them.

In line 10, we clear the `snake` list. In Python you can remove all the items of a List with the `clear()` function.

In line 13, we create a new head by calling the `create_head()` function. And in line 16, we update the score to 0, which means a new game will start.

Delay:

To be able to see the snake moving, we need some delay. We use the `time` module for this. The time module has a method called `sleep()`. The `time.sleep(duration)` function suspends execution of the current program for a given number of seconds. In other words, Python waits for that amount of seconds before executing the next line.

```
[1  # delay function
0]: 1
      2
      3 def delay(duration):
      4     time.sleep(duration)
```

Collisions:

The game will be over if the snake collides either to its own body or to any of the four borders of the screen. Let's define the functions for collisions.

```
[1  # function for border collisions
1]: 1
      2
      3 def check_border_collisions():
      4
```

```

5      # if the head position (x, y) is out the ranges
6      (X_RANGE, Y_RANGE) -> we collide
7
8      #TODO - get x and y coordinates of the head
9      turtle
10
11     if #TODO decide the collision :
12
13         # set direction
14         #TODO - set the global direction variable to
15         'stop'
16
17         # reset screen after 1 second
18         #TODO - call delay function with 1 seconds
19         #TODO - call reset function

```

In the **check_border_collisions** function, we get the current **x** and **y** coordinates of the **head** in lines 7 and 8.

In line 10, we check for the borders. If the **x** coordinate is less than the **-X_RANGE** or greater than the **X_RANGE** than this means we have crossed the vertical borders. The same logic applies to the **y** coordinate and the **Y_RANGE** value. So if we collide to any of the borders we set the global **direction** to **'stop'** and delay the screen for 1 second and call the **reset** function.

Now let's see how we handle the body collisions:

```

[1] 1      # body collisions
[2]: 2
3      def check_body_collisions():
4

```

```

5      # if the distance between the head and any of
6      # the segments is less than the TURTLE_SIZE
7      # then this means we collide
8
9      for ...#TODO - get all turtles and indices for
10     # the snake...:
11
12     # exclude head
13     if #TODO - exclude head index:
14
15         # if ...#TODO - get the distance between the
16         # head and the current turtle in the loop... <
17         # TURTLE_SIZE - 1:
18
19         # set direction
20         #TODO - set the global direction
21         # variable to 'stop'
22
23         # reset screen after 1 second
24         #TODO - call delay function with 1
25         # seconds
26         #TODO - call reset function

```

In the `check_body_collisions` function, we check if the distance between the `head` and any of the segments is less than the `TURTLE_SIZE`. If that's `True`, we conclude that the head has been collide to that segment.

Remember, our snake is a list of turtle objects. We loop over every object in it and we exclude the head as: `i > 0`. The `if` statement in line 10, make sure that we pass the first segment, which is the `head`. In line 11, we check the distance between the `head` and the current segment `t`. And if it's less than the

`TURTLE_SIZE` then we set the global `direction` to ‘`stop`’ and delay the screen for 1 second and call the `reset` function. The segments are also turtle objects which we see later.

Move:

To be able to move the snake, we have to move all of its segments. We will define three separate move functions. The first one is the main `move()` function, the second one is the `move_head()` and the third one is the `move_segments()`. Let’s define them:

```
[1
3]: 1      # move function
      2
      3  def move():
      4      if window._RUNNING:
      5          # move only if the direction is not stop
      6          if direction != 'stop':
      7              # move the segments
      8              #TODO - call move_segments function
      9
      10         # move the head
      11         #TODO - call move_head function
```

The move function in cell 13, checks if the window is still running as: `window._RUNNING`. If it’s running then we can move the snake. The second check is, if the direction is **not** ‘`stop`’. Because if the direction is already ‘`stop`’ we cannot move the snake. To move it, we call the `move_segments()` and the

`move_head()` functions respectively. The order of function calls is very important here. You will see why in a minute.

```
[1
4]: 1  # fn to move segments
2
3  def move_segments():
4
5      # move each segment in reverse order -> from
6      # last segment
7      # move each segment into the position of the
8      # previous one
9      # ignore the head
10     # start from the last one -> len(snake)-1
11     # up to head -> 0
12     # backwards -> -1
13
14     for i in range(len(snake)-1, 0, -1):
15         #TODO - get the x and y coordinate of the
16         # previous segment
17         #TODO - place the current turtle in the
18         # loop at x and y
```

To be able to move the snake we have to start from the very last segment (from the tail). The last one should move to the position of the previous one. And the second one in the last, will move to the position of the previous one. And it goes on like this, up to the first body segment. This is the trick here. You have to move the segments in reverse order. Why? If you move them in the normal order then the next segment will never know where to

move, because the position of the previous one will be changed. That's why we move the segments from tail to the head.

Inside the for loop, the current segment is `snake[i]` and the previous one is `snake[i-1]`. We get the `x` coordinate of the previous one as: `x = snake[i-1].xcor()`. And we get the `y` coordinate too. Then in line 15, we move the current segment to `(x, y)` coordinates as: `snake[i].goto(x, y)`.

After we move all the body segments in reverse order, now we can move the `head`. That's why the order of the function call is so crucial inside the `move()` function.

```
[1
5]: 1  # fn to move the head
2
3  def move_head():
4
5      # get current coordinate
6      #TODO - get x and y coordinates of the head
7      turtle
8
9      if direction == 'up':
10          head.sety(y + TURTLE_SIZE)
11      elif direction == 'down':
12          #TODO - set the y coordinate of the head
13          #appropriately -> remember the turtle moves
14          #TURTLE_SIZE pixels
15      elif direction == 'left':
16          #TODO - set the x coordinate of the head
17          #appropriately -> remember the turtle moves
18          #TURTLE_SIZE pixels
19      elif direction == 'right':
```

15

*#TODO - set the x coordinate of the head
appropriately -> remember the turtle moves
TURTLE_SIZE pixels*

To move the **head**, we first check the current **direction**. Let's say the direction is 'up'. Then we only increase the y coordinate of the **head** by **TURTLE_SIZE** as: **head.sety(y + TURTLE_SIZE)**. Which means the **head** will move up by **TURTLE_SIZE**.

Food:

We need to feed the snake. Each food it eats will become a body segment for it. We will have different functions for the food.

The first one will be **add_food()** which will create a food on the screen. A food is nothing but a turtle object.

The second function will be **move_food()**. It will move the food turtle to a random position on the screen.

The third one will be **eat_food()** function. We will define what 'eating' means and the actions after eating the food.

Let's define these functions:

[1
6]:

```
1  # create the food object
2
3  def add_food():
4
5      if window._RUNNING:
6          global food
7
```

```

8      # create a turtle -> single -> Singleton
9      Pattern
10     if food == None:
11         #TODO - create the food the turtle and
12         # give it a random shape
13         # color
14         #TODO - give food turtle a random color
15
16         # move the food
17         #TODO - call move_food function with
              the food turtle

```

In the **add_food** function, we first check if the **window** is still **running**. Then in line 9, we check if the global **food** variable is **None**. We create a new food, only if there is no food on the screen. Then we create a new Turtle object and assign it to the global **food** variable.

In line 11, we give a random shape to the **food**. Instead of defining the shape here, we call another function **get_shape()**. The **get_shape()** function returns a random shape. We will define it later.

In lines 12, 13 and 14, we set the size, speed and pen of this new **food** object. Then in line 17, we call another function: **get_color()**. The **get_color()** function returns a random color and we set this color to the **food**. We will define the **get_color()** function later.

Finally in line 20, we call the `move_food()` function and pass this new `food` as the argument. The `move_food()` function is responsible for moving the food.

```
[1
7]: 1  # function to move the food
2
3  def move_food(food):
4      # x coordinate
5      x = #TODO - get a random integer between -
6      # X_RANGE and X_RANGE
7
8      # y coordinate
9      y = #TODO - get a random integer between -
10     # Y_RANGE and (Y_RANGE - 2 * TURTLE_SIZE)
11
12     # replace the food
13     food.goto(x, y)
```

The `move_food()` function, first generates a set of random x and y coordinates. It uses the `random.randint()` function for this. The `randint(a, b)` function returns a random integer `N` such that `a <= N <= b`. And we pass the limits to this function.

For the `x` coordinate it is just the left and the right borders of the screen : `x = random.randint(-X_RANGE, X_RANGE)`.

For the `y` coordinate, since we will have the score area at the top of the screen we pass the arguments as: `random.randint(-Y_RANGE, Y_RANGE - 2 * TURTLE_SIZE)`.

After getting a random pair of `(x, y)` coordinates on the screen we move the `food` to that position as: `food.goto(x, y)`.

```
[1
8]: 1 # function to eat the food
2
3 def eat_food():
4
5     # check the distance between the head and the
6     # food
7     if head.distance(food) < TURTLE_SIZE - 1:
8
9         # move the food
10        #TODO - call the move_food function with
11        # the food
12
13        # change the food shape
14        #TODO - change the food shape to a
15        # random one
16
17        # create a segment for the snake
18        #TODO - call create_segment function
19
20        # change the food color
21        #TODO - give food turtle a random color
22
23        # update score
24        #TODO - call update_score function with 10
25        # as increment
```

In cell 18, we define the `eat_food()` function. The snake eats the food, when the distance between the head and the food is less than the `TURTLE_SIZE`. That's the condition for eating and you see it in line 6. When the snake eats the food, here are the actions we take:

- in line 9, move the food to a new position:
move_food(food)
- in line 12, change the shape of the food:
food.shape(get_shape())
- in line 15, create a new segment for the snake:
create_segment()
- in line 18, change the food color:
food.color(get_color())
- in line 21, update the score: **update_score(10)**

Body Segments:

Now that the snake eats the food, let's define functions to add body segments to it. We will have two separate functions: **create_segment** and **get_last_segment_position**.

```
[1
9]: 1  # function to create segment
2
3  def create_segment():
4      """Creates a new segment for snake."""
5
6      global snake
7
8      # create a segment
9      #TODO - create the segment turtle with
10     #appropriate shape
11
12     #TODO - set the color of the segment turtle to
13     #global food_color
```

```
13     # position the segment
14     x, y = #TODO - call get_last_segment_position
15     function
16
17     segment.goto(x, y)
18
19     # add this segment into snake list
20     #TODO - append the segment to global snake
21     list
```

In the `create_segment` function, we create a new turtle object and name it as `segment`. We set its shape and speed. In line 12, we set its color as the same color with the food. Because the segment comes from the food.

In line 16, we get the position of the last segment. We call the `get_last_segment_position` function for this. And it returns the `x` and `y` coordinates of the last segment of the snake. We need these coordinates because, we will place the new segment at that position. And that's what we do in line 17 as: `segment.goto(x, y)`.

In line 20, we append this new segment to the snake list as: `snake.append(segment)`.

```
[2
0]: 1     # last segment position
2
3     def get_last_segment_position():
4
5     # last element -> snake[-1]
6     #TODO - get the x and y coordinates of the last
7     segment in the snake
```

```

8      # direction
9      # if direction is up -> same x, y is
10     TURTLE_SIZE less
11     if direction == 'up':
12         y = y - TURTLE_SIZE
13
14     # if direction is up -> same x, y is
15     TURTLE_SIZE more
16     elif direction == 'down':
17         y = #TODO - assign the new y value
18
19     # if direction is right -> same y, x is
20     TURTLE_SIZE less
21     elif direction == 'right':
22         x = #TODO - assign the new x value
23
24     # if direction is left -> same y, x is
25     TURTLE_SIZE more
26     elif direction == 'left':
27         x = #TODO - assign the new x value
28
29     return #TODO - return a tuple of x and y

```

In cell 20, we define the `get_last_segment_position` function. The last segment of the snake is the last element in the list. We get it as: `snake[-1]`. We get its `x` and `y` coordinates in lines 6 and 7. Then we check the current direction. We need to know, in which direction the snake is currently moving.

Let's say, it's moving to the '`right`'. This is the condition in line 19. Then the `y` coordinate of the new segment will be the same as the last one. That's why we do not touch the `y` coordinate. But the `x` coordinate will be `x - TURTLE_SIZE`. Because the new

segment will come after the last one. That's why we have a minus sign for the 'right' direction.

In line 26, the function returns a tuple of the new values of (x, y).

Shape:

Every time the snake eats the food, we want the food to be of a new random shape. Remember, we have the global variable **shapes**. It is a dictionary and it keeps the numbers and corresponding shapes. We define a function to return a random shape each time we call it.

```
[2
1]: 1 # get a random shape
      2
      3 def get_shape():
      4     # get a random index from the shapes
      5     index = random.randint(0, len(shapes)-1)
      6     # return the item at that index
      7     return shapes[index]
```

The function creates a random **index** at line 5 and returns the element at that index from the **shapes** dictionary.

Color:

```
[2
2]: 1 # get a random color
      2
      3 def get_color():
      4
      5     global food_color
```

```
6
7     index = #TODO - get a random integer between
8     0 and the length of colors -1
9
10    color = colors[index]
11
12    food_color = color
13
14    return color
```

Every time the snake eats the food, we want the food to be of a new random color. And we define a function for it. The `get_color()` function creates a random `index` by using the length of the `colors` dictionary as: `index = random.randint(0, len(colors)-1)`. And it gets the color at this index as: `color = colors[index]`.

In line 10, it assigns this `color` to the global `food_color` because the new segment will be using this global variable. Finally it returns the `color`.

Screen:

We need to update the screen, because we turned off automatic screen updates. Let's define the `update_screen()` function:

```
[2
3]: 1     # function to update screen
2
3     def update_screen():
4
5         while window._RUNNING:
6
7             # side collisions
```

```

8      #TODO - call check_border_collisions
9      function
10     # body collisions
11     #TODO - call check_body_collisions
12     function
13     # move the head
14     #TODO - call move function
15
16     # delay
17     #TODO - call delay function with global
18     delay_time
19
20     # create the food
21     #TODO - call add_food function
22
23     # eat the food
24     #TODO - call eat_food function
25
26     # get rid of update error
      window.update()

```

The `update_screen()` function first checks if the window is still running. Then it starts to call other functions. Here are these functions:

- in line 8, it checks border collisions:
`check_border_collisions()`
- in line 11, it checks border collisions:
`check_body_collisions()`
- in line 14, it moves the snake: `move()`

- in line 17, it calls the `delay` function: `delay(delay_time)`
- in line 20, it adds the food: `add_food()`
- in line 23, it calls `eat_food`: `eat_food()`
- in line 26, it calls the built in `update()` method on the `window` object: `window.update()`

Since each function knows exactly what to do, we only need to call them. That's all. Be careful that, the `update_screen()` function will call these functions as far as the screen is running. Because this is the loop condition for the `while` loop in line 5. You can think of it as an infinite loop, if the screen keeps running. And it will wait until `delay_time` seconds at each iteration (line 17).

Main:

Now that we have all the functions defined, we need one final function to call them. This is going to be the `main()` function. In general, `main()` functions are used to start the program execution. That's what we will do here. We will call the necessary functions inside our `main()` function. And as soon as we call the `main()` function, our snake game will start.

```
[2
4]: 1  # main function to call other functions
      2
      3  def main():
      4
      5      # set the screen
```

```
6     set_screen()
7
8     # listen keyboard events
9     listen_events()
10
11    # create head
12    create_head()
13
14    # create score
15    create_score()
16
17    # update screen
18    update_screen()
```

In cell 24, we define the main function. Its purpose is to call the necessary function to set and start the game. It calls `set_screen()`, `listen_events()`, `create_head()`, `create_score()` and `update_screen()` functions in this order. Now let's call the `main()` function and play the game.

```
[2
5]: 1 # call the main function to start the game
     2 main()
```

As soon as we run the cell 25, and call the `main()` function Python will start the game. You can start playing it now. To start a new game just press any of the direction buttons on the keyboard.

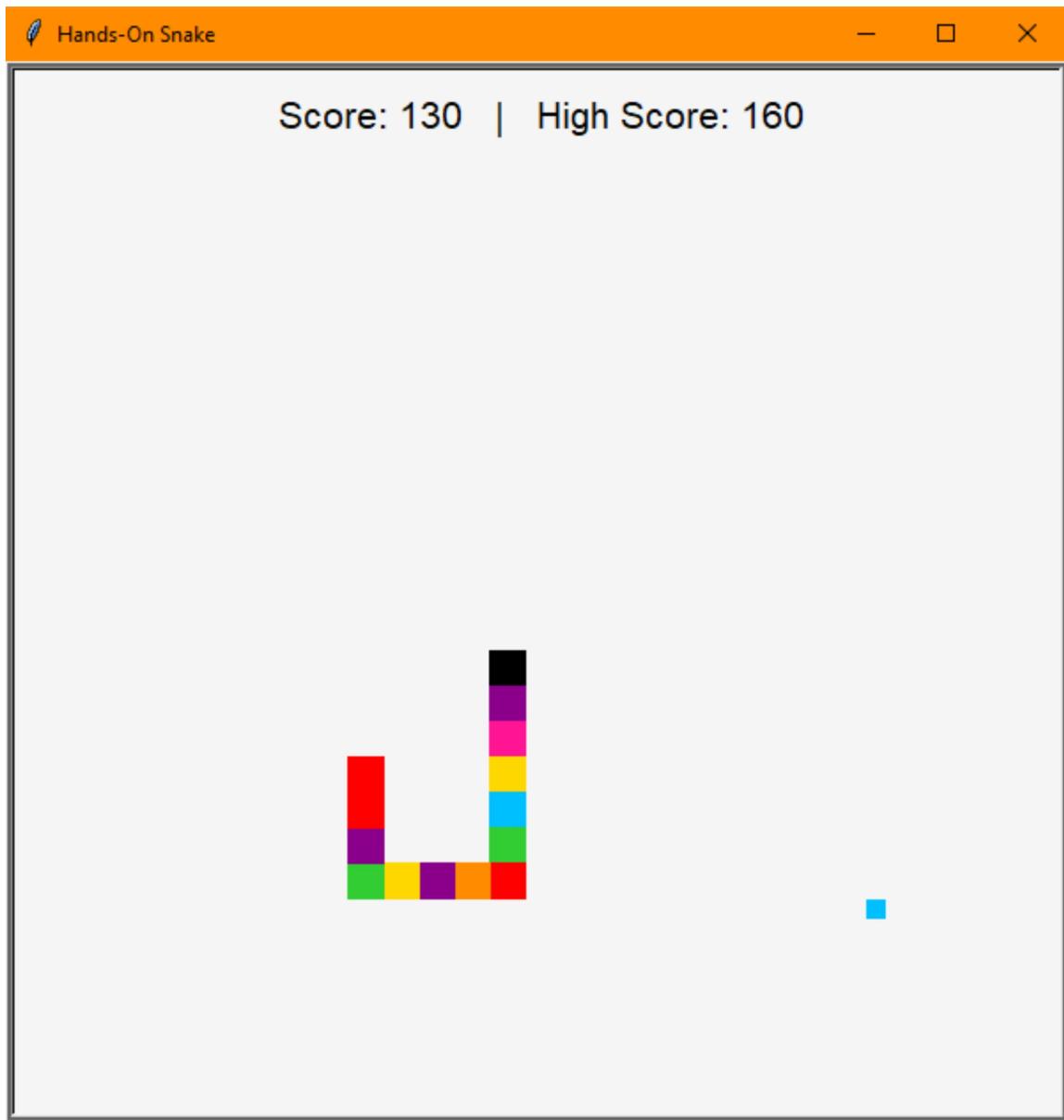


Figure 20-2: The snake game completed and ready to play

Final Warning:

Since our game is built on the turtle module, the last line should always be `turtle.mainloop()`. You will start playing the game by calling the `main()` function. When you decide to close the game, run this line of code. You should only run it before you

close the game. You shouldn't run it while playing the game. It's the terminator line. That's why, it is the last cell in our project:

```
[2  
6]: 1 # the last line  
2 turtle.mainloop()
```

OceanofPDF.com

21. Final Exam

The Final Exam includes the topics we have covered in this book. It is a multiple choice exam with 20 questions. Each question is 5 points, which makes 100 points in total. You should get at least 70 points to be successful. The duration is 120 minutes. Here is the number of questions in the respective chapters:

Topic	Questions
First Program	1
Variables	1
Functions I	1
Conditional Statements	2
Functions II	2
Loops	2
Strings	2
List	2
Dictionary	2
Tuple	2
Set	1
Comprehension	2
Total	20

If you want to solve the final exam in Jupyter environment, you can download the notebook file for the final exam, **Final_Exam.ipynb**, from the [Github Repository](#) of this book. It is in the folder named *21_Final_Exam*. Here are the questions for the Final Exam:

QUESTIONS:

Q1:

$$5 * \sqrt{x} = 3^3 - 17$$

How do you solve this equation to find x, with Python?

- A `(3**3 - 17) / 5**2`
- B `((3**3 - 17) / 5)**2`
- C `(3**3 - 17) / 5***(1/2)`
- D `((3**3 - 17) / 5)**1/2`

Q2:

In Python we have rules for variable names. Which one below is a correct variable name in Python?

- A `3_quares`
- B `constant@number`
- C `set-item`
- D `OddNumbersSum`

Q3:

Define a function that take 3 integer parameters.

We want this function to return the maximum of these three numbers.

Which one below is the correct definition for this function?

Expected Output:

`get_max(5, 2, 8) -> 8`

- A `def get_max(n1, n2, n3):`
 `"""`

This function takes 3 numbers and returns the maximum.

Parameters: int n1, n2, n3

Returns: Maximum value via Python max function

`"""`

```
maximum = max(n1, n2, n3)
```

B `def get_max(n1, n2, n3):`

```
    """
```

This function takes 3 numbers and returns the maximum.

Parameters: int n1, n2, n3

Returns: Maximum value via Python max function

```
    """
```

```
maximum = max(n1, n2, n3)
```

C `def get_max(n1, n2, n3):`

```
    "
```

This function takes 3 numbers and returns the maximum.

Parameters: int n1, n2, n3

Returns: Maximum value via Python max function

```
    "
```

```
maximum = max(n1, n2, n3)
```

```
return maximum
```

D `def get_max(n1, n2, n3):`

```
    """
```

This function takes 3 numbers and returns the maximum.

Parameters: int n1, n2, n3

Returns: Maximum value via Python max function

```
    """
```

```
maximum = max(n1, n2, n3)
```

```
return maximum
```

Q4:

We want to define a function named **polygon_name**.

It will ask for number of sides from the user.

Then it will return one of the names below based on number of sides:

- if number of sides is 3 -> 'Triangle'
- if number of sides is 4 -> 'Quadrilateral'
- if number of sides is 5 -> 'Pentagon'
- if number of sides is 6 -> 'Hexagon'
- if number of sides is 7 or higher -> 'Polygon'
- if number of sides is less than 3 -> 'Unknown'

Which of the function definitions below is correct for this function?

Please enter the number of sides: **8**

Expected Output:

'Polygon'

A **def polygon_name():**

```

        name = ""

        num_of_sides = int(input("Please enter the number of
sides: "))

        if num_of_sides < 3:
            name = "Unknown"
        elif num_of_sides == 3:
            name = "Triangle"
        elif num_of_sides == 4:
            name = "Quadrilateral"
        elif num_of_sides == 5:
            name = "Pentagon"
        elif num_of_sides == 6:
            name = "Hexagon"
        else:
            name = "Polygon"

        return name
    
```

B **def polygon_name():**

```

        name = ""
    
```

```
num_of_sides = input("Please enter the number of sides: ")

if num_of_sides < 3:
    name = "Unknown"
elif num_of_sides == 3:
    name = "Triangle"
elif num_of_sides == 4:
    name = "Quadrilateral"
elif num_of_sides == 5:
    name = "Pentagon"
elif num_of_sides == 6:
    name = "Hexagon"
else:
    name = "Polygon"

return name
```

C **def polygon_name():**

```
name = ""

num_of_sides = int(input("Please enter the number of sides: "))

if num_of_sides > 3:
    name = "Unknown"
elif num_of_sides == 3:
    name = "Triangle"
elif num_of_sides == 4:
    name = "Quadrilateral"
elif num_of_sides == 5:
    name = "Pentagon"
elif num_of_sides == 6:
    name = "Hexagon"
else:
    name = "Polygon"

return name
```

```

D def polygon_name():

    name = ""

    num_of_sides = int(input("Please enter the number of
sides: "))

    if num_of_sides == 3:
        name = "Unknown"
    elif num_of_sides == 3:
        name = "Triangle"
    elif num_of_sides == 4:
        name = "Quadrilateral"
    elif num_of_sides == 5:
        name = "Pentagon"
    elif num_of_sides == 6:
        name = "Hexagon"
    else:
        name = "Polygon"

    return name

```

Q5:

We want to define a function.

The function will ask for a positive integer from the user.

It will check if the input is an integer or not.

And if the input is an integer then it will check if it is positive or not.

- If the input is not an integer -> It will return as 'Please enter an integer.' and it will exit the program (return)
- If the input is not positive -> It will return as 'Please enter a positive number.' and it will exit the program (return)

The function will decide if the number is 'FUNNY' based on:

- If the number is odd then it is 'FUNNY'

- If the number is even but between 2 (inc) and 41 (exc) then it is 'NOT FUNNY'
- If the number is even and greater than 40 then it is 'FUNNY'

Which of the function definitions below is correct for this function?

Hint: If the input is a negative number (-5) then isdigit() function will not accept it as an integer.

Enter a number and I will tell you, if it's FUNNY: 24

Expected Output:

'NOT FUNNY'

A `def is_funny():`

```
n = input("Enter a number and I will tell you, if it's
FUNNY: ")

if not n.isdigit():
    print('Please enter an integer.')

n = int(n)

if n <= 0:
    print('Please enter a positive number.')
else:
    if n % 2 == 1:
        print("FUNNY")
    else:
        if 2 <= n < 41:
            print("NOT FUNNY")
        elif n > 40:
            print("FUNNY")
```

B `def is_funny():`

```
n = input("Enter a number and I will tell you, if it's
FUNNY: ")

if not n.isdigit():
```

```
print('Please enter an integer.')
return

n = int(n)

if n <= 0:
    print('Please enter a positive number.')
    return

else:
    if n % 2 == 0:
        print("FUNNY")
    else:
        if 2 <= n < 41:
            print("NOT FUNNY")
        elif n > 40:
            print("FUNNY")
```

C `def is_funny():`

```
n = input("Enter a number and I will tell you, if it's
FUNNY: ")

if not n.isdigit():
    print('Please enter an integer.')
    return

n = int(n)

if n <= 0:
    print('Please enter a positive number.')
    return

else:
    if n % 2 == 1:
        print("FUNNY")
    else:
        if 2 <= n < 41:
            print("NOT FUNNY")
        elif n > 40:
```

```
    print("FUNNY")
```

D **def** is_funny():

```
    n = input("Enter a number and I will tell you, if it's
FUNNY: ")

    if not n.isdigit():
        print('Please enter an integer.')
        return

    n = int(n)

    if n <= 0:
        print('Please enter a positive number.')
        return

    else:
        if n % 2 == 1:
            print("FUNNY")
        else:
            if 2 > n < 41:
                print("NOT FUNNY")
            elif n > 40:
                print("FUNNY")
```

Q6:

We want to define a function named
total_and_number_of_elements.

It will take unknown number of parameters.

And it will return the summation and number of elements for
parameters.

It will return these result in a tuple.

Which one below is NOT CORRECT definition and function
call for this function?

(Either definition or function call is not correct.)

A **def** total_and_number_of_elements(args):
 total = sum(args)
 num_of_elements = len(args)

```
    return (total, num_of_elements)  
total, num_of_elements =  
total_and_number_of_elements(2, 5, 11)  
print(total)  
print(num_of_elements)
```

B `def total_and_number_of_elements(*args):
 total = sum(args)
 num_of_elements = len(args)
 return (total, num_of_elements)`

```
total, num_of_elements =  
total_and_number_of_elements(2, 5, 11)  
print(total)  
print(num_of_elements)
```

C `def total_and_number_of_elements(*args):
 total = sum(args)
 return (total, len(args))
(total, num_of_elements) =
total_and_number_of_elements(2, 5, 11)
print(total)
print(num_of_elements)`

D `def total_and_number_of_elements(*args):
 return (sum(args), len(args))
(total, num_of_elements) =
total_and_number_of_elements(2, 5, 11)
print(total)
print(num_of_elements)`

Q7:

We want to define a function named **area_of_circle**.
It will take the radius (r) as parameter. The default value will
be 10 for radius.

The function will not create any variables and its body will be single line of code.

Area of Circle = $\pi * r^2$

Which one below is NOT CORRECT definition and function call for this function?

(Either definition or function call is not correct.)

A **import math**

```
def area_of_circle(r=10):  
    return math.pi * r**2  
  
area = area_of_circle()
```

B **import math**

```
def area_of_circle(r=10):  
    return math.pi * r**2  
  
area = area_of_circle(10)
```

C **import math**

```
def area_of_circle(r=10):  
    return math.pi * r**2  
  
area = area_of_circle(r=10)
```

D **import math**

```
def area_of_circle(r=10):  
    return math.pi * r**2  
  
area(10) = area_of_circle()
```

Q8:

We want to define function named **four_six**.

It will loop over the numbers from 1 to 50 (both included).

- If the number is a multiple of 4 -> it will print 'Four' instead of the number
- If the number is a multiple of 6 -> it will print 'Six' instead of the number

- If the number is a multiple of both 4 and 6 -> it will print 'FourSix' instead of the number

In any case other than these, it will NOT PRINT anything.
Which one below can achieve this task?

Expected Output:

Four
Six
Four
FourSix
Four
Six
Four
FourSix
Four
Six
Four
FourSix
Four
Six
Four
FourSix
Four
Six
Four
FourSix

A **def four_six():**

```

four = "Four"
six = "Six"

for i in range(1, 51):
    if i % 4 == 0 and i % 6 == 0:
        print(four+six)
    elif i % 4 == 0:
        print(four)
    elif i % 6 == 0:
        print(six)
    else:
        print(i)

```

```
four_six()
```

B `def four_six():`

```
four = "Four"
six = "Six"

for i in range(1, 51):
    if i % 4 == 0 and i % 6 == 0:
        print(four+six)
    elif i % 4 == 0:
        print(four)
    elif i % 6 == 0:
        print(six)
```

`four_six()`

C `def four_six():`

```
four = "Four"
six = "Six"

for i in range(1, 51):
    if i % 4 == 0 and i % 6 == 0:
        print(four+six)
    elif i % 4 == 0:
        print(six)
    elif i % 6 == 0:
        print(four)
```

`four_six()`

D `def four_six():`

```
four = "Four"
six = "Six"

for i in range(1, 51):
    if i % 4 == 0 or i % 6 == 0:
        print(four+six)
    elif i % 4 == 0:
```

```
    print(four)
elif i % 6 == 0:
    print(six)

four_six()
```

Q9:

We want to draw a triangle of stars.

The function will take the number of stars as parameter.

Let's say the number of stars is 5:

```
*
```



```
* *
```



```
* * *
```



```
* * * *
```



```
* * * * *
```

Which one below can achieve this task?

A `def triangle(n):`
 `for i in range(n):`
 `stars = ""`
 `for j in range(i-1):`
 `stars += "* "`

 `print(stars)`

`triangle(5)`

B `def triangle(n):`
 `for i in range(n):`
 `stars = ""`
 `for j in range(i):`
 `stars += "* "`

 `print(stars)`

`triangle(5)`

C `def triangle(n):`
 `for i in range(n):`
 `stars = ""`

```
for j in range(i+1):
    stars += "* "
print(stars)

triangle(5)
```

D `def triangle(n):
 for i in range(n+1):
 stars = ""
 for j in range(i+1):
 stars += "* "
 print(stars)
triangle(5)`

Q10:

We want to define a function `title_case`.

It will take a text as parameter and convert this text into "title case".

"title case" means; all the letters are lower case except the first letters. Only the first letters are in capital case.

'pYtHon iS cOoL' -> 'Python Is Cool'

A `def title_case(text):
 return text.title().upper()
title_case("pYtHon iS cOoL")`

B `def title_case(text):
 return text.title().lower()
title_case("pYtHon iS cOoL")`

C `def title_case(text):
 return text.lower().upper()
title_case("pYtHon iS cOoL")`

D `def title_case(text):
 return text.lower().title()`

```
title_case("pYtHon iS cOoL")
```

Q11:

Our text is 'Python Django Numpy'.

Which one below is NOT CORRECT when you do slicing operations?

text = 'Python Django Numpy'

- A The last character in the text is `text[-1]` and its value is 'y'.
- B The last 4 characters is `text[-5:]` and its value is 'umpy'.
- C You can use `text[::-1]` to get the text in reverse order.
- D The characters between 2nd and 8th are `text[2:8]` and the value is 'thon D'.

Q12:

We want to define a function named **quadrapol**.

It will take a list as parameter.

The function will calculate the power of 4 of each element in the list and append it to a new list.

At the end, it will return this new list of power 4.

[1, 2, 3, 4, 5] -> [1, 16, 81, 256, 625]

Which one below CAN NOT achieve this task?

A **def quadrapol(a_list):**
 new_list = []
 for i **in** a_list:
 new_list.add(i**4)

 return new_list

B **def quadrapol(a_list):**
 new_list = **list**()
 for i **in** a_list:
 new_list.append(i**4)

 return new_list

C **def quadrapol(a_list):**

```
new_list = []
for i in a_list:
    new_list.extend([i**4])
return new_list
```

D `def quadrapol(a_list):`
 `new_list = list()`
 `for i in a_list:`
 `new_list.insert(len(new_list), i**4)`
 `return new_list`

Q13:

We want to define a function named `crop`.

Cropping is deleting the last two elements in a sequence.

The function will take a list and parameter and will mutate this list.

It will crop the original list and returns nothing.

Which one below is this function?

```
a_list = [1, 2, 3, 4, 5, 6, 7]
crop(a_list)
print(a_list) -> [1, 2, 3, 4, 5]
```

A `def crop(a_list):`
 `a_list.pop(-1)`
 `a_list.pop(-2)`

B `def crop(a_list):`
 `a_list.pop(len(a_list))`
 `a_list.pop()`

C `def crop(a_list):`
 `del a_list[0]`
 `a_list.pop()`

D `def crop(a_list):`
 `del a_list[-1]`
 `a_list.pop()`

Q14:

We want to define a dictionary named movie as follows:

```
movie = {  
    'name': 'Magnolia',  
    'year': 1999,  
    'director': 'Paul Thomas Anderson',  
    'imdb': 8.0  
}
```

Which one below CAN NOT create this dictionary?

A `movie = {}`

`movie['name'] = 'Magnolia'`

`movie['year'] = 1999`

`movie.update({'director': 'Paul Thomas Anderson'})`

`movie.update({'imdb': 8.0})`

B `movie = dict()`

`movie.add({'name': 'Magnolia'})`

`movie.add({'year': 1999})`

`movie.add({'director': 'Paul Thomas Anderson'})`

`movie.add({'imdb': 8.0})`

C `movie = dict({`

`'name': 'Magnolia',`

`'year': 1999,`

`'director': 'Paul Thomas Anderson',`

`'imdb': 8.0`

`})`

D `movie = dict({})`

`movie.update({`

`'name': 'Magnolia',`

`'year': 1999,`

`'director': 'Paul Thomas Anderson',`

`'imdb': 8.0`

`})`

Q15:

If we run the code below, what will be the value of the **result** variable?

```
result = {}

d1 = {'a': 1, 'b': 3, 'c':5}
d2 = {'a': 4, 'b': 6, 'd':9}

for key in d1:
    if key in d2:
        result[key] = d1[key] * d2[key]
```

- A {'a': 4, 'b': 18, 'c':5}
- B {'a': 4, 'b': 18, 'c':5, 'd': 9}
- C {'a': 4, 'b': 18}
- D {'a': 1, 'b': 3}

Q16:

We want to define a function which takes a Tuple and an element as parameters.

The function name will be **how_many_occurrences**.

The function will return the number occurrences of this element in the Tuple.

```
tup = (4, 3, 5, 2, 3, 3, 4, 2, 1, 3, 4, 5, 2, 1, 3)
element = 3
```

Expected Output:

```
how_many_occurrences (tup, element) -> 5
```

Which one below is this function?

- A

```
def how_many_occurrences(tup, element):
    occurrences = 0
    for e in list(tup):
        if e == element:
            occurrences += 1
    return occurrences
```
- B

```
def how_many_occurrences(tup, element):
    occurrences = 0
```

```

for i, e in enumerate(tup):
    if i == element:
        occurrences += 1

return occurrences

```

C **def** how_many_occurrences(tup, element):
 occurrences = 0
for key, value **in** tup.items():
if value == element:
 occurrences += 1

return occurrences

D **def** how_many_occurrences(tup, element):
 occurrences = 0
for i, e **in** list(tup):
if e == element:
 occurrences += 1

return occurrences

Q17:

We have the tuple below:

tup = ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j')

Which of the question-answer pairs below is NOT CORRECT?

- A Q: Find the items from 3rd index to 6 (both included).
 A: tup[3:7]
- B Q: Find the elements starting from the 5th one (inc).
 A: tup[4:]
- C Q: write the tup in reverse order with step-size 3.
 A: tup[::-3]
- D Q: Get all the items up to the 2nd (exc) in the last via negative index.
 A: tup[:-3]

Q18:

Which one below is NOT CORRECT about the **Sets** (Set Data Structure)?

- A We use empty curly braces {} to create a Set.
- B An element cannot exist more than once in a Set.
- C We use 'intersection()' method to get common elements of two Sets.
- D We use 'add()' method to add elements to a Set.

Q19:

We want to print all the words in a paragraph.
We can achieve this via for loop as follows:

```
paragraph = ["Lorem ipsum dolor sit amet.",  
            "Ut enim ad minim veniam.",  
            "Duis aute irure dolor.",  
            "Excepteur sint."]  
  
words = []  
  
for sentence in paragraph:  
    for word in sentence.split():  
        words.append(word)
```

What is the **Comprehension** form of this loop?

- A words = [sentence
 for sentence **in** paragraph
 for word **in** sentence.split()]
- B words = [word
 for sentence **in** paragraph
 for word **in** sentence.split()]
- C words = [word, sentence
 for sentence **in** paragraph
 for word **in** sentence.split()]
- D words = [word

```
for sentence in paragraph.split()
    for word in sentence.split()
```

Q20:

We have a dictionary of heroes:

```
heroes = {
1: ['Wonder Woman', 'Diana Prince'],
2: ['Batman', 'Bruce Wayne'],
3: ['Superman', 'Clark Kent'],
4: ['Spiderman', 'Peter Parker']
}
```

We only want to get the **odd** numbered heroes with their secret names and create a new dictionary with this data.

How can we succeed this via **Comprehension**?

Expected Output:

```
secret_names = {1: 'Diana Prince', 3: 'Clark Kent'}
```

- A secret_names = {i: name[1] for i, name in heroes.items()}
- B secret_names = {key: value[0] for key, value in heroes.items() if key % 2 != 0}
- C secret_names = {num: sec[1] for num, sec in heroes.items() if num % 2 == 1}
- D secret_names = {k: v[1] for k, v in heroes.items() if k % 2 == 0}

ANSWERS OF THE FINAL EXAM QUESTIONS:

Question	Answer
Q1	B
Q2	D
Q3	D
Q4	A
Q5	C
Q6	A
Q7	D
Q8	B
Q9	C
Q10	D
Q11	B
Q12	A
Q13	D
Q14	B
Q15	C
Q16	A
Q17	D
Q18	A
Q19	B
Q20	C

22. Conclusion

This is the last chapter in this book. We have covered almost all the fundamental concepts in Python and we covered them in great detail. We started with setting our development environment. Then we defined our First program and said ‘Hello World’ to Python. Then we moved on with Variables, Functions and we finished our First Project. We learned Conditional Statements and we moved on with the second part of the Functions. Then we learned Loops and Strings. After Strings we built the Second Project. Then came the Lists, Dictionaries, Tuples, Sets and Comprehensions. And we finished the third Project after these topics. Finally you got the Final Exam to test yourself.

The approach we followed in this book is quite unique and intense. It aims to teach you Python in a solid and unforgettable way. That’s why we had more than 200 coding exercises, quizzes and assignments. The idea is to make sure we used all the possible ways to help you learn Python programming. And I hope, I achieved this.

This book is a part of the Hands-On Python Series. There are three books in this series: Beginner, Intermediate and Advanced. This book is the first one, the Beginner one. You can start the

Intermediate part after you finish this book and feel comfortable with the basics of Python.

Dear reader!

I want to thank you with all my heart, for your interest in my book, your patience and your desire to learn Python. You did a great job finishing this intense book. And I am very happy to be a part of this. I hope we see each other again in another programming book. Till then, I hope, you have wonderful life and reach your dreams.

Good bye.

Musa Arda

[1] Anaconda Installation: <https://docs.anaconda.com/anaconda/install/index.html>

[2] Jupyter Notebook Security: <https://jupyter-notebook.readthedocs.io/en/stable/security.html>

[3] The JupyterLab Interface: <https://jupyterlab.readthedocs.io/en/stable/user/interface.html>

[4] Python Virtual Environments: <https://docs.python.org/3/tutorial/venv.html>

[5] Central Repository for Python Packages: <https://pypi.org/>

[6] The Turtle Module official documentation:

<https://docs.python.org/3/library/turtle.html>

[7] String Methods documentation:

<https://docs.python.org/3/library/stdtypes.html#string-methods>

OceanofPDF.com