Introduction to Python

Module I: Basics

Objectives

- Interact with Python using the command line.
- Run Python in both interactive and scripting modes.
- Launch Jupyter Lab from your operating system.
- Utilize Jupyter Lab capabilities and working environments.
- Create a Jupyter notebook.
- Execute Python codes inside a Jupyter notebook.
- Use various environments and IDEs for writing and running Python codes.
- Define variables in Python.
- Use various data types in Python.
- Ask for the user's input and store those values inside variables.
- Explain the purpose of functions in Python.
- Correctly call built-in functions in Python.
- Convert variables from one type to another using built-in functions.
- Use help to display documentation for built-in functions.
- Use comments in Python.
- Explain various error messages.
- Use lists as a collection of data.
- Append items to a list.
- Modify a list.
- Understand how to slice strings into substrings.
- Explain differences between lists and tuples in Python.
- Correctly use dictionaries.
- Explain what "for" loops are normally used for.
- Trace the execution of a loop and correctly state the values of variables in each iteration.
- Correctly write programs that use if and else statements and simple Boolean expressions.
- Explain the differences between function definition and function call.

- Write a function that takes a fixed number of arguments and produces results.

Introduction

This lesson is an introduction to programming in Python for people with little or no previous programming experience. Python is a popular language for research computing, data analytics, and great for general-purpose programming as well. Installing all of the required packages individually can be a bit difficult and therefore, we recommend Anaconda, which is an all-in-one installer for Python and its most popular libraries. We covered the setup process of python in our "Python Setup" document in detail. First, we cover executing a Python code, the differences between python interactive and scripting mode, we will overview various Python IDEs including Jupyter Lab, and finally cover the topic of defining variables in Python and different types of variables. By the end of the lesson, you will have a solid foundation in working with Jupyter Lab and variables in Python. The second part of the lesson aims to introduce some of the main built-in functions in Python and explain how to use lists for simplifying the process of data collection and analysis. Finally, the last part of the lesson aims to introduce more advanced techniques in coding via Python. We will learn how to use tuples, dictionaries, loops, conditional statements and finally how to define and write a function ourselves.

1. Interact with Python via command line

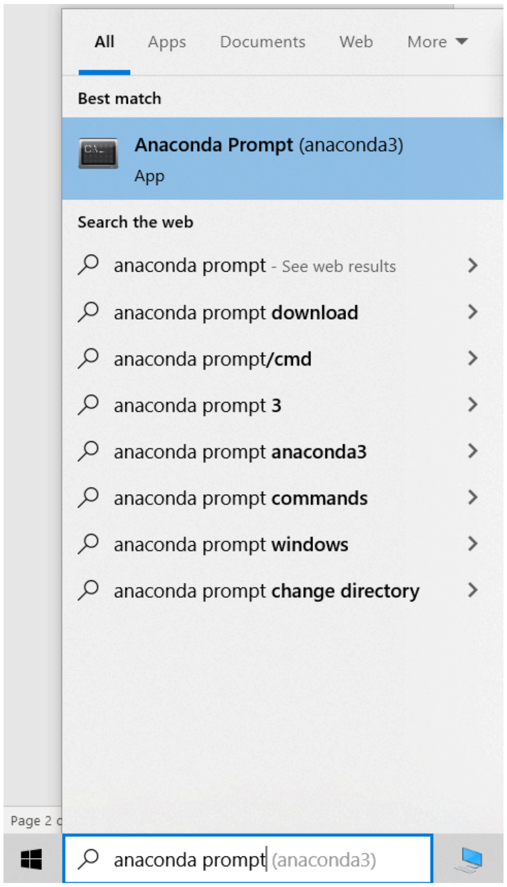
The first thing you need to learn is how to run commands and scripts in Python. The first question that you may ask is what the difference between a code and a script is? In computing, the code is a language that the computer understands. Codes are sequences of instructions that computers can follow to perform tasks. It is common for coding instructions to use human language to ease code comprehension. The term code can refer to a set of instructions comprising a program. A simple function or a statement can also be considered a code. On the other hand, a script is a file consisting of a logical sequence of instructions or a batch-processing file that is interpreted by another program instead of directly by the computer processor.

Note

In simple terms, a Python script is a simple program stored in a plain text file which contains Python code.

In order to execute Python code or Python script, you need a Python interpreter. The interpreter is a layer of software that works as a bridge between the program and the system hardware. It translates text commands into machine language and executes them. A Python interpreter is an application that is responsible for running Python scripts and codes. When you install the Anaconda distribution a Python interpreter is installed automatically. The most basic way of coding with Python is to use a command line. We will need to open a terminal on our computer to access Python via command line. While the command line is the most "basic" way to interact with Python, an IDE is the most common and we will cover that in chapter 2.

In windows, you can open a terminal by looking up “Anaconda Prompt” in the search bar of windows and then clicking on it (see the screenshot below).



anaconda prompt (anaconda3)

Figure 1: Launching Anaconda Prompt in windows.

In Mac OS, you can open a terminal by launching the terminal (see the screenshot below).

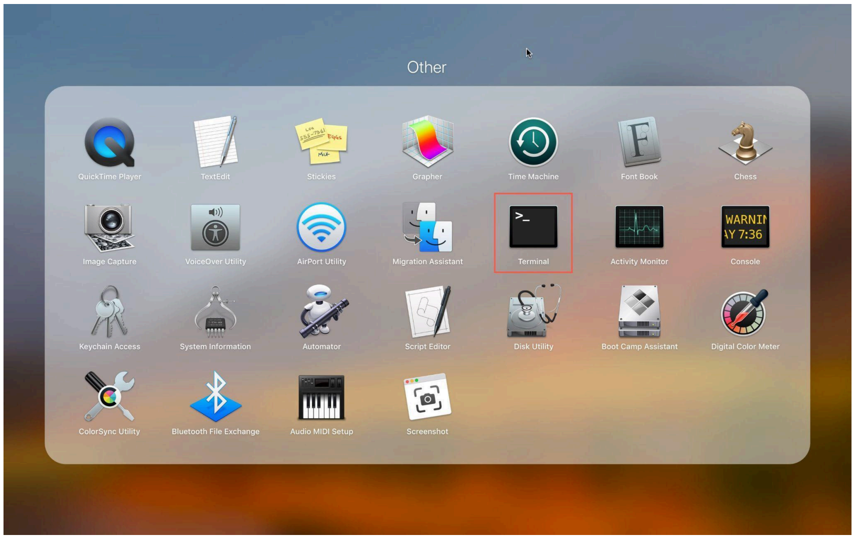
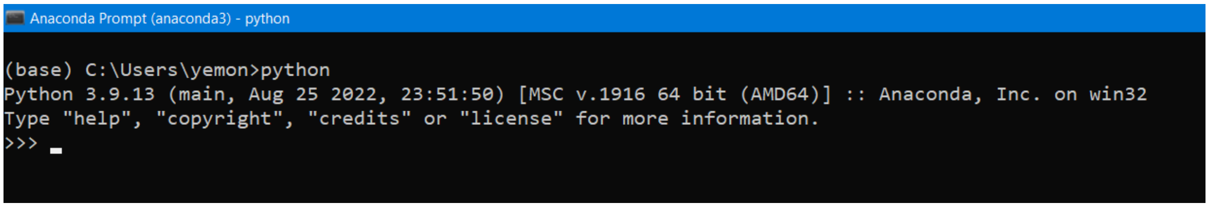


Figure 2: Launching terminal in Mac OS.

On GNU/Linux, the command line can be accessed by several applications like xterm, Gnome Terminal or Konsole.

Once you open the terminal, create or navigate to your working directory. A working directory is the directory or folder where you want to store and organize project-related data, scripts, plots, and notes. Once you navigate to your working directory, you need to type in “python” or “python3” to start coding in Python. You should see a three-arrows sign (“>>>”) which indicates that the terminal is ready to accept your Python commands. This sign on the terminal represents the standard prompt for the interactive mode. You can also check the version of your Python interpreter here (see the screenshot below).



```
(base) C:\Users\yemon>python
Python 3.9.13 (main, Aug 25 2022, 23:51:50) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 
```

Figure 3: Interacting with Python using terminal (windows OS).

The statements you write when working with an interactive session are evaluated and executed immediately. As an example, you can use Python as an advanced calculator. Note that in Python, we can add comments by using hashtag *(#)* in front of a sentence.

Code:

```
# Using Python for adding two numbers
2 + 3
```

Output:

```
5
```

This is the interactive mode of Python. The interactive mode is useful when you are developing a program and you need to check the results of a command (or a short sequence of commands) quickly. The main disadvantage of interactive mode is that when you close the interactive session, the code no longer exists. This is the main motivation for writing Python code as a script and then running it.

Note

- You can run Python commands in two modes: interactive mode and scripting mode.
- Interactive mode is usually used when a user wants to run one single line or one block of code.
- Scripting Mode is usually used when the user is working with more than one single code or a block of code.

One way to write a Python program is using a plain text editor. To indicate that a file contains Python code, the extension ".py" is used. Following this convention, humans and computers can quickly identify and distinguish Python files from other types. If you are at the beginning of working with Python, you can use editors like Nano or Notepad which are easy to use, or any other text editors.

Now it's time to create your first test script. In order to do that, open your most-suited text editor and write the following code:

Code:

```
# Example for printing a sentence on screen
print('Hello World!')
```

Then save the file in your working directory with the name first_script.py or anything you like. Remember you need to give the .py extension only. The most basic and easy way to run a Python script is by using the python command. You need to open a command line (terminal) and type the word “python” followed by the path to your script file like this:

Code:

```
# Running your first script in Python via terminal
python first_script.py
```

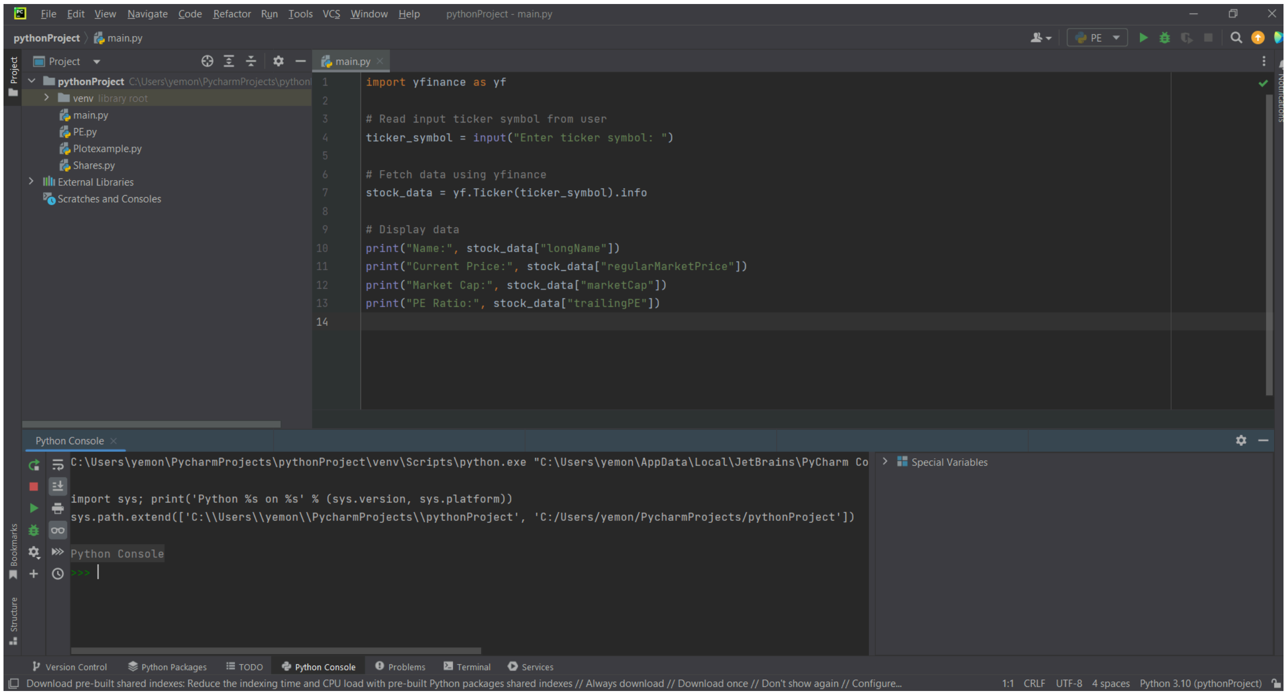
Output:

```
Hello World!
```

Then you hit the ENTER button on the keyboard, and that's it. You can see the phrase “Hello World!” on the screen. Congrats! You just ran your first Python script.

2. Overview of Python IDEs: PyCharm and Spyder

To create or edit a more advanced Python program, software developers often use an integrated development environment (IDE) like PyCharm, Spyder, VS code or Jupyter Lab to be able to have more control over their Python scripts and codes. IDEs provide many useful features such as keyword highlighting, error highlighting, code suggestion, keeping track of all variables and running the code without the need for launching the command line. They also provide a way to manage the project's files in an easier way in the case of a multi-file application. There are several Python IDEs to choose from and here we want to take a look at the three most important ones. Here you can a screenshot from PyCharm environment:



```python
import yfinance as yf

# Read input ticker symbol from user
ticker_symbol = input("Enter ticker symbol: ")

# Fetch data using yfinance
stock_data = yf.Ticker(ticker_symbol).info

# Display data
print("Name:", stock_data["longName"])
print("Current Price:", stock_data["regularMarketPrice"])
print("Market Cap:", stock_data["marketCap"])
print("PE Ratio:", stock_data["trailingPE"])
```

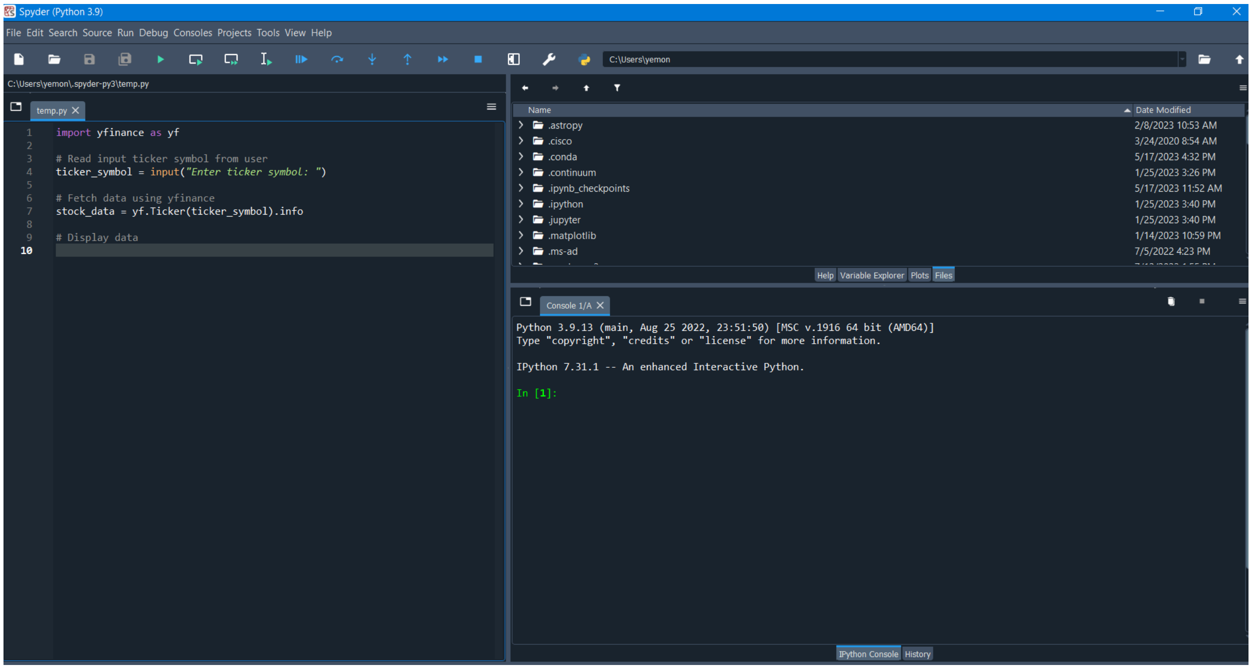
```
C:\Users\yemon\PycharmProjects\pythonProject\venv\Scripts\python.exe "C:\Users\yemon\AppData\Local\JetBrains\PyCharm Co

import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['C:\\Users\\yemon\\PycharmProjects\\pythonProject', 'C:/Users/yemon/PycharmProjects/pythonProject'])

Python Console
>>> 
```

Download pre-built shared indexes: Reduce the indexing time and CPU load with pre-built Python packages shared indexes // Always download // Download once // Don't show again // Configure...

Figure 4: PyCharm environment.

In summary, PyCharm environment consists of a top panel, a project manager (file manager) window on the top left, a main working area on the top right, and bottom tabs on the bottom. Using PyCharm you can view and manage your project files, working on your Python scripts, and also interact with Python in interactive mode via Python Console at the same time.

The next Python IDE that we briefly introduce here is Spyder.


Spyder (Python 3.9)
File Edit Search Source Run Debug Consoles Projects Tools View Help
C:\Users\yemon
C:\Users\yemon\.spyder-py3\temp.py
temp.py
Help Variable Explorer Plots Files
Console 1/A
IPython Console History

```
import yfinance as yf

# Read input ticker symbol from user
ticker_symbol = input("Enter ticker symbol: ")

# Fetch data using yfinance
stock_data = yf.Ticker(ticker_symbol).info

# Display data

```

Name	Date Modified
.astropy	2/8/2023 10:53 AM
.cisco	3/24/2020 8:54 AM
.conda	5/17/2023 4:32 PM
.continuum	1/25/2023 3:26 PM
.ipynb_checkpoints	5/17/2023 11:52 AM
.ipython	1/25/2023 3:40 PM
.jupyter	1/25/2023 3:40 PM
.matplotlib	1/14/2023 10:59 PM
.ms-ad	7/5/2022 4:23 PM

```
Python 3.9.13 (main, Aug 25 2022, 23:51:50) [MSC v.1916 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more information.

IPython 7.31.1 -- An enhanced Interactive Python.

In [1]:
```

Figure 5: Spyder environment.

3. Execute Python Commands in JupyterLab

For the rest of the Python course and to run Python codes, we are going to use Jupyter Notebooks via JupyterLab. Jupyter notebooks are common in data science and visualization and serve as a convenient experience for running Python code interactively where we can easily view and share the results of our Python code. JupyterLab has several other handy features:

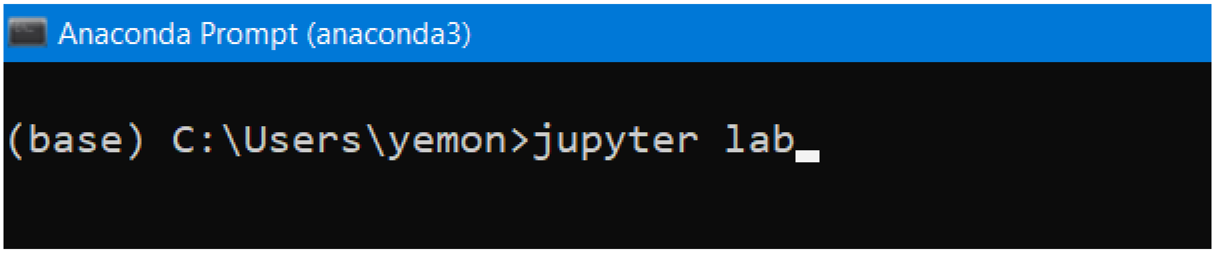
- You can easily type, edit, copy, and paste blocks of code.
- Tab complete allows you to easily access the names of things you are using and learn more about them.

- It allows you to annotate your code with links, different sized text, bullets, etc. to make it more accessible to you and your collaborators.
- It allows you to display figures next to the code that produces them to tell a complete story of the analysis.

Each notebook contains one or more cells that contain code, text, or images. JupyterLab is an application server with a web user interface from Project Jupyter that enables one to work with documents and activities such as Jupyter notebooks, text editors, terminals, and even custom components in a flexible, integrated, and extensible manner. JupyterLab requires a reasonably up-to-date browser (ideally a current version of Chrome, Safari, or Firefox); Internet Explorer versions 9 and below are not supported.

3.1. Launching JupyterLab

JupyterLab is included as part of the Anaconda Python distribution so no further setup process is required. To launch JupyterLab, you first need to open your terminal (command line). In Windows, you need to start Anaconda Prompt (terminal in MacOS), navigate to your working directory and then type: “jupyter lab” and press enter or return (see the below screenshot from Anaconda Prompt).



```
(base) C:\Users\yemon>jupyter lab
```

Figure 6: Launching Jupyter Lab from Anaconda Prompt for Windows (or terminal in MacOS).

JupyterLab will be launched in your default web browser (Google Chrome, FireFox, Safari, Microsoft Edge etc). The below screenshot shows what JupyterLab will look like after launching in your web browser. JupyterLab does not need an internet connection to launch and it will run locally on your computer through a port (by default port 8888).

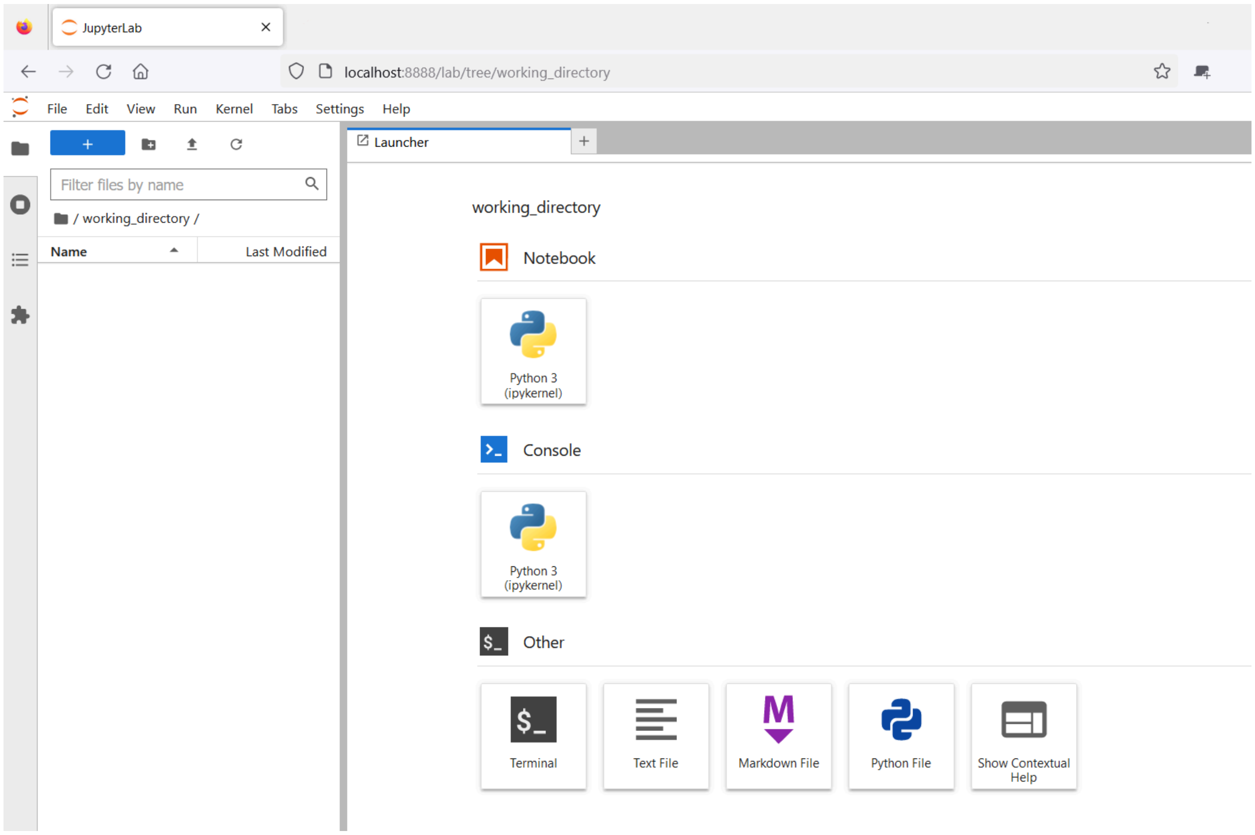


Figure 7: JupyterLab Interface.

JupyterLab has many features found in other integrated development environments (IDEs) like PyCharm or Spyder but is focused on providing flexible building blocks for interactive coding and scientific computing/illustrations.

3.2. JupyterLab Interface

The interface of JupyterLab includes a menu bar at the top, a left sidebar, and the main working area which contains tabs of notebooks, consoles, terminals, documents and activities. The menu bar at the top of JupyterLab has the top-level menus that expose various actions available in JupyterLab.

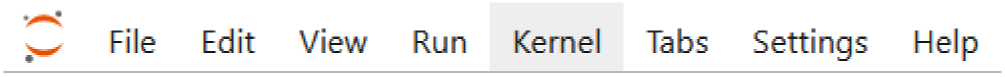


Figure 8: JupyterLab top menu options.

The following menus are included in the top menu by default.

- **File:** Actions related to files and directories such as *New*, *Open*, *Close*, *Save*, etc. The *File* menu also includes the *Shut Down* action used to shutdown the JupyterLab server.
- **Edit:** Actions related to editing documents and other activities such as *Undo*, *Cut*, *Copy*, *Paste*, etc.
- **View:** Actions that alter the appearance of JupyterLab.
- **Run:** Actions for running code in different activities such as notebooks and code consoles (discussed below).
- **Kernel:** Actions for managing kernels. Kernels in Jupyter are processes that are responsible for running your code in different programming languages and environments. Kernels are like the backend engine of Jupyter which can communicate with a Python interpreter to translate your codes into machine language (the language consists of a series of binary codes, 0 and 1s) and return the result of computations back to the Jupyter interface.
- **Tabs:** A list of the open documents and activities in the main work area.
- **Settings:** Common JupyterLab settings can be configured using this menu. There is also an *Advanced Settings Editor* option in the dropdown menu that provides more fine-grained control of JupyterLab settings and configuration options.
- **Help:** A list of JupyterLab and kernel help links.

The left sidebar in JupyterLab contains a number of commonly used tabs, such as a file browser (showing the contents of the directory where the JupyterLab server was launched), a list of running kernels and terminals, the command palette, and a list of open tabs in the main work area. A screenshot of the default Left Side Bar is provided below.

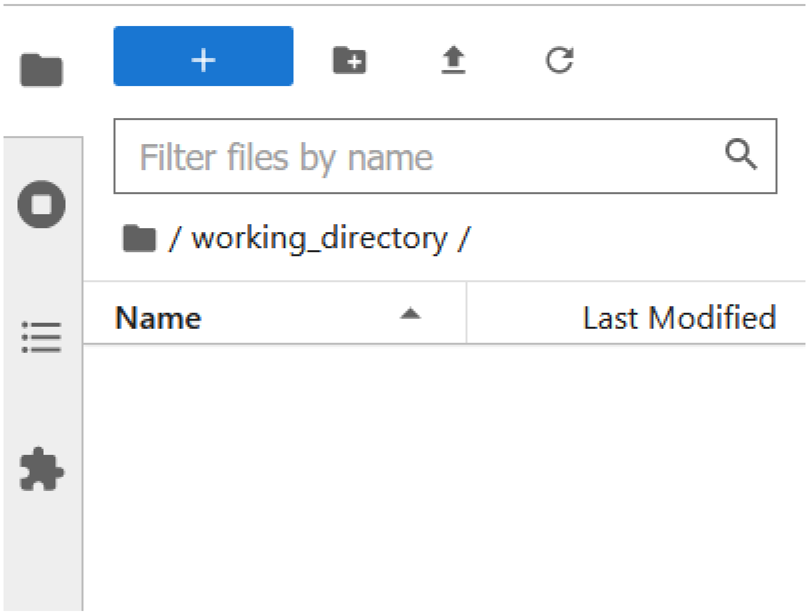


Figure 9: JupyterLab left side menu options.

The main work area in JupyterLab enables you to arrange documents (notebooks, text files) and other activities (terminals, consoles) into panels of tabs that can be resized or subdivided. A screenshot of the default main working area is provided below.

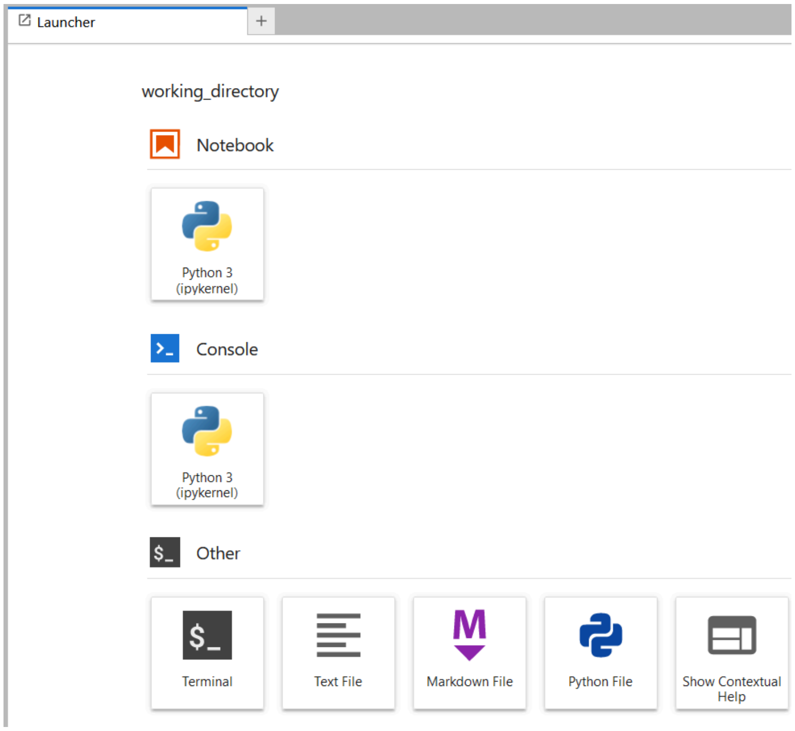


Figure 10: JupyterLab main working area.

If you do not see the Launcher tab, click the blue plus sign under the “File” and “Edit” menus and it will appear.

3.3. Creating a Python script in JupyterLab

To create a python script in JupyterLab, you have two options: first, you can click on the “Text File” under “Other” in the main working area (via launcher) to create a plain text file. Another way to create a text file is by selecting the *New -> Text File* from the *File* menu in the top menu bar. Once you created your plain text file, you can write your Python code inside this text file. After finishing your code, you can rename your text file to a name that ends with the “.py” extension. The “.py” extension lets everyone (including the operating system) know that this text file is a Python program. For example, you can save your file as “test_script.py”.

The second way (and the best way) to create a Python script in JupyterLab is by selecting the “Python File” under “Other” in the main working area (via launcher). The Python file will automatically save your code into a file with “.py” extension.

3.4. Creating a Jupyter Notebook in JupyterLab

To open a new notebook, click the Python 3 icon under the *Notebook* header in the Launcher tab in the main work area. You can also create a new notebook by selecting *New -> Notebook* from the *File* menu in the Menu Bar.

Additional notes on Jupyter notebooks.

- Notebook files have the extension “.ipynb” to distinguish them from plain-text Python programs.
- Notebooks can be exported as Python scripts that can be run from the command line.

Jupyter lets you mix code and text by using different types of blocks, called cells. We often use the term “code” to refer to the source code of software written in a language such as Python. A “code cell” in a Notebook is a cell that contains software; a “text cell” is one that contains ordinary prose written for human beings. If you press Esc and

Return alternately, the outer border of your code cell will change from gray to blue. These are the **Command** (gray) and **Edit** (blue) modes of your notebook. Command mode allows you to edit notebook-level features, and Edit mode changes the content of cells. For example you can add new cells (by clicking on the new icon or pressing “b” on the keyboard) or you can delete a cell (by clicking on remove or pressing “x” on the keyboard).

Notebooks can also render Markdown which is a simple plain-text format for writing lists, scientific formulas, links, and other things that might go into a web page or report. For example, you can use * to create bullet lists, numbers to create numbered lists, You can use indents to create sublists, or $ to render scientific equations and mathematical operators.

As an example, you can use Markdown to create a list of tasks inside a Jupyter Notebook. To convert a cell to Markdown, you can select your cell and then choose “markdown” option from the drop down menu in JupyterLab (see the below screenshot) or by pressing “M” on your keyboard when you are in the Command mode (gray border).

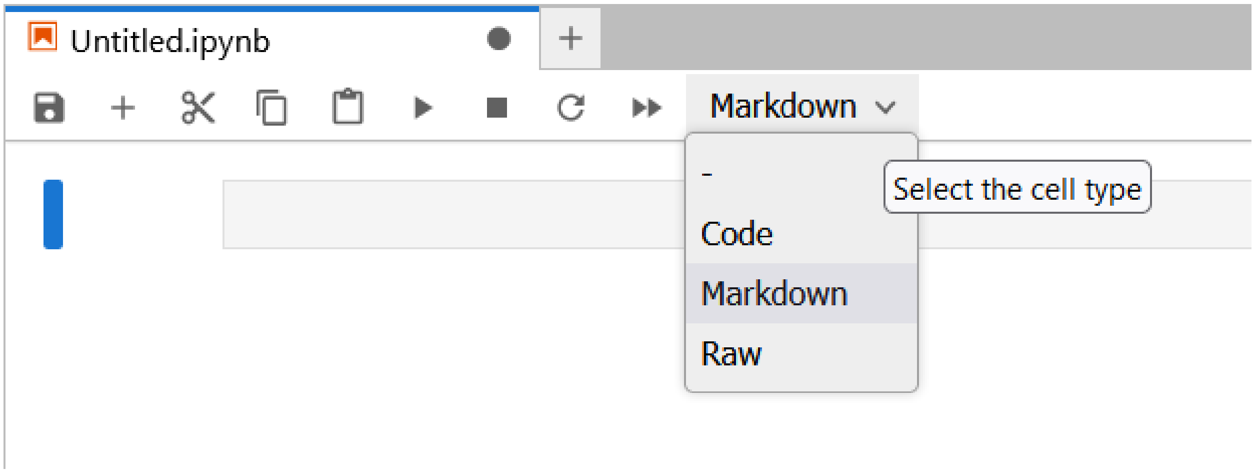


Figure 11: How to convert a cell from code into markdown in Jupyter.

After you convert a cell to Markdown, you can create a list:

Code:

```
# Using Markdown and * to create a list
```

```
* First topic
* Second topic
* Third topic
```

To run a Markdown or a code in Jupyter Notebook, you need to hold down "Shift" and then press "Enter" or "Return".

Output:

```
●  First topic
●  Second topic
●  Third topic
```

Exercise 3.1

Create a Markdown with three bullet points (first, second and third), and two sublists for the second bullet point (first subtopic and second subtopic). Remember that you should use indents to create sublists in Jupyter.

Another useful feature of Jupyter Notebook is checkpoints. Jupyter creates a checkpoint file every single time you create an .ipynb file, and then it updates the checkpoint file every single time you manually save your progress for the initial .ipynb.

A manual save is what's done when the file is saved by clicking the Save and Checkpoint button. Auto-saving, on the other hand, updates only the initial .ipynb file, not the checkpoint file. When you revert from the initial .ipynb file to a previously saved checkpoint by using the "Revert to Checkpoint" button from the file menu, the checkpoint file is what gets accessed and opened inside Jupyter. As a side note, the checkpoint file is located within a hidden folder named .ipynb_checkpoints. This folder is located within the same folder as the initial .ipynb file.

Finally, you can shut down and close JupyterLab. From the Menu Bar select the "File" menu and then choose "Shut Down" at the bottom of the dropdown menu. You will be prompted to confirm that you wish to shutdown the JupyterLab server (don't forget to save your work!). Click "Shut Down" to shutdown the JupyterLab server. You can re-launch Jupyter through your terminal and continue to work on all of your previous codes/files.

4. Variables in Python

In the previous examples of coding in Python, we saw that we can use Python as an advanced calculator. However, the results of these computations are lost after we perform our operations. If we want to save and reuse any value in a code, we need to save these computations in a place in computer memory to be able to access it in the future. Variables are names that we associate with a memory location in computers. As such variables are not values themselves but they point to a memory location where the actual values are stored. Variable names in Python can only contain letters, digits, and underscore _ (typically used to separate words in long variable names) and they cannot start with a digit. Variables in Python are also **case sensitive**. For example, age, Age and AGE are three different variables. The name we choose for variables should be meaningful so you or another programmer know what it is.

In Python the “=” symbol assigns the value on the right to the name on the left. The variable is created when a value is assigned to it. Here is how we can assign a number to a variable called age in Python:

Code:

```
# Defining a variable called age and assigning a number
to this variable

age = 36
```

You can hold “Shift” and press “Enter” to run this cell. Now the value of 36 is assigned to a memory location in the computer which is called age. If I want to see what is inside variable age, I can use a built-in function in Python called print( ). The values passed to the print function should be inside parentheses and they are called **arguments.** We can write:

Code:

```
# Using print function to print something on screen

print(age)
```

Output:

```
      36
```

Instead of numbers, we can also assign a series of characters or strings to a variable. Note that characters should be put inside single quotes or double quotes in Python.

Code:

```
# Defining a variable called first_name and assigning a
series of characters to this variable

first_name = 'yashar'
print(first_name)
```

Output:

```
      yashar
```

We can use print function to print two or more variables and some statements in between:

Code:

```
print('my first name is', first_name, 'and my age is',
age)
```

Output:

```
my first name is yashar and my age is 36
```

As you can see, the print function automatically puts a single space between items to separate them and wraps around to a new line at the end. If a variable doesn't exist yet, or if the name has been mis-spelled, Python reports an error.

Code:

```
print(last_name)
```

Output:

```
---------------------------------------------------------
------------------
NameError                             Traceback (most
recent call last)
<ipython-input-1-c1fbb4e96102> in <module>()
----> 1 print(last_name)

NameError: name 'last_name' is not defined
```

The last line in the error message is usually the most important one. Here the last line reads a NameError in which 'last_name' is not defined, which is exactly the source of the issue. We can also use variables in calculations as if they were values.

Code:

```
my_future_age = age + 10
print('my age in 10 years would be', my_future_age)
```

Output:

```
my age in 10 years would be 46
```

Exercise 4.1

Write a Python code that converts your height from meters to feet, and print it out.

In Python, we can also define multiple variables in a single line:

Code:

```
first_name, last_name = 'John', 'Reyes'
print('My first name is', first_name, 'and my last name
is', last_name)
```

Output:

```
My first name is John and my last name is Reyes
```

5. Data types in Python

Every value (data) in a Python program has a specific type. There are four main types of values in Python:

- Integer (int): represents positive or negative whole numbers like 4 or -12.
- Floating point number (float): represents real numbers like 3.14 or -2.5.
- Character string (usually called “string”, str): text.
- Boolean values like TRUE or FALSE.

In Python, we can use the built-in function “type( )” to find out what type a value has. Type works on variables as well but you should remember that the value has the type and the variable is just a label.

Code:

```
print(type(14))
```

Output:

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

Code:

```
print(type(first_name))
```

Output:

```
<class = 'str'>
```

Code:

```
print(type(3.4))
```

Output:

```
<class = 'float'>
```

Exercise 5.1

Define a variable called first_number and assign the value 3 to this variable. Define a second variable called second_number and assign the value 3.1 to this variable. Define a third variable called sum which should be the summation of first_number and second_number. Find out what is the type of this last variable and print it on the screen.

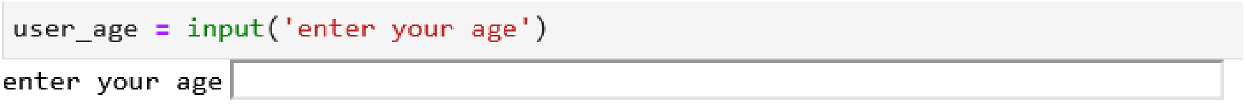
6. Taking input from users

In Python we can take values from users by using input( ) function. For example, if we want a user to enter their age and assign this value to a variable called user_age, we can write the following code:

Code:

```
user_age = input('enter your age')
```

The user will then have the opportunity to enter their age inside the box like the screenshot below, and then that value will be stored in the user_age variable. Note that this function will return a string. Therefore, if you need to use this data in any form of calculation, you need to first convert the string to an integer (or float) using int() built-in function in python. The details of variable type conversions will be provided in the next session.



```
user_age = input('enter your age')
enter your age
```

Figure 12: How to use the input function.

Exercise 6.1

Write a Python program that asks users to provide their first name, last name, and their age. Then the program should print their full name and their age in this format: *My name is [first name] [last name] and I am [age] years old.*

Exercise 6.2

Write a Python program that takes the two integer numbers from a user and calculates their sum.

Exercise 6.3

Write a Python code that asks a user to provide their height in centimeters and then convert it to feet, and print it out.

Exercise 6.4

Convert the code in Exercise 6.3 into a Python script and run the script in your terminal.

As a final note, we will explore an alternative way to print a string or contents of a variable on the output screen in Python, and it is called formatted string literals or f-strings for short. This feature was added to Python in version 3.6. The real benefit of using f-strings instead of print() is shown when you need to format strings using multiple values. Rather than doing multiple string concatenation, you can directly use the name of a variable or include an expression in the string. Being able to embed entire expressions inside of a string literal is a useful feature, and can also make for more clear and concise code. This will become very clear when you begin to write more code and the use cases for your code becomes non-trivial.

f-strings are string literals that have an *f* before the opening quotation mark. They can include Python expressions enclosed in curly braces. Python will replace those expressions with their resulting values. So, this behavior turns f-strings into a string interpolation tool.

Code:

```
name = "Yashar"
age = 36
f"Hello, {name}! You're {age} years old."
```

Output:

```
Hello, Yashar! You're 36 years old.
```

As you can see, the code is more readable and concise using the f-string syntax.

7. Functions and built-in functions in Python

A function is a block of code that performs a specific task and will only run when it is called. Suppose you need to create a program to create a blank box and then fill the box with some characters. You can create three functions to solve this problem: a function that creates a blank box, a function that reads the characters from users, and a function that fills the box with user characters. Therefore, we can divide a complex problem into smaller sections which makes our program easy to understand and reuse.

We have three different types of functions in Python: built-in functions, external-library functions, and user-defined functions. In this lesson, we want to focus on built-in functions in Python that are available to use in the base version of Python and do not need to be defined by you or imported via external libraries.

Note

- Three main types of functions in Python are built-in functions which exist in the base version of Python, external-library functions which come with the external libraries, and they are defined inside the library, and finally user-defined functions which can be defined by you.

We already used a few built-in functions in Python. For example, print() is a built-in function that will print the specified message on the screen, or other standard output device.

To use a function, we need to call it first. Any function that is called in Python should have parentheses after its name. An argument is a value inside the parentheses which will be passed into a function. print() for example can take zero or more arguments. print() with no arguments (zero arguments) prints a blank line.

Code:

```
print('first line')
print()
print('second line')
```

Output:

```
first line

second line
```

Every function call produces some specific result. If the function doesn't have a useful result to return, it usually returns the special value *None*. *None* is a Python object that stands in anytime there is no value.

Code:

```
Report = print('reporting')
print('result of the print is', Report)
```

Output:

```
reporting
result of the print is None
```

8. Type conversion built-in function

The process of converting a Python data type into another data type is known as type conversion. There are mainly two types of type conversion methods in Python: implicit type conversion and explicit type conversion. In Python, when the data type conversion takes place during interpretation or during the runtime, it’s called an implicit data type conversion. Python handles the implicit data type conversion, so we don’t have to explicitly convert the data type into another data type. See the below example for an implicit data type conversion:

Code:

```
first_number = 4
second_number = 4.25
sum_nums = first_number+second_number
print(sum_nums)
print(type(sum_nums))
```

Output:

```
8.25
<class ‘float’>
```

In the above example we have taken two variables, of integer and float data types, and added them. Further, we have declared another variable named ‘sum’ and stored the result of the addition in it. When we checked the data type of the sum variable, we could see that the data type of the sum variable had been automatically converted into the float data type by the Python interpreter. This is called implicit type conversion. The reason that the sum variable was converted into the float data type and not the integer data type is that if the compiler had converted it into the integer data type, then it would’ve had to remove the fractional part, which would have resulted in data loss. So, Python always converts smaller data types into larger data types to prevent the loss of data.

Explicit type conversion is also known as typecasting. In an explicit type conversion, the programmers clearly and explicitly define the type. For explicit type conversion, there are some built-in Python functions but here, we will focus on the main three built-in functions for data conversion:

int(x): converts x to an integer.
float(x): converts x to a float.
str(x): converts x to a string.

Note

- Type conversion is the process of converting a data type into another data type.
- Implicit type conversion is performed by a Python interpreter only.
- Explicit type conversion is performed by the user by explicitly using type conversion functions in the program code.

Code:

```
number = 4
string = '5.5'
sum_nums = number+string
print(sum_nums)
```

Output:

```
---------------------------------------------------------
-------------------
TypeError                                 Traceback
(most recent call last)
~\AppData\Local\Temp\ipykernel_14896\159005223.py in
<module>
      1 number = 4
      2 string = '5.5'
----> 3 sum = number+string
      4 print(sum)

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

As expected, Python produces an error in response to our code as it cannot perform a sum operation on an integer and a string. Here to fix the code, we need to first convert the string data into a float and then perform the mathematical sum.

Code:

```
number = 4
string = '5.5'
sum = number+float(string)
print(sum)
```

Output:

```
9.5
```

Now let’s look at another situation when we convert a number into a string and then use a summation:

Code:

```
number = 44
string = 'Years'
sum = str(number)+string
print(sum)
```

Output:

```
44Years
```

Exercise 8.1

Which of the following will return the floating point number 4.0? Note: there may be more than one right answer.

```
first = 2.0
second = "2"
```

```
third = "2.1"
```

1. first + float(second)
2. float(second) + float(third)
3. first + int(third)
4. first + int(float(third))
5. int(first) + int(float(third))
6. 2.0 * second

Exercise 8.2

Write a Python program that asks a user to enter their birth year, then calculate their age and print it on screen.

In addition to print(), input() and type(), there are a lot of built-in functions in Python which we can use to facilitate our computations. A few of commonly used statistical built-in functions in Python are: min() which can be used to find minimum (smallest) value, and max() which can be used to find maximum (largest) value. Note that min() and max() functions both work on character strings as well as numbers. To determine larger and smaller, Python uses (0-9, A-Z, a-z) to compare letters.

Code:

```
print(min(1,2,3))
```

Output:

```
1
```

Code:

```
print(max(1,2,3))
```

Output:

```
3
```

Code:

```
print(min('a', 'A'))
```

Output:

```
a
```

Note that the arguments of min() and max() functions should be things that can meaningfully be compared.

Code:

```
print(max(1, 'a'))
```

Output:

```
TypeError                                 Traceback (most
recent call last)
<ipython-input-52-3f049acf3762> in <module>
----> 1 print(max(1, 'a'))

TypeError: '>' not supported between instances of 'str'
and 'int'
```

In addition to min() and max(), we can use the round() function to round off a floating-point number.

Code:

```
print(round(2.78))
```

Output:

```
2
```

Note that in the round() function, we can specify the number of decimal places (n) we want in the value x by using the following format: round(x, n).

Code:

```
print(round(2.78, 1))
```

Output:

```
2.7
```

Exercise 8.3

Use the Python built-in functions to round the Pi constant (Pi=3.1415926) to its two decimal places.

9. Help in Python

In addition to using a search engine like Google, a large language model bot like ChatGPT or Gemini, or a social media platform/forum, there are two direct methods for seeking additional information about a Python function, class, or library. First, we can use the Python help() function to get the documentation of specified modules, classes, functions, variables etc. This method is generally used with the Python interpreter console to get details about python objects. Note that every built-in function has online documentation. For example, if we want to seek more information about round() function, we can type:

Code:

```
help(round)
```

Output:

```
Help on built-in function round in module builtins:

round(number, ndigits=None)
```

```
    Round a number to a given precision in decimal
digits.

    The return value is an integer if ndigits is omitted
or None.  Otherwise
    the return value has the same type as the number.
ndigits may be negative
```

Another way of seeking information in Jupyter Lab is by typing the function name in a cell with a question mark after it, and then run the cell.

Code:

```
round?
```

Exercise 9.1

Try to find the documentation on abs() function in Python. Describe when you should use this built-in function and what it does.

10. Comments in Python

As mentioned before, we can use comments to add documentation to programs by adding a hashtag (#) before the comment:

Code:

```
# This is a comment
```

11. Error messages in Python

As you do more and more programming, you will naturally encounter a lot of errors (or bugs). Understanding error messages in Python is important as it can help us to understand the source of error in our code faster and rectify the issue.

There are generally two different types of errors in Python: syntax error and runtime error. A syntax error happens when Python can't understand what you are saying. A run-time error happens when Python understands what you are saying, but runs into trouble when following your instructions.

Code:

```
print 'This is a sentence'
```

Output:

```
Traceback (most recent call last):
  In line 1 of the code you submitted:
    print 'This is a sentence'
                    ^
SyntaxError: Missing parentheses in call to 'print'. Did
you mean print('This is a sentence')?
```

In this example, we forgot to use the parenthesis that are required by the print() function. Python does not understand what we are trying to do and therefore, generates a syntax error.

Code:

```
print(sentence)
```

Output:

```
Traceback (most recent call last):
  In line 1 of the code you submitted:
    print(sentence)
NameError: name 'sentence' is not defined
```

In the last example, we forget to define the sentence variable. Python knows what you want it to do, but since no sentence has been defined, a name error occurs.

Exercise 11.1

Debug the following Python program by typing the code in Jupyter Lab and then reading the error message carefully:

```
my_ages = 53
remaining = 100 - my_age
print(remaining)
```

12. Lists in Python

Lists are used to store multiple items in a single variable. Lists are one of the built-in data types in Python used to store collections of data, together with Tuple and Dictionary, all with different qualities and usage. If we run an experiment and collect a physical quantity like temperature of water over time, then we may have multiple variables like temperature_001, temperature_002, temperature_003 etc. Doing calculations with a hundred variables called temperature_001, temperature_002, etc., would be at least as slow as doing them by hand. We use a list to store many values together contained within square brackets [...]. Remember values should be separated by commas:

Code:

```
temperatures = [18, 19, 20, 17.5, 22]
print('temperatures:', temperatures)
```

Output:

```
temperatures: [18, 19, 20, 17.5, 22]
```

We can also create list using a list constructor:

Code:

```
temperatures = list((18,19,20,17.5,22))
print('temperatures:', temperatures)
```

Output:

```
temperatures: [18, 19, 20, 17.5, 22]
```

It is also important to know how to initialize empty lists. In many situations, you can solve problems in data engineering by using an empty list, such as creating placeholders that will later be filled in with data.

Code:

```
Empty_list = []
```

We can use a built-in function len() to find the length of a list, or how many values are in a list:

Code:

```
temperatures = [18, 19, 20, 17.5, 22]
print('length:', len(temperatures))
```

Output:

```
length: 5
```

List items are indexed, the first item has index [0], the second item has index [1], the last item has index [-1] etc:

Code:

```
print('zeroth item of temperatures:', temperatures[0])
print('third item of temperatures:', temperatures[3])
print('the last item of temperatures:', temperatures[-1])
```

Output:

```
zeroth item of temperatures: 18
third item of temperatures: 17.5
The last item of temperatures: 22
```

A list can contain different data types:

Code:

```
list_example = ["abc", 34, True, 40, "male"]
```

List items are ordered, changeable, and allow duplicate values. Use an index expression on the left of assignment to replace a value:

Code:

```
temperatures[0] = 14.8
print('temperatures is now:', temperatures)
```

Output:

```
temperatures is now: [14.8, 19, 20, 17.5, 22]
```

Consider a Python list, in order to access a range of elements in a list, you need to slice a list. One way to do this is to use the simple slicing operator (:). With this operator, one can specify where to start the slicing, where to end, and specify the step. List slicing returns a new list from the existing list.

Code:

```
List_alphabets = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h',
'i']
print(List_alphabets[2:7])
```

Output:

```
['c', 'd', 'e', 'f', 'g']
```

Exercise 12.1

If start and stop are both non-negative integers, how long are the list values[start:stop]?

If you add new items to a list, the new items will be placed at the end of the list. Use *list_name.append* to add items to the end of a list:

Code:

```
primes = [2, 3, 5]
print('primes is initially:', primes)
primes.append(7)
print('primes has become:', primes)
```

Output:

```
primes is initially: [2, 3, 5]
primes has become: [2, 3, 5, 7]
```

Here, *append* is a method of lists. Like a function, but tied to a particular object. Use object_name.method_name to call methods. This deliberately resembles the way we refer to things in a library. We will meet other methods of lists as we go along. Use help(list) for a preview. *extend* is similar to append, but it allows you to combine two lists:

Code:

```
teen_primes = [11, 13, 17, 19]
middle_aged_primes = [37, 41, 43, 47]
print('primes is currently:', primes)
primes.extend(teen_primes)
print('primes has now become:', primes)
primes.append(middle_aged_primes)
print('primes has finally become:', primes)
```

Output:

```
primes is currently: [2, 3, 5, 7]
primes has now become: [2, 3, 5, 7, 11, 13, 17, 19]
primes has finally become: [2, 3, 5, 7, 11, 13, 17, 19,
[37, 41, 43, 47]]
```

Note that while extend maintains the flat structure of the list, appending a list to a list means the last element in primes will itself be a list, not an integer. Lists can contain values of any type; therefore, lists of lists are possible.

The count() method returns the number of elements with the specified value.

Code:

```
fruits = ['apple', 'cherry', 'banana', 'cherry']
number_of_cherries = fruits.count("cherry")
print(number_of_cherries)
```

Output:

```
2
```

Exercise 12.2

Write a Python code that returns the number of times the value 9 appears in the following list:

[1, 4, 2, 9, 7, 8, 9, 3, 1]

The reverse() method reverses the sorting order of the elements.

Code:

```
fruits = ['apple', 'banana', 'cherry']
fruits.reverse()
print (fruits)
```

Output:

```
['cherry', 'banana', 'apple']
```

The sort() method sorts the list ascending by default. You can also make a function to decide the sorting criteria(s) via this format: ***list*.sort(reverse=True|False, key=Function).** Here reverse=True will sort the list descending. Default is reverse=False. Key is also a function to specify the sorting criteria(s).

Code:

```
cars = ['Ford', 'BMW', 'Volvo']
cars.sort(reverse=True)
print(cars)
```

Output:

```
['Volvo', 'Ford', 'BMW']
```

Exercise 12.3

Look at the following list of prime numbers:
prime_numbers = [11, 3, 7, 5, 2]

(a) Sort them from smallest to largest.
(b) Sort them from largest to smallest.

Exercise 12.4

Sort the following list of strings based on their lengths:
text = ["abc", "wxyz", "gh", "a"]

The clear() method removes all the elements from a list.

Code:

```
cars = ['Ford', 'BMW', 'Volvo']
cars.clear()
print(cars)
```

Output:

```
[]
```

The pop() method removes the element at the specified position.

Code:

```
fruits = ['apple', 'banana', 'cherry']
fruits.pop(1)
print(fruits)
```

Output:

```
['apple', 'cherry']
```

We can also use *del* list_name[index] to remove an element from a list (in the previous example, 9 is not a prime number) and thus shorten it. *del* is not a function or a method, but a statement in the language:

Code:

```
primes = [2, 3, 5, 7, 9]
print('primes before removing last item:', primes)
del primes[4]
print('primes after removing last item:', primes)
```

Output:

```
primes before removing last item: [2, 3, 5, 7, 9]
primes after removing last item: [2, 3, 5, 7]
```

Exercise 12.5

Fill in the blanks so that the program below produces the output shown.

```
values = []
values.____(1)
values.____(3)
values.____(5)
print('first time:', values)
values = values[___:]
print('second time:', values)
```

```
Output:
first time: [1, 3, 5]
```

```
second time: [3, 5]
```

The characters (individual letters, numbers, and so on) in a string are ordered. For example, the string 'AB' is not the same as 'BA'. Because of this ordering, we can treat the string as a list of characters. Each position in the string (first, second, etc.) is given a number. This number is called an index. Therefore, character strings can be indexed like lists. Indices are numbered from 0. We can use the position's index in square brackets to get the character at that position. We can get single characters from a character string using indexes in square brackets:

Code:

```
element = 'carbon'
print('zeroth character:', element[0])
print('third character:', element[3])
```

Output:

```
zeroth character: c
third character: b
```

Note that character strings are immutable (unchangeable). Cannot change the characters in a string after it has been created. Immutable means the values can't be changed after creation. In contrast, lists are mutable which means they can be modified in place. Python considers the string to be a single value with parts, not a collection of values.

Code:

```
element = 'carbon'
element[0] = 'c'
```

Output:

```
TypeError: 'str' object does not support item assignment
```

Python reports an IndexError if we attempt to access a value that doesn’t exist.

Code:

```
print('99th element of element is:', element[99])
```

Output:

```
IndexError: string index out of range
```

A part of a string is called a substring. A substring can be as short as a single character. An item in a list is called an element. Whenever we treat a string as if it were a list, the string’s elements are its individual characters. A slice is a part of a string (or, more generally, a part of any list-like thing). We take a slice with the notation [start:stop], where start is the integer index of the first element we want and stop is the integer index of the element just after the last element we want. The difference between stop and start is the slice’s length. Taking a slice does not change the contents of the original string. Instead, taking a slice returns a copy of part of the original string.

Code:

```
atom_name = 'sodium'
print(atom_name[0:3])
```

Output:

```
sod
```

Exercise 12.6

What does the following program print?

```
atom_name = 'oxygen'
print('atom_name[1:3] is:', atom_name[1:3])
```

Exercise 12.7

Given the following string:

```
star_name = "Alpha Orionis")
```

What would these expressions return?

1. star_name[2:8]
2. star_name[11:] (without a value after the colon)
3. star_name[:4] (without a value before the colon)
4. star_name[:] (just a colon)
5. star_name[11:-3]
6. star_name[-5:-3]
7. What happens when you choose a stop value which is out of range? (i.e., try star_name[0:20] or star_name[:103])

13. Tuples

Tuples are used to store multiple items in a single variable. Tuple is one of the main built-in data types in Python used to store collections of data like a list. A tuple is a collection which is ordered and unchangeable. Tuple items much like list items are indexed, the first item has index [0], the second item has index [1] etc. Since tuples are indexed, they can have items with the same value. Tuples are also ordered which means that the items have a defined order and that order will not change.

Code:

```
tuple_example = ('apple', 'tomato', 'banana')
print(tuple_example)
```

Output:

```
('apple', 'tomato', 'banana')
```

Unlike lists, tuples are unchangeable which means that we cannot change, add or remove items after the tuple has been created.

Code:

```
tuple_example = ('apple', 'tomato', 'banana')
Tuple_example[0] = 'cherry'
```

Output:

```
TypeError: 'tuple' object does not support item
assignment
```

Note

- Tuples are written with round brackets.
- Tuples are ordered.
- Unlike lists, tuple items are unchangeable (immutable).

To determine how many items a tuple has, we can use the len() function.

Code:

```
tuple_example = ("apple", "tomato", "banana", "apple")
print(len(tuple_example))
```

Output:

```
4
```

Finally, a tuple can contain different data types.

Code:

```
tuple_example = ('apple', 2, 5.2)
print(tuple_example[0])
```

Output:

```
'apple'
```

Exercise 13.1

Write a Python code to get the 4th element from the last element of the following tuple:

```
sample_tuple = ("w", 3, "r", "e", "s", "o", "u", "r", "c", "e")
```

14. Dictionaries

Dictionaries are used to store data values in key:value pairs. A dictionary is a collection which is ordered, changeable and does not allow duplicates. (NOTE: Python dictionaries were unordered up until version 3.6, for version 3.7 and later they are now ordered). Dictionaries are written with curly brackets, and have keys and values:

Code:

```
translation = {'one': 'first', 'two': 'second'}
print(translation['one'])
```

Output:

```
'first'
```

Dictionaries work a lot like lists - except that you index them with keys. You can think about a key as a name or unique identifier for the value it corresponds to. To add an item to the dictionary we assign a value to a new key:

Code:

```
translation['three'] = 'third'
print(translation)
```

Output:

```
{'one': 'first', 'two': 'second', 'three': 'third'}
```

Exercise 14.1

Write a Python script to add a key to a dictionary.

Sample Dictionary : {0: 10, 1: 20}
Expected Result : {0: 10, 1: 20, 2: 30}

15. Conditionals

Python supports the usual logical conditions from mathematics:

Equals: a == b
Not Equals: a != b
Less than: a < b
Less than or equal to: a <= b
Greater than: a > b
Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops. An "if statement" is written by using the *if* keyword:

Code:

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

Output:

```
b is greater than a
```

In this example we use two variables, a and b, which are used as part of the if statement to test whether b is greater than a. As a is 33, and b is 200, we know that 200 is greater than 33, and so we print to screen that "b is greater than a".

An if statement (more properly called a conditional statement) controls whether some block of code is executed or not. Structure is similar to a for statement, as the first line opens with if and ends with a colon, and the body containing one or more statements is indented (usually by 4 spaces).

Note

- An if statement (more properly called a conditional statement) controls whether some block of code is executed or not.
- Structure is similar to a for statement, as the first line opens with if and ends with a colon, and the body containing one or more statements is indented (usually by 4 spaces).

Code:

```
mass = 3.54
if mass > 3.0:
      print(mass, 'is large')

mass = 2.07
if mass > 3.0:
    print (mass, 'is large')
```

Output:

```
3.54 is large
```

Note that Python steps through the branches of the conditional in order, testing each in turn and therefore, ordering matters:

Code:

```
grade = 85
if grade >= 70:
    print('grade is C')
elif grade >= 80:
    print('grade is B')
elif grade >= 90:
    print('grade is A')
```

Output:

```
grade is C
```

You can have *if* statements inside *if* statements, this is called nested *if* statements:

Code:

```
# Nested if
x = 41

if x > 10:
```

```
print("Above ten,")
if x > 20:
print("and also above 20!")
else:
print("but not above 20.")
```

Output:

```
Above ten,
and also above 20!
```

Exercise 15.1

What does this program print?

```
pressure = 71.9
if pressure > 50.0:
     pressure = 25.0
elif pressure <= 50.0:
     pressure = 0.0
print(pressure)
```

16. Loops in Python

The two types of loops in Python are "for" loops and "while" loops. "for" loops are generally more common so we will elaborate on those first. A *for* loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string). Doing calculations on the values in a list or tuple one by one is as painful as working with pressure_001, pressure_002, etc. A for loop tells Python to execute some statements once for each value in a list, a character string, or some other collection. It tells Python that for each thing in this group, do these operations.

Code:

```
for number in [1, 2, 6]:
    print(number)
```

This for loop is equivalent to:

Code:

```
print(1)
print(2)
print(6)
```

and the output would be equal to:

Output:

```
1
2
6
```

A for loop is made up of a collection, a loop variable, and a body. In the above example, [1, 2, 6] is the collection, number is the loop variable, and print() is the body. The loop variable, number in this example, is what changes for each iteration of the loop or the current thing that is being counted. Note that the colon at the end of the first line signals the start of a block of statements. Python uses indentation rather than {} or begin/end to show nesting. Any consistent indentation is legal, but almost everyone uses four spaces.

Code:

```
for number in [1, 2, 6]: # i will add a different example here
    print(number)
```

Output:

```
1
2
6
```

Similar to all variables, loop variables are created on demand and their name can be anything.

Code:

```
for kitten in [1, 2, 6]:
print(kitten)
```

Output:

```
IndentationError: expected an indented block
```

The body of a loop can contain many statements:

Code:

```
primes = [2, 3, 5]
for p in primes:
      squared = p ** 2
      cubed = p ** 3
      print(p, squared, cubed)
```

Output:

```
2 4 8
3 9 27
5 25 125
```

The built-in function range() produces a sequence of numbers. The numbers are produced on demand to make looping over large ranges more efficient (it's not a list!). It means range(N) produces the numbers 0...N-1.

Code:

```
print('a range is not a list: range(0, 3)')
for number in range(0, 3):
     print(number)
```

Output:

```
a range is not a list: range(0, 3)
0
1
2
```

A common application of a for loop is for accumulation. In this application, we usually initialize an accumulator variable to zero, the empty string, or the empty list, and then update the variable with values from a collection:

Code:

```
# Sum the first 10 integers.
total = 0
for number in range(10):
   total = total + (number + 1)
print(total)
```

Output:

```
55
```

Here read total = total + (number + 1) as add 1 to the current value of the loop variable number, add that to the current value of the accumulator variable total, and finally assign that to total, replacing the current value.

With the break statement, we can stop the loop before it has looped through all the items:

Code:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
      print(x)
      if x == "banana":
      break
```

Output:

```
apple
banana
```

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

Code:

```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```

Output:

```
0
1
2
3
4
5
Finally finished!
```

Finally, a nested loop is a loop inside a loop. The "inner loop" will be executed one time for each iteration of the "outer loop":

Code:

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
     for y in fruits:
     print(x, y)
```

Output:

```
red apple
red banana
red cherry
big apple
big banana
big cherry
tasty apple
tasty banana
tasty cherry
```

One last way to use a for loop is by "enumerating" through the list. This is similar to the example `for x in fruits` but if you wanted to also access the index number along with the item x. The syntax is described below.

Code:

```
# general syntax
# for index,item in enumerate(array):

provinces = ['nova scotia','newfoundland and labrador',
           'ontario','quebec']

for idx,item in enumerate(provinces):
     print(idx,item)
```

Output:

```
0 nova scotia
1 newfoundland and labrador
2 ontario
3 quebec
```

Exercise 16.1

Reorder and properly indent the lines of code below so that they print a list with the cumulative sum of data. The result should be [1, 3, 5, 10].

```
cumulative.append(total)
for number in data:
cumulative = []
total = total + number
total = 0
print(cumulative)
data = [1,2,2,5]
```

Exercise 16.2

Fill in the blanks in each of the programs below to produce the indicated result.

```
# Total length of the strings in the list: ["red", "green", "blue"]
=> 12
total = 0
for word in ["red", "green", "blue"]:
    ____ = ____ + len(word)
print(total)
```

Exercise 16.3

Create an acronym: Starting from the list ["red", "green", "blue"], create the acronym "RGB" using a for loop.

Hint: You may need to use a string method *.upper()* to properly format the acronym.

In addition to "for" loops, there are also "while" loops. These will run based on a conditional, as long as those conditional returns "True". Be careful that you do not create an infinite loop that is never ending!

Code:

```
total = 0

while total < 10:
     print(f"Current total: {total}")
     total += 3.14

print(f"Final total: {total}")
```

Output:

```
3.14
6.28
9.42
12.56
```

17. Combining Conditionals with Loops

Combining conditionals while looping through data is a very powerful concept.

Conditionals are often used inside loops:

Code:

```
masses = [3.54, 2.07, 9.22, 1.86, 1.71]
for m in masses:
     if m > 3.0:
     print(m, 'is large')
```

Output:

```
3.54 is large
9.22 is large
```

We can use *else* to execute a block of code when an *if* condition is not true:

Code:

```
masses = [3.54, 2.07, 9.22, 1.86, 1.71]
for m in masses:
     if m > 3.0:
```

```
    print(m, 'is large')
    else:
    print(m, 'is small')
```

Output:

```
3.54 is large
2.07 is small
9.22 is large
1.86 is small
1.71 is small
```

We can also use *elif* to specify additional conditions. The *elif* keyword is Python's way of saying "if the previous conditions were not true, then try this condition":

Code:

```
masses = [3.54, 2.07, 9.22, 1.86, 1.71]
for m in masses:
    if m > 9.0:
    print(m, 'is HUGE')
    elif m > 3.0:
    print(m, 'is large')
    else:
    print(m, 'is small')
```

Output:

```
3.54 is large
2.07 is small
9.22 is HUGE
1.86 is small
1.71 is small
```

18. Writing a Function

Break programs down into functions to make them easier to understand and also enables re-use of the same functions. Defining a section of code as a function in Python is done using the *def* keyword. For example a function that takes two arguments and returns their sum can be defined as:

Code:

```
def add_function(a, b):
    result = a + b
    return(result)

z = add_function(20, 22)
print(z)
```

Output:

```
42
```

Therefore, we begin the definition of a new function with *def*, followed by the name of the function (add_function here). Then function parameters or arguments in parentheses (a, b here). We should use empty parentheses if the function doesn't take any inputs. Then a colon, finally an indented block of code.

Code:

```
def print_greeting():
    print('Hello!')
```

Note that defining a function does not run it and we have to call the function to execute the code it contains:

Code:

```
print_greeting()
```

Output:

```
Hello!
```

Also note that arguments in a function call are matched to its defined parameters:

Code:

```
def print_date(year, month, day):
      joined = str(year) + '/' + str(month) + '/' + 
str(day)
      print(joined)

print_date(1871, 3, 19)
```

Output:

```
1871/3/19
```

Alternatively, we can name the arguments when we call the function, which allows us to specify them in any order and adds clarity to the call site; otherwise as one is reading the code they might forget if the second argument is the month or the day for example.

Code:

```
print_date(month=3, day=19, year=1871)
```

Output:

```
1871/3/19
```

Functions may return a result to their caller using *return* command:

Code:

```
def average(values):
    if len(values) == 0:
    return None
    return sum(values) / len(values)
a = average([1, 3, 4])
print('average of actual values:', a)
```

Output:

```
average of actual values: 2.6666666666666665
```

Remember: every function returns something. A function that doesn't explicitly return a value automatically returns *None*:

Code:

```
result = print_date(1871, 3, 19)
print('result of call is:', result)
```

Output:

```
1871/3/19
result of call is: None
```

Exercise 18.1

What does the following program print?

```
def report(pressure):
    print('pressure is', pressure)
print('calling', report, 22.5)
```

One last very important concept involving functions in Python is "recursion". Recursion is when you call functions within other functions. Doing this is very powerful as you can write complex code in a much more simple and easy to read way.

Code:

```
# define functions
def equation(k):
      output = 4 * (-1)**k / (2*k + 1)
      return(output)

# this function recursively uses the previous function
def summation(n):
      X = 0
      for idx in range(n):
      X += equation(idx)
      return(X)

# call the functions
for N in [1,10,100,1000,10000,100000]:
      y = summation(N)
      print(f'for {N} terms, summation = {y}')
```

Output:

```
for 1 terms, summation = 4.0
for 10 terms, summation = 3.0418396189294032
for 100 terms, summation = 3.1315929035585537
for 1000 terms, summation = 3.140592653839794
for 10000 terms, summation = 3.1414926535900345
for 100000 terms, summation = 3.1415826535897198
```

Key Points

- Python commands can be executed in interactive and scripting modes.
- You can run Python commands in terminal, or via various IDEs like PyCharm, Spyder, or JupyterLab.
- **print** can be used to display the output in Python.
- There are four main types of variables in Python: Integer, Float, String and Boolean.
- To define a variable, we can use **=** command.
- **type** can be used to identify various data types (or variable types).
- **input** can be used to ask for the user's input.
- using f-strings can make the code more readable and concise.
- Python functions can be user-defined or built-in, or come from an external library.
- We can convert between various types of data with built-in type conversion functions.
- To ask for help in Python, we can use *help()* or *?* in front of the command.
- To add comments to the code, we can use a hashtag (#) before the comment.
- There are two types of errors in Python in general: syntax error and runtime error.
- Lists are used to store multiple items in a single variable.
- Lists are mutable (changeable).
- Character strings can be indexed like lists.
- Tuples are similar to lists but they are immutable (unchangeable).
- Dictionaries are used to store data values in *key:value* pairs.
- A *for* loop is made up of a collection, a loop variable, and a body.
- An *if* statement (more properly called a conditional statement) controls whether some block of code is executed or not.
- Defining a section of code as a function in Python is done using the *def* keyword.
- Functions can be called from other functions, this is called *recursion*

References and Further Resources:

- How to run Python scripts - Knowledge Hut
- Python variables - Official Documentation
- Python scripting vs interactive modes- W3Schools
- Python - Carpentries
- Python - Official Documentation
- Python Functions - W3Schools
- Python List Methods - W3Schools
- Python Functions - W3Schools
- Python For Loops - W3Schools
- Python Conditionals - W3Schools
- Python Dictionaries - W3Schools