Getting Started With Jupyter Notebook for Python

In the following tutorial, you will be guided through the process of installing Jupyter Notebook. Furthermore, we'll explore the basic functionality of Jupyter Notebook and you'll be able to try out first examples.

Jupyter Notebook is a web application that allows you to create and share documents that contain:

- live code (e.g. Python code)
- visualizations
- explanatory text (written in markdown syntax)

Jupyter Notebook is great for the following use cases:

- learn and try out Python
- data processing / transformation
- numeric simulation
- statistical modeling
- machine learning

Let's get started and install Jupyter Notebook on your computer ... The first step to get started is to visit the project's website at http://www.jupyter.org (http://www.jupyter.org):

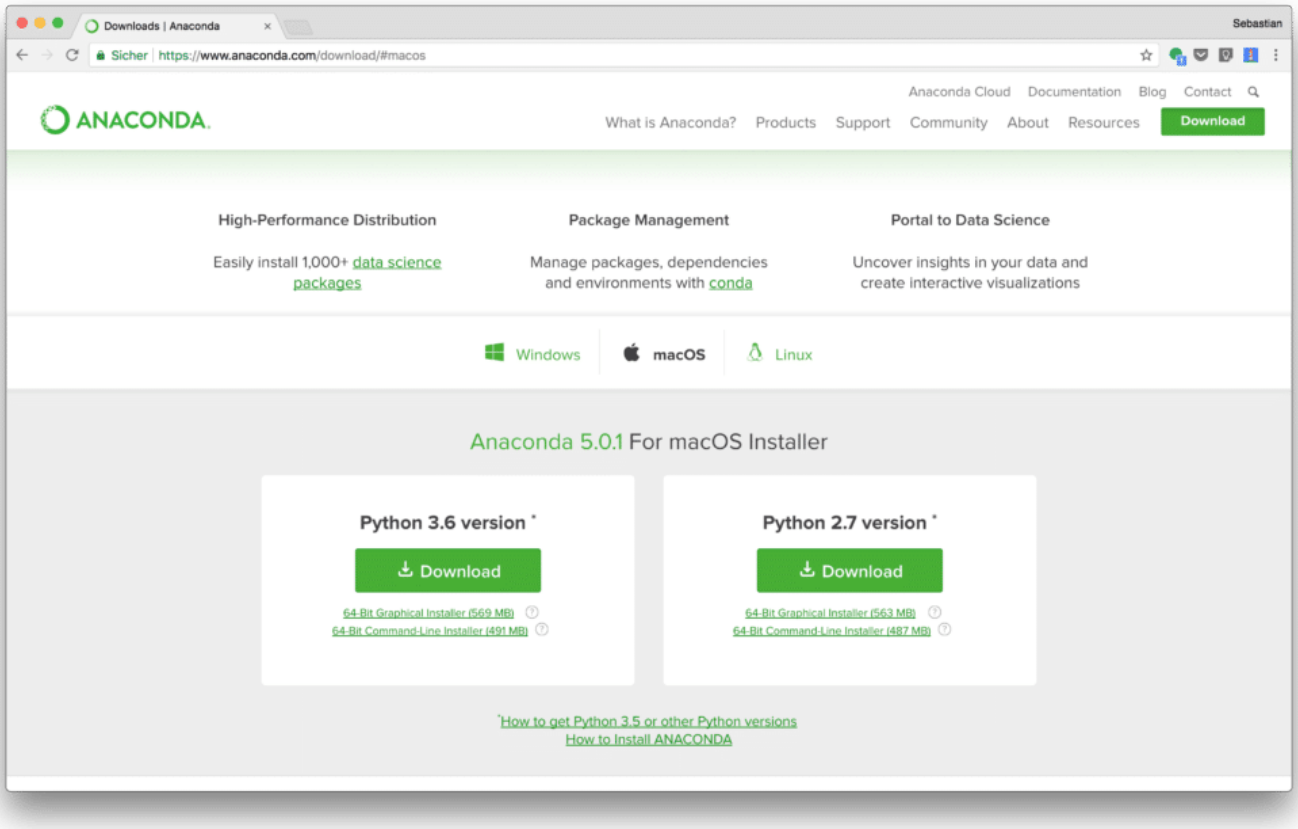
If you scroll down, you'll find two options:

- Try it in your browser
- Install the Notebook

With the first option "Try it in your browser", you can access a hosted version of Jupyter Notebook. This will get you direct access without needing to install it on your computer. The second option "Install the Notebook" will take you to another page which gives you detailed instruction for the installation. There are two different ways:

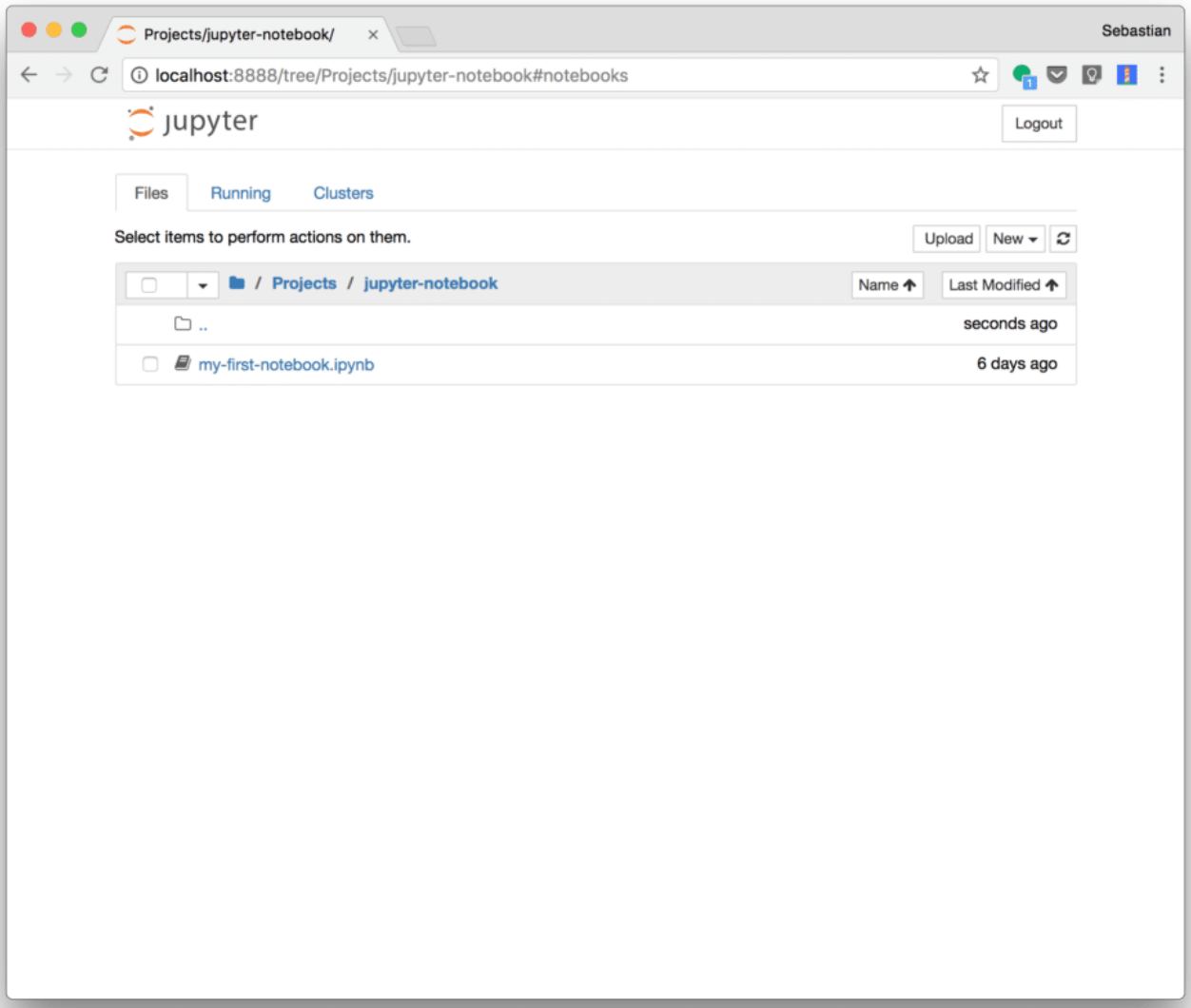
- Installing Jupyter Notebook by using the Python's package manager pip
- Installing Jupyter Notebook by installing the Anaconda distribution

Especially if you're new to Python and would like to set up your development environment from scratch using the Anaconda distribution is a great choice. If you follow the link (https://www.anaconda.com/download/ (https://www.anaconda.com/download/)) to the Anaconda download page you can choose between installers for Windows, macOS, and Linux:



Download and execute the installer of your choice. Having installed the Anaconda distribution we can now start Jupyter Notebook by clicking the Jupyter notebook icon in the Anaconda GUI, or starting Jupyter from the command prompt.

The web server is started and the Jupyter Notebook application is opened in your default browser automatically. You should be able to see a browser output, which is similar to the following screenshot:



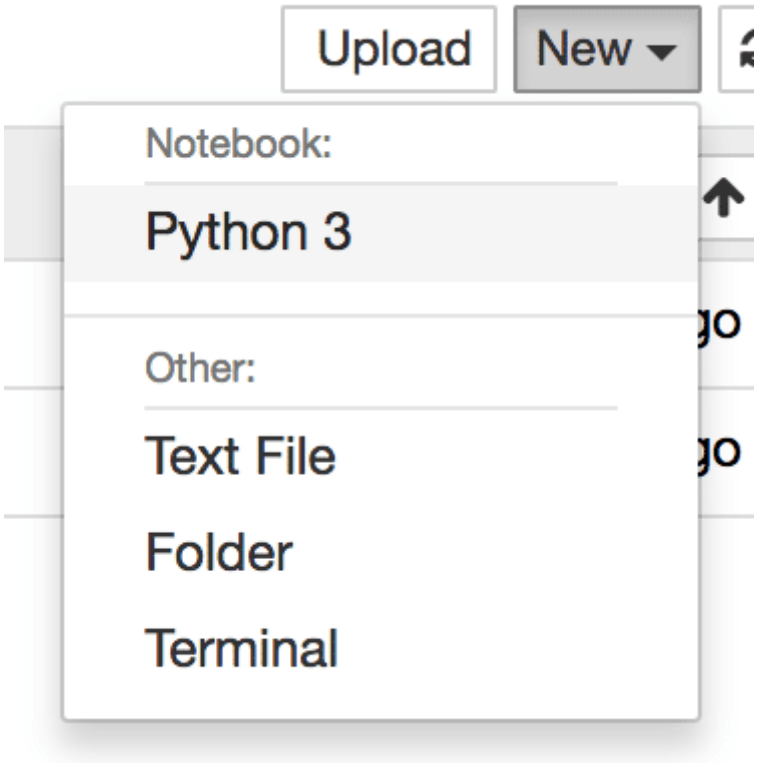
As you can see the user interface of Jupyter Notebook is split up into three sections (tabs):

- Files
- Running

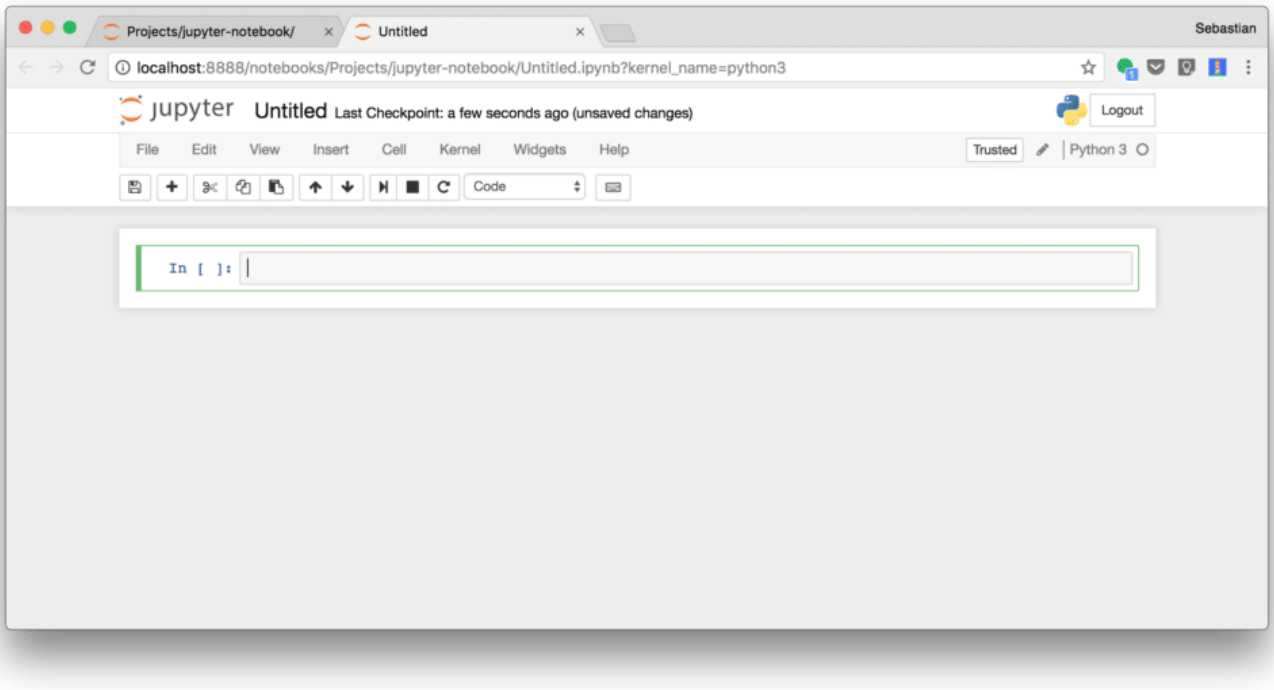
The default view is the Files tab from where you can open or create notebooks.

Creating A New Notebook

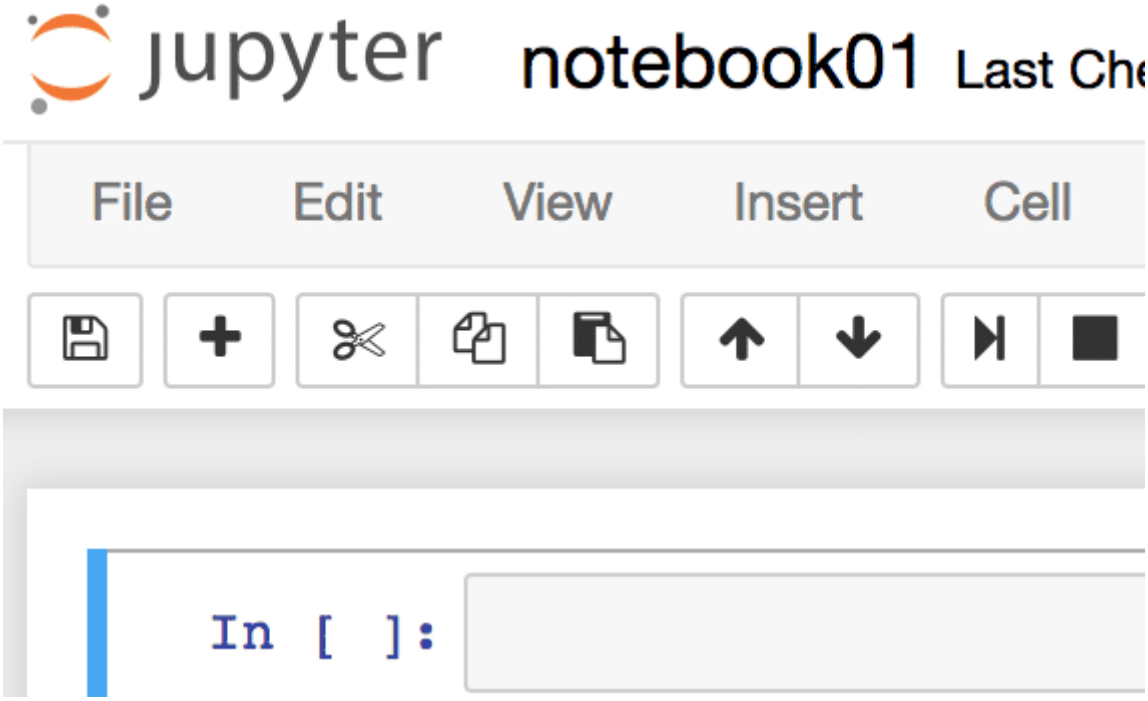
Creating a new Jupyter Notebook is easy. Just use the New dropdown menu and you'll see the following options:



Select option Python 3 to open a new Jupyter Notebook for Python. The notebook is created and you should be able to see something similar to:



The notebook is created but still untitled. By clicking into the text “Untitled” on the top, you can give it a name. By giving it a name the notebook will also be saved as a file of the same name with extension .ipynb. E.g. name the notebook notebook01:

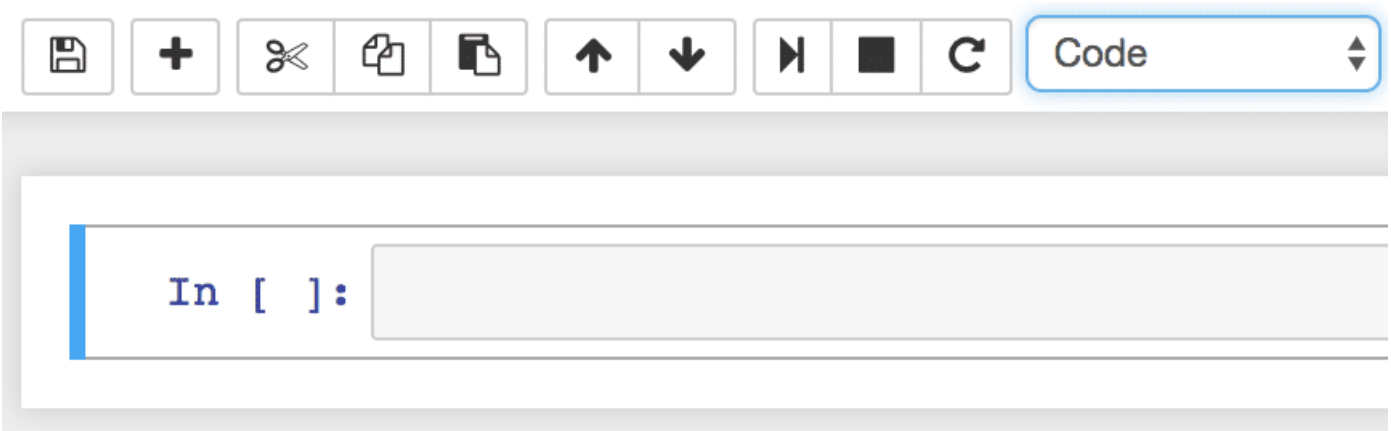


Switching back to the Files tab you'll be able to see a new file notebook01.ipynb:

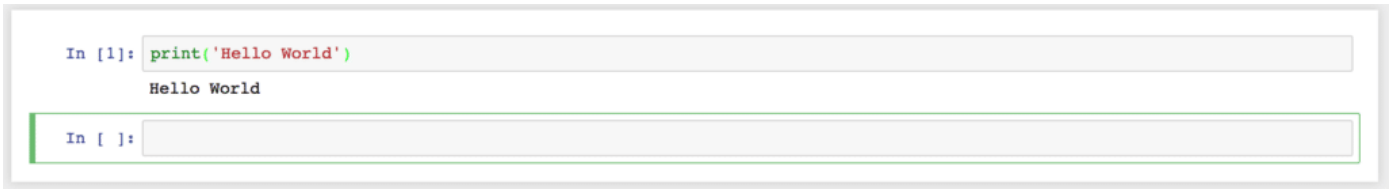
Because this notebook file is opened right now the file is marked with status Running. From here you can decided to shutdown this notebook by clicking on button Shutdown. However before shutting down the notebook let's switch back to the notebook view and try out a few things to get familiar with the notebook concept.

Working with the Notebook

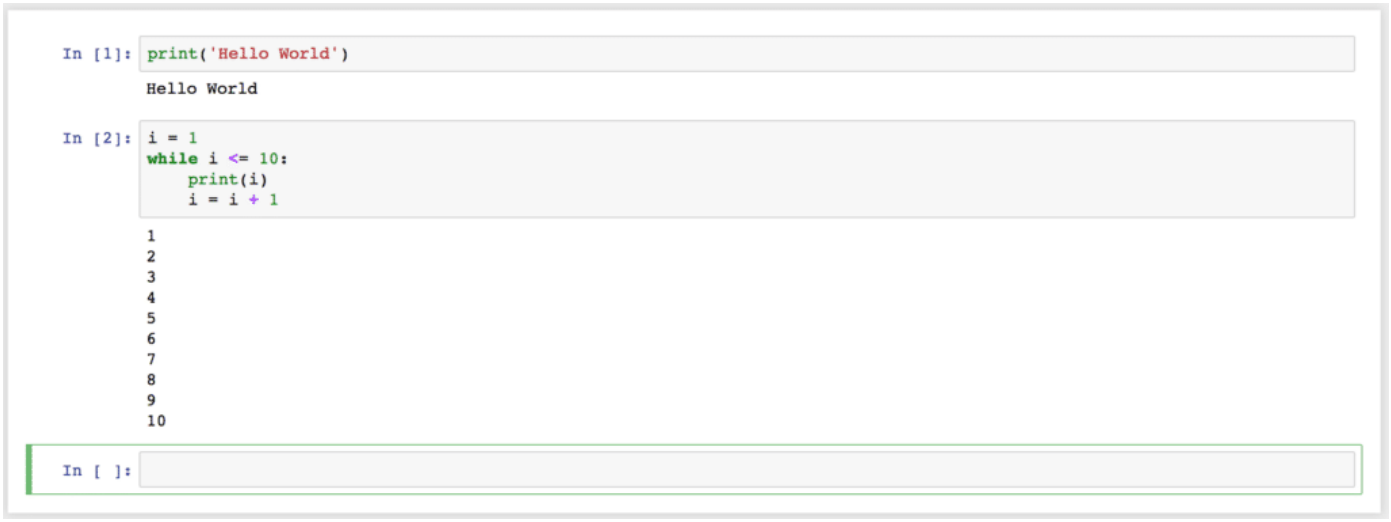
The notebook itself consists of cells. A first empty cell is already available after having created the new notebook:



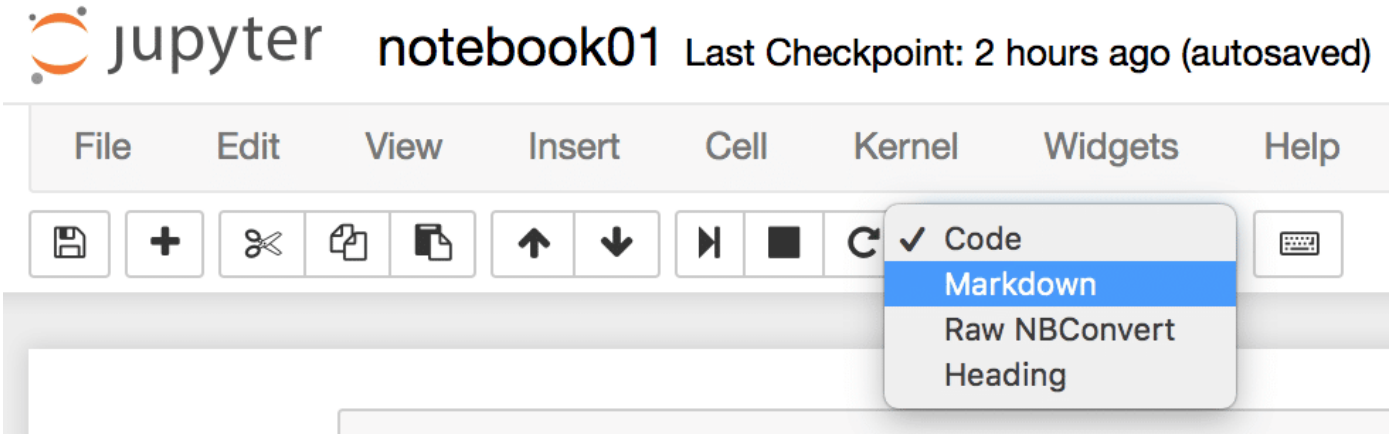
This cell is of type "Code" and you can start typing in Python code directly. Executing code in this cell can be done by either clicking on the run cell button or hitting Shift + Return keys:



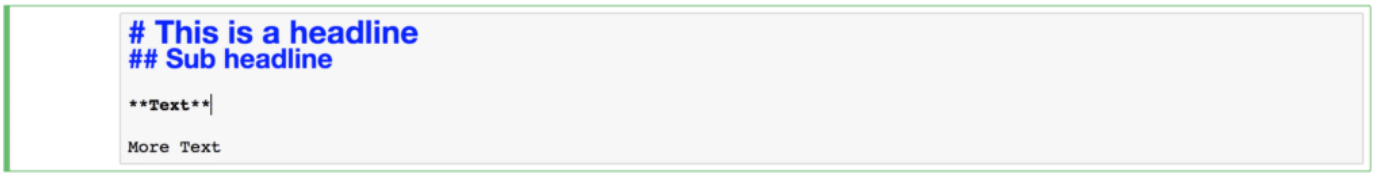
The resulting output becomes visible right underneath the cell. The next empty code cell is created automatically and you can continue to add further code to that cell. Just another example:



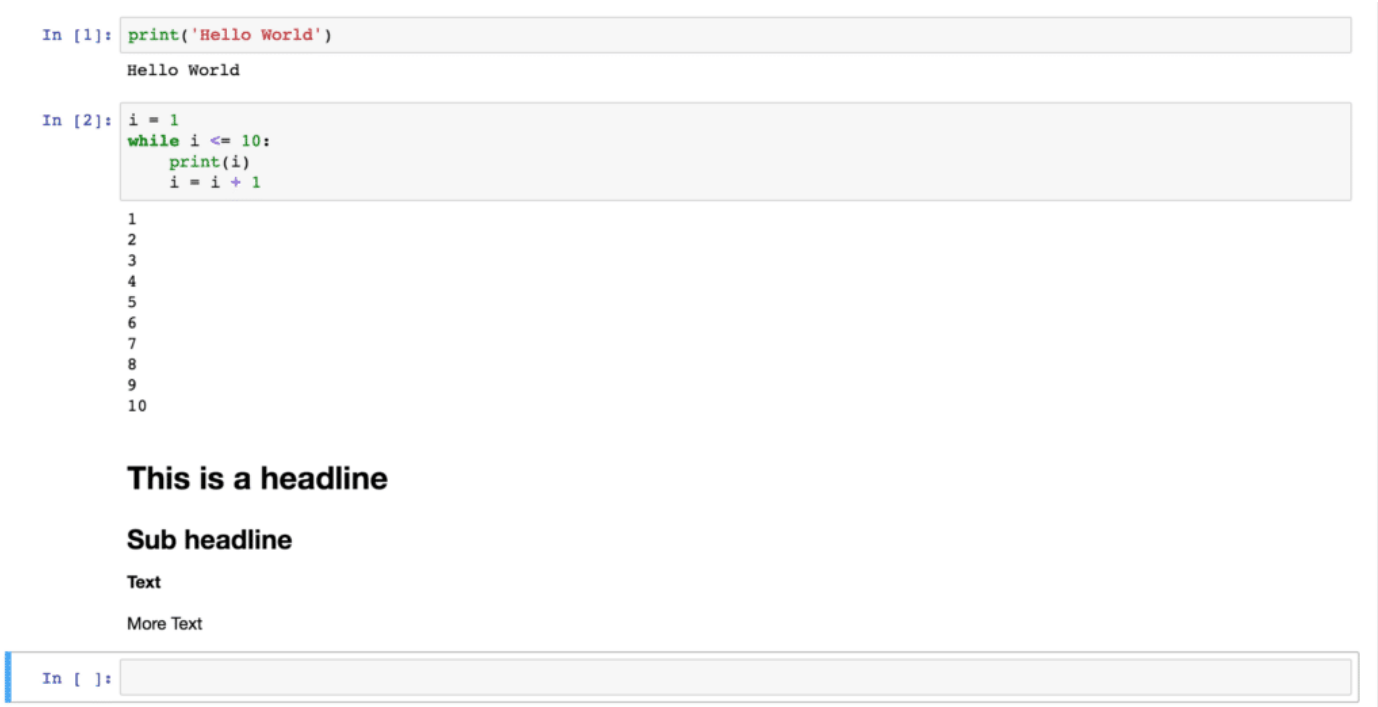
You can change the cell type from Code to Markdown to include explanatory text in your notebook. To change the type you can use the dropdown input control:



Once switched the type to Markdown you can start typing in markdown code:



After having entered the markdown code you can compile the cell by hitting Shift + Return once again. The markdown editor cell is then replaced with the output:



If you want to change the markdown code again you can simply click into the compiled result and the editor mode opens again.

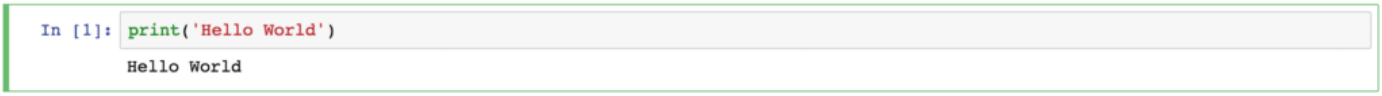
Edit And Command Mode

If a cell is active, two modes distinguished:

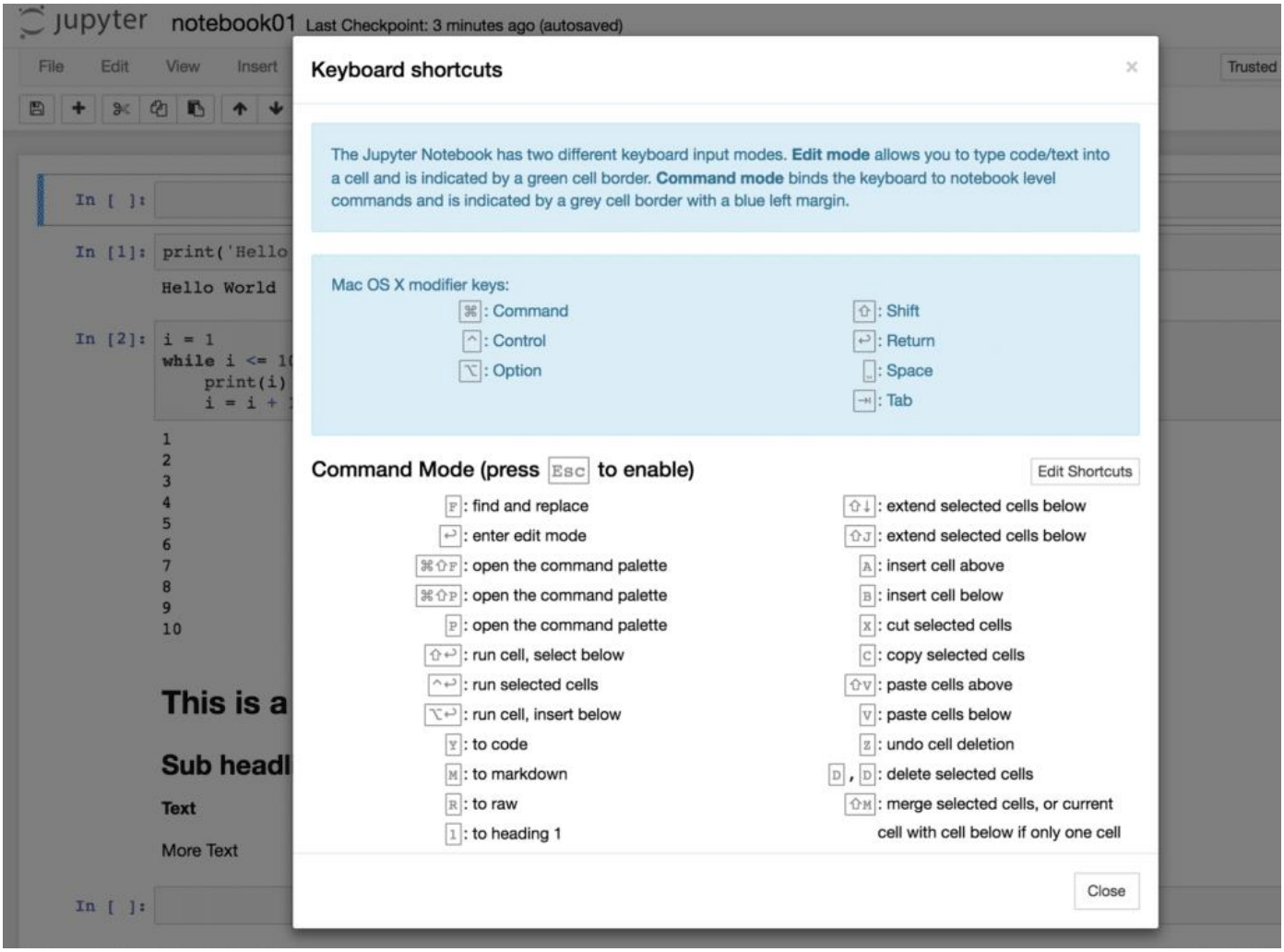
- edit mode
- command mode

If you just click in one cell the cell is opened in command mode which is indicated by a blue border on the left:

The edit mode is entered if you click into the code area of that cell. This mode is indicated by a green border on the left side of the cell:



If you'd like to leave edit mode and return to command mode again you just need to hit ESC. To get an overview of functions which are available in command and in edit mode you can open up the overview of key shortcuts by using menu entry Help → Keyboard Shortcuts:

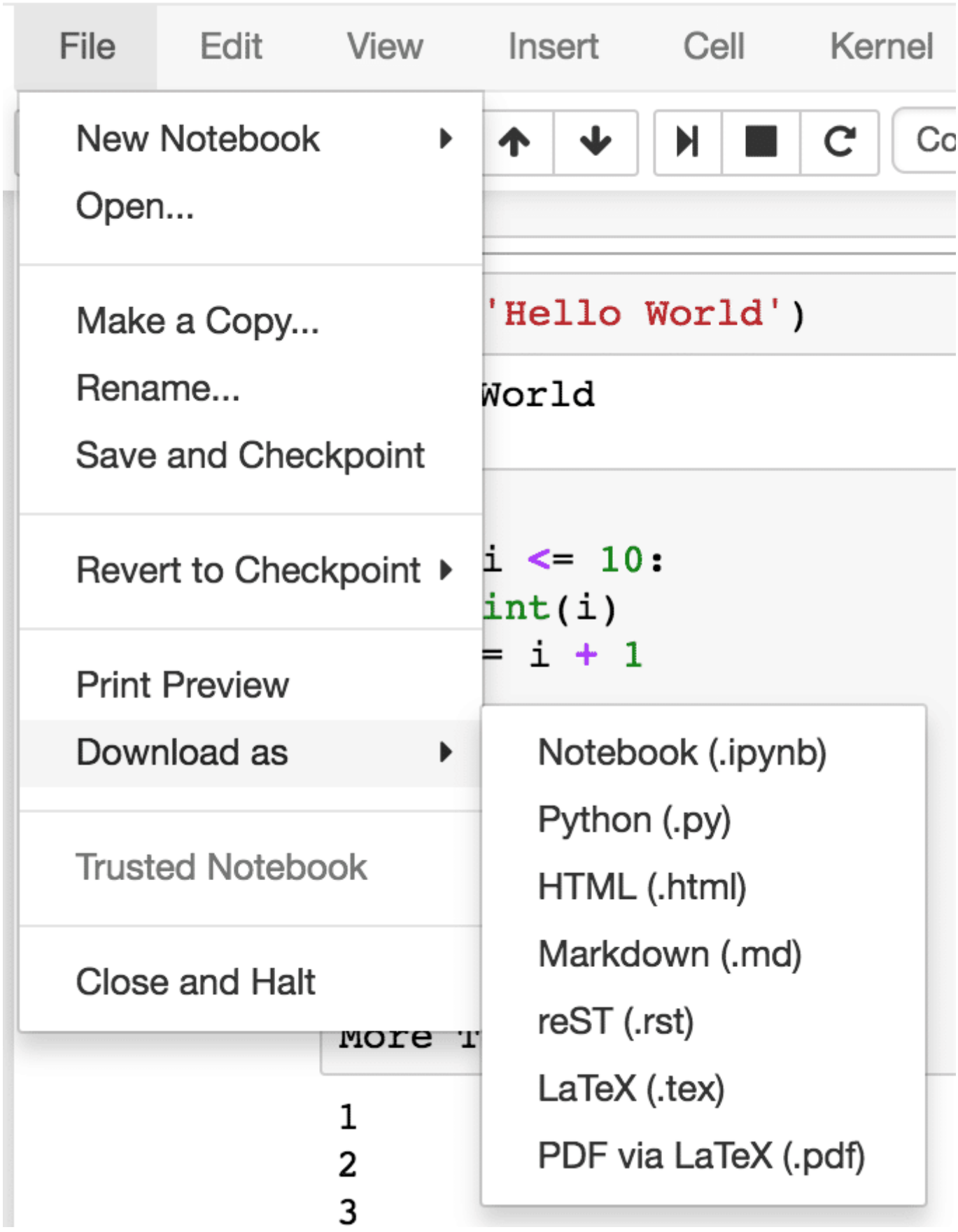


Checkpoints

Another cool function of Jupyter Notebook is the ability to create checkpoint. By creating a checkpoint you're storing the current state of the notebook so that you can later on go back to this checkpoint and revert changes which have been made to the notebook in the meantime. To create a new checkpoint for your notebook select menu item Save and Checkpoint from the File menu. The checkpoint is created and the notebook file is saved. If you want to go back to that checkpoint at a later point in time you need to select the corresponding checkpoint entry from menu File → Revert to Checkpoint.

Exporting The Notebook

Jupyter Notebook gives you several options to export your notebook. Those options can be found in menu File → Download as:



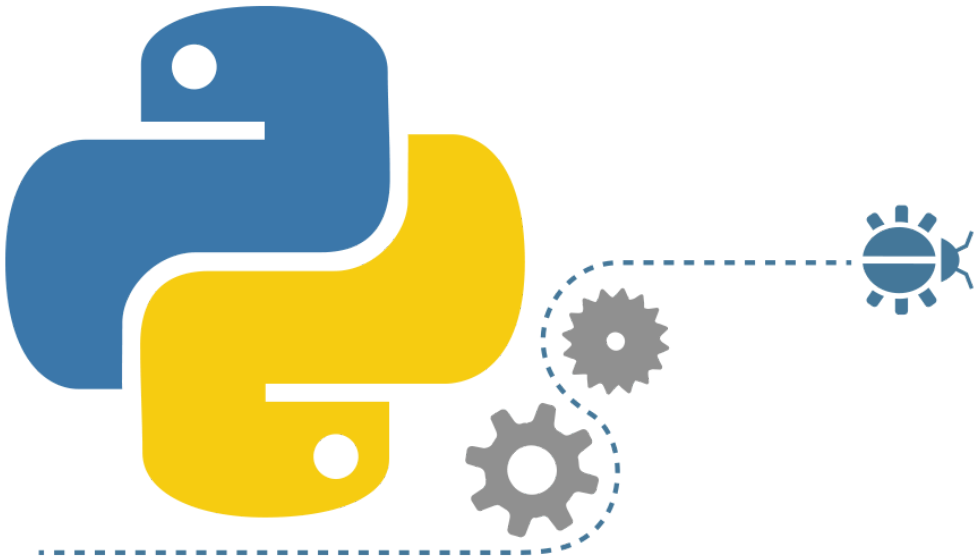
In [1]:

```
print("Hello World")
```

```
Hello World
```

In the next section (Programming in Python), you'll learn some basic concepts about proramming in Python.

PROGRAMMING IN PYTHON



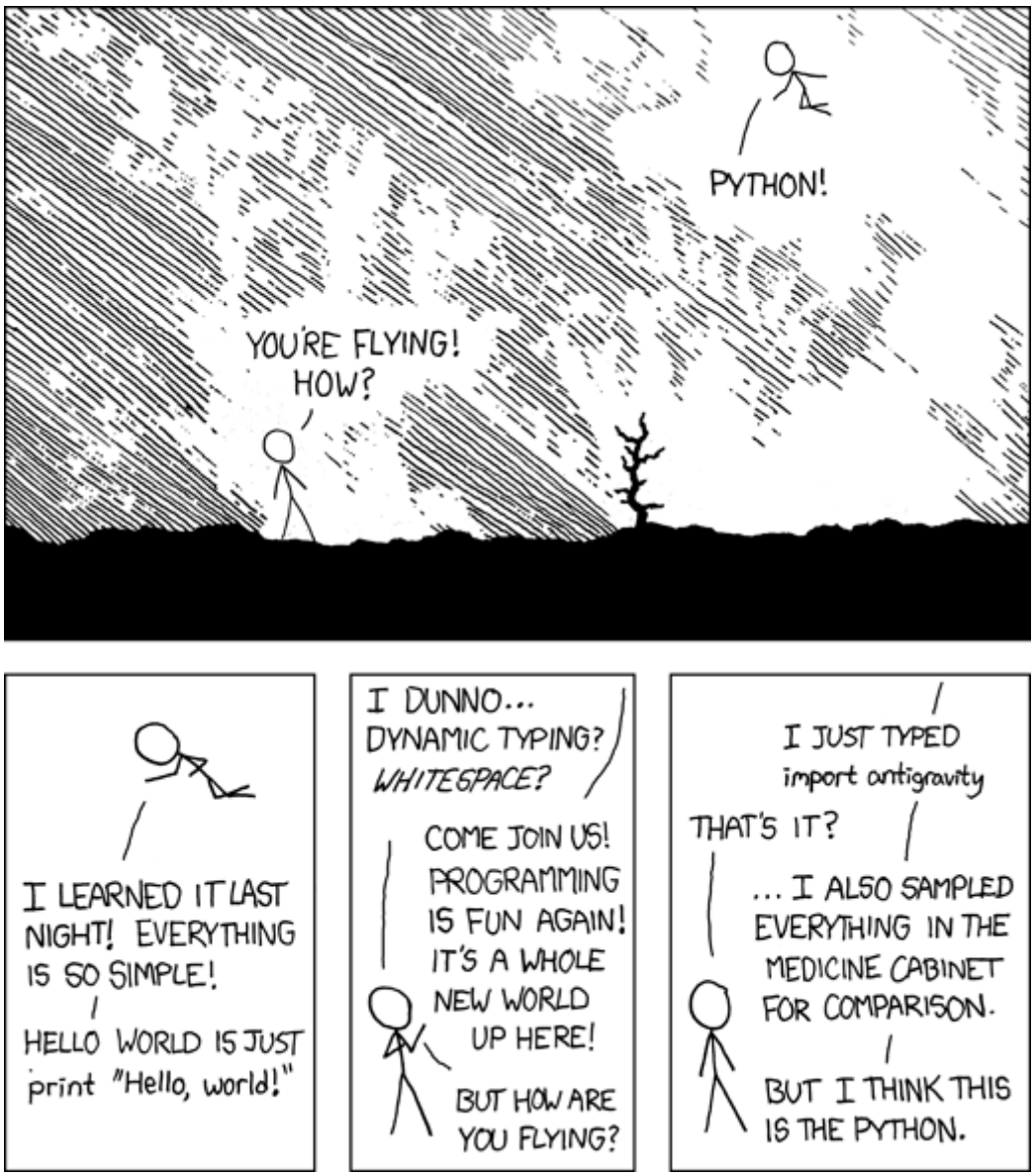
Dutch programmer Guido van Rossum, who is currently working at Dropbox (previously at Google), invented Python. It was not a popular language until somewhat recently. The popularity exponentially shot up due to its adoption for machine learning projects and the availability of many libraries.

Python is an essential skill every data scientist should possess in order to excel in data exploration, extraction, analysis, and visualization.

Python is an interpreted language. (FYI, C/C++ are compiled languages.) Once you write your code, the compiler will start executing it line by line from the top of the file.

WHY USE PYTHON FOR MACHINE LEARNING?

- Python is Easy To Use: Python is simple with an easily readable syntax and that makes it well-loved by both seasoned developers and experimental students.
- Python has multiple Libraries and Frameworks: Python is already quite popular and consequently, it has hundreds of different libraries and frameworks that can be used by developers.
- Python has Community and Corporate Support: Python has been around since 1990 and that is ample time to create a supportive community.
- Python is Portable and Extensible: A lot of cross-language operations can be performed easily on Python because of its portable and extensible nature



Python has become the first choice of programmers in machine learning. The services of Python are suitable for ML developers. If you are developing software in ML, then use Python. The credibility of this language is higher than in others. Also, it is easy to use and understand. So it has become a popular choice for ML.

The developers have stick to this language for their programming tasks. If you observe, then you will find that most of the ML algorithm. The Python code is suitable to use and implement on any platform. Also, flexibility plays a vital role in Python. I have shared my views about the use and the benefits of Python in ML. And I hope that this post has helped you in getting a clear image of Python's role in ML.

Moreover, the benefits of Python are not limited here. It has many advantages which you will notice when using it. Also, it is best for beginners in ML programming.

Programming Styles

There are three main ways to write Python codes.

- Unstructured
- Procedural
- Object-oriented

In unstructured programming, you write the code as one big monolithic file. It is discouraged to use this style of writing for large programs as it is quite difficult to manage. However, for small code snippets, like what we are going to do in this tutorial, it is a convenient way of writing programs.

In procedural programming, we group code into functional units called functions. There are two steps involved here:

- Define (write) the function
- Invoke (call) the function You write a function once and invoke as many time as you want to execute it. In this post, we will mainly be using unstructured and procedural coding styles.

In object-oriented programming, you identify blueprints and create what is called a class for each blueprint. We will be exploring object-oriented programming in Python in a later post.

Read more in Sources

https://levelup.gitconnected.com/python-for-absolute-beginners-a-quick-primer-c7db94a5d0e (https://levelup.gitconnected.com/python-for-absolute-beginners-a-quick-primer-c7db94a5d0e)

https://hackernoon.com/why-python-used-for-machine-learning-u13f922ug (https://hackernoon.com/why-python-used-for-machine-learning-u13f922ug)

https://levelup.gitconnected.com/python-for-absolute-beginners-a-quick-primer-c7db94a5d0e (https://levelup.gitconnected.com/python-for-absolute-beginners-a-quick-primer-c7db94a5d0e))

https://www.geeksforgeeks.org/best-python-libraries-for-machine-learning/ (https://www.geeksforgeeks.org/best-python-libraries-for-machine-learning/)

https://www.geeksforgeeks.org/introduction-machine-learning-using-python/ (https://www.geeksforgeeks.org/introduction-machine-learning-using-python/)

https://www.geeksforgeeks.org/python-generate-test-datasets-for-machine-learning/ (https://www.geeksforgeeks.org/python-generate-test-datasets-for-machine-learning/)

Basic Arithmetic In Python

In [2]:

```
#Addition
1+1
```

Out[2]:

2

In [3]:

```
# Subtraction
2-1
```

Out[3]:

```
1
```

In [4]:

```
# Multiplication
2*3
```

Out[4]:

```
6
```

Division

3/3

In [5]:

```
# Division always returns floats!
1/1
```

Out[5]:

```
1.0
```

In [6]:

```
# Powers
2 ** 3
```

Out[6]:

```
8
```

In [7]:

```
2**(3 / 2)
```

Out[7]:

```
2.8284271247461903
```

In [8]:

```
# Order of Operations
1 - (10 + 1)
```

Out[8]:

```
-10
```

In [9]:

```
(1 + 2) * (1000+1)
```

Out[9]:

```
3003
```

VARIABLES AND DATATYPES IN PYTHON

Python Scalar Types

We begin by discussing the basic scalar types in Python. These are **INTEGERS, FLOATS AND BOOLEAN AND STRINGS**

PYTHON NUMBERS

Integers are whole numbers, while Float (floating point) numbers are numbers with a decimal point.

In [10]:

```
type("bat")
```

Out[10]:

```
str
```

In [11]:

```
bat = 5
```

In [12]:

```
100
```

Out[12]:

```
100
```

In [13]:

```
type(100)
```

Out[13]:

```
int
```

In [14]:

```
1.2
```

Out[14]:

```
1.2
```

In [15]:

```
type(1.0)
```

Out[15]:

```
float
```

In [16]:

```
type(100.0)
```

Out[16]:

```
float
```

In [17]:

```
type(100.)
```

Out[17]:

```
float
```

Assigning Variables

In [18]:

```
a = 40
```

In [19]:

```
a
```

Out[19]:

```
40
```

In [20]:

```
type(a)
```

Out[20]:

```
int
```

In [21]:

```
a + 3
```

Out[21]:

```
43
```

In [22]:

```
b = 5
```

In [23]:

```
a / b
```

Out[23]:

```
8.0
```

In [24]:

```
# Reassignment
a = 1000
```

In [25]:

```
a + b
```

Out[25]:

```
1005
```

Reassignment with same variable

In [26]:

```
a
```

Out[26]:

```
1000
```

In [27]:

```
# Keep in mind, if you run this more than once, you will keep running a = a+a!
a = a + a
```

In [28]:

```
a
```

Out[28]:

```
2000
```

In [29]:

```
a = a + a
```

In [30]:

```
a
```

Out[30]:

```
4000
```

Python Strings

A string is an ordered sequence of characters. Two key words here, **ordered** and **characters.** Ordered means that we will be able to use *indexing* and *slicing* to grab elements from the string.

Creating strings.

In [31]:

```
# Comment. Won't show up when you run a script.
```

In [32]:

```
# Single or double quotes are okay.
"hello this is my first string"
```

Out[32]:

```
'hello this is my first string'
```

In [33]:

```
"hello"
```

Out[33]:

```
'hello'
```

In [34]:

```
# Keep in mind potential errors
'I'm a beginner in python programming!
```

```
  File "<ipython-input-34-eca165bf5429>", line 2
    'I'm a beginner in python programming!
       ^
SyntaxError: invalid syntax
```

In [35]:

```
# Use another set of quotes to capture that inside single quote
" I'm a beginner in python programming! "
```

Out[35]:

```
" I'm a beginner in python programming! "
```

Basic Printing of Strings

In the jupyter notebook, a single string in a cell is automatically returned back. However, this is different than printing a string. Printing a string allows us to have multiple outputs. Let's see some useful examples:

In [36]:

```
'test'
```

Out[36]:

```
'test'
```

In [37]:

```
'one'
'two'
```

Out[37]:

```
'two'
```

In [38]:

```
print('one')
print('two')
```

```
one
two
```

In [39]:

```
print('this is a new line \n notice how this is on a new line')
```

```
this is a new line
 notice how this is on a new line
```

In [40]:

```
print("Python Code")
print('this is a tab \t notice how this prints with space between')
```

```
Python Code
this is a tab    notice how this prints with space between
```

Indexing and Slicing

Since strings are *ordered sequences* of characters, it means we can "select" single characters (indexing) or grab sub-sections of the string (slicing).

Indexing

Indexing starts a 0, so the string hello:

```
character:    h    e    l    l    o
index:        0    1    2    3    4
```

You can use square brackets to grab single characters

In [41]:

```
word = "hello"
print(word)
```

```
hello
```

In [42]:

```
word[0]
```

Out[42]:

```
'h'
```

In [43]:

```
word[3]
```

Out[43]:

```
'l'
```

Python also supports reverse indexing:

```
character:        h     e     l     l     o
index:            0     1     2     3     4
reverse index:    0    -4    -3    -2    -1
```

Reverse indexing is used commonly to grab the last "chunk" of a sequence.

In [44]:

```
word[-2]
```

Out[44]:

```
'l'
```

Slicing

We can grab entire subsections of a string with *slice* notation.

This is the notation:

```
[start:stop:step]
```

Key things to note:

1. The starting index direclty corresponds to where your slice will start
2. The stop index corresponds to where you slice will go up to. **It does not include this index character!**
3. The step size is how many characters you skip as you go grab the next one.

Let's see some examples

In [45]:

```
alpha = 'abcdef'
```

In [46]:

```
# NOTICE HOW d IS NOT INCLUDED!
alpha[0:3]
```

Out[46]:

```
'abc'
```

In [47]:

```
alpha[0:4]
```

Out[47]:

```
'abcd'
```

In [48]:

```
alpha[2:4]
```

Out[48]:

```
'cd'
```

In [49]:

```
alpha[2:]
```

Out[49]:

```
'cdef'
```

In [50]:

```
alpha[:2]
```

Out[50]:

```
'ab'
```

In [51]:

```
alpha[0:6:2]
```

Out[51]:

```
'ace'
```

Basic String Methods

Methods are actions you can call off an object usually in the form .method_name() notice the closed parenthesis at the end. Strings have many, many methods which you can check with the Tab functionality in jupyter notebooks, let's go over some of the more useful ones!

In [52]:

```
basic = "hello world I am still a beginner pythonista"
```

In [53]:

```
basic.upper()
```

Out[53]:

```
'HELLO WORLD I AM STILL A BEGINNER PYTHONISTA'
```

In [54]:

```
basic.lower()
```

Out[54]:

```
'hello world i am still a beginner pythonista'
```

In [55]:

```
# Preview, we'll learn about lists later on!
basic.split()
```

Out[55]:

```
['hello', 'world', 'I', 'am', 'still', 'a', 'beginner', 'pythonista']
```

In [56]:

```
basic.split('i')
```

Out[56]:

```
['hello world I am st', 'll a beg', 'nner python', 'sta']
```

Print Formatting

You can use the .format() method off a string, to perform what is formally known as **string interpolation**, essentially inserting variables when printing a string.

In [57]:

```
user_name = "Newbie"
password = 12345
```

In [58]:

```
print("Welcome {} and your password is {}".format(user_name, password))
```

```
Welcome Newbie and your password is 12345
```

In [59]:

```
action = 'learn'
```

In [60]:

```
print("The {} needs to {}".format(user_name,action))
```

```
The Newbie needs to learn
```

In [61]:

```
print("The {a} needs to {b}".format(a=user_name,b=action))
```

```
The Newbie needs to learn
```

In [62]:

```
print("The {b} needs to {a}".format(a=user_name,b=action))
```

```
The learn needs to Newbie
```

Formatting Numbers

In [63]:

```
num = 245.9083
print("The number is: {}".format(num))
```

```
The number is: 245.9083
```

In [64]:

```
print("The code is: {:.1f}".format(num))
```

```
The code is: 245.9
```

In [65]:

```
print("The code is: {:.2f}".format(num))
```

```
The code is: 245.91
```

In [66]:

```
print("The code is: {:.3f}".format(num))
```

```
The code is: 245.908
```

In [67]:

```
print("The code is: {:.4f}".format(num))
```

```
The code is: 245.9083
```

Slice out " ext and we are going "

In [68]:

```
test = "This is a really long piece of text and we are going use it for practice"
```

In [ ]:

```

```

Booleans

Booleans are data types that indicate a logical state of **True** or **False**. Python also has a placeholder object called **None**. Let's explore what these look like. We will work with them a lot more once we begin to learn about control flow with Python, but until then, let's just get to understand what they look like (notice the syntax highlighting).

In [69]:

```
# Booleans
a = True
```

In [70]:

```
a
```

Out[70]:

```
True
```

In [71]:

```
type(a)
```

Out[71]:

```
bool
```

In [72]:

```
b = False
```

In [73]:

```
type(b)
```

Out[73]:

```
bool
```

Later on we will learn how comparison operators return booleans.

In [74]:

```
1 > 2
```

Out[74]:

```
False
```

We can use None as a placeholder for an object that we don't want to reassign yet:

In [75]:

```
c = None
```

In [76]:

```
type(c)
```

Out[76]:

```
NoneType
```

In [ ]:

```

```

Lists and Loops

Lists

We've learned that strings are sequences of characters. Similarly, lists are sequences of objects, they can hold a variety of data types in order, and they follow the same sequence and indexing bracket rules that strings do. They can also take in mixed data types.

Let's explore some useful examples:

In [77]:

```
alist = []
type(alist)
```

Out[77]:

```
list
```

In [78]:

```
my_list = [1,2,3]
```

In [79]:

```
my_list
```

Out[79]:

```
[1, 2, 3]
```

In [80]:

```
my_list2 = ['a','b','c', 1, 2]
```

In [81]:

```
a = 100
b = 200
c = 300
my_list3 = [a,b,c]
```

In [82]:

```
my_list3
```

Out[82]:

```
[100, 200, 300]
```

Indexing and Slicing

This works the same as in a string!

In [83]:

```
mylist = ['a','b','c','d']
```

In [84]:

```
mylist[3]
```

Out[84]:

```
'd'
```

In [85]:

```
mylist[0:3]
```

Out[85]:

```
['a', 'b', 'c']
```

The len function

Python has built in functions that you can call. We'll slowly introduce more of them as we need them. One useful built in function is the **len** function which returns back the length of an object.

In [86]:

```
len('string')
```

Out[86]:

```
6
```

In [87]:

```
len(my_list)
```

Out[87]:

```
3
```

Useful List Methods

Methods are actions you can call off a function. Their typical format is:

```
mylist = [1,2,3]
mylist.some_method()
```

You must call the parenthesis to execute the method! Let's go through a few useful ones pertaining to lists.

In [88]:

```
mylist = [1,2,3]
```

In [89]:

```
mylist.append(6)
```

In [90]:

```
mylist
```

Out[90]:

```
[1, 2, 3, 6]
```

In [91]:

```
mylist.pop()
```

Out[91]:

```
6
```

In [92]:

```
mylist
```

Out[92]:

```
[1, 2, 3]
```

In [93]:

```
mylist.append
```

Out[93]:

```
<function list.append(object, /)>
```

In [94]:

```
mylist.append(4)
```

In [95]:

```
mylist.append(10)
mylist.append(20)
```

In [96]:

```
mylist
```

Out[96]:

```
[1, 2, 3, 4, 10, 20]
```

In [97]:

```
lastitem = mylist.pop()
```

In [98]:

```
lastitem
```

Out[98]:

```
20
```

In [99]:

```
mylist
```

Out[99]:

```
[1, 2, 3, 4, 10]
```

In [100]:

```
first_item = mylist.pop(0)
```

In [101]:

```
first_item
```

Out[101]:

```
1
```

In [102]:

```
mylist
```

Out[102]:

```
[2, 3, 4, 10]
```

In [103]:

```
mylist = [1,2,3]
```

In [104]:

```
# This method doesn't return anything.
# Instead it performs the action "in-place" , or on the list itself without returning ar
mylist.reverse()
```

In [105]:

```
mylist
```

Out[105]:

```
[3, 2, 1]
```

In [106]:

```
# Also in place
mylist.sort(reverse=True)
```

In [107]:

```
mylist
```

Out[107]:

```
[3, 2, 1]
```

In [108]:

```
# THIS WON'T WORK!
result = mylist.reverse()
```

In [109]:

```
# Doesn't return anything
result
```

In [110]:

```
print(result)
```

```
None
```

In [111]:

```
mylist
```

Out[111]:

```
[1, 2, 3]
```

In [112]:

```
mylist.insert(3,'middle')
```

In [113]:

```
mylist
```

Out[113]:

```
[1, 2, 3, 'middle']
```

Nested Lists

Lists can hold other lists! This is called a nested list. Let's see some examples.

In [114]:

```
new_list = [1,2,3,['a','b','c']]
```

In [115]:

```
new_list[3]
```

Out[115]:

```
['a', 'b', 'c']
```

In [116]:

```
type(new_list)
```

Out[116]:

```
list
```

In [117]:

```
new_list[3]
```

Out[117]:

```
['a', 'b', 'c']
```

In [118]:

```
new_list[3][0]
```

Out[118]:

```
'a'
```

In [119]:

```
list_1 = [2,3,"four", [20,30,40, ["one", "two", "three"]]]
```

In [120]:

```
list_1[3][3][1:]
```

Out[120]:

```
['two', 'three']
```

Loops

WHILE LOOPS

A while loop will repeatedly execute a single statement or group of statements as long as the condition being checked is true. The reason it is called a 'loop' is because the code statements are looped through over and over again until the condition is no longer met.

Code indentation becomes very important as we begin to work with loops and control flow.

In [121]:

```
a = 1

while a < 10:
    print("a is less than 10")
    a = a + 1
```

```
a is less than 10
a is less than 10
a is less than 10
a is less than 10
a is less than 10
a is less than 10
a is less than 10
a is less than 10
a is less than 10
```

In [122]:

```
# Start by setting variable x to 0
x = 0

while x < 3:
    print('X is currently')
    print(x)
    print("Adding 1 to x")
    x = x + 1 #alternatively you could write x += 1
```

```
X is currently
0
Adding 1 to x
X is currently
1
Adding 1 to x
X is currently
2
Adding 1 to x
```

Note This!

Be careful with while loops! There is a potential to write a condition that always remains True, meaning you have an infinite running while loop. If this happens to you, you can stop/restart the kernel.

break keyword

The break keyword allows you to "break" out of the loop that contains the break keyword. For example

In [123]:

```
x = 0

while x < 10:
    print(x)
    print('adding one to x')
    x = x + 1

    if x == 3:
        # This will cause to break out of the top loop
        # Note that if statements don't count as loops
        break
```

```
0
adding one to x
1
adding one to x
2
adding one to x
```

Excellent work recruit! Let's move on to discuss for loops!

for loop

A **for loop** acts as an iterator in Python, it goes through items that are in a sequence or any other iterable item. Objects that we've learned about that we can iterate over include strings,lists,tuples, and even built in iterables for dictionaries, such as the keys or values.

Here's the general format for a for loop in Python:

```
for item in object:
    statements to do stuff
```

The variable name used for the item is completely up to the coder, so use your best judgment for choosing a name that makes sense and you will be able to understand when revisiting your code.

for loop with a list

In [124]:

```
mylist = [1,2,3,4]

for num in mylist:
    print(num**2)
```

```
1
4
9
16
```

In [125]:

```
for totally_made_up in mylist:
    print(totally_made_up)
```

```
1
2
3
4
```

In [126]:

```
for num in mylist:
    print(num,end=' ')
```

```
1 2 3 4
```

In [127]:

```
for num in mylist:
    print("I am in a for loop")
```

```
I am in a for loop
I am in a for loop
I am in a for loop
I am in a for loop
```

for loop with strings

In [128]:

```
for character in "This is a string":
    print(character )
```

```
T
h
i
s

i
s

a

s
t
r
i
n
g
```

In [129]:

```
mystring = 'This is a string'

for word in mystring.split():
    print(word)
```

```
This
is
a
string
```

for loop with tuple

In [130]:

```
tup = (1,2,3,4)

for num in tup:
    print(num)
```

```
1
2
3
4
```

tuple unpacking

In [131]:

```
list_of_tups = [(1,2),(3,4),(5,6),(7,8),(9,10)]
```

In [132]:

```
for x in list_of_tups:
    print(x)
```

```
(1, 2)
(3, 4)
(5, 6)
(7, 8)
(9, 10)
```

In [133]:

```
for x in list_of_tups:
    print(x[0])
```

```
1
3
5
7
9
```

In [134]:

```
for x in list_of_tups:
    print(x[1])
```

```
2
4
6
8
10
```

In [135]:

```
# Doesn't need the parenthesis

for num1,num2 in list_of_tups:
    print(num1)
    print(num2)
    print('\n')
```

```
1
2


3
4


5
6


7
8


9
10
```

for loop with dictionaries

In [136]:

```
my_dictionary = {'a':1,'b':2,'c':3}
```

Remember that dictionaries don't retain any order! So only loop through them with this in mind!

In [137]:

```
for item in my_dictionary:
    print(item)
```

```
a
b
c
```

In [138]:

```
for k in my_dictionary.values():
    print(k)
```

```
1
2
3
```

In [139]:

```
for k in my_dictionary.keys():
    print(k)
    print(my_dictionary[k])
    print('\n')
```

```
a
1


b
2


c
3
```

---

continue

The continue keyword can be a bit tricky to see its usefulness, but it allows you to continue with the top level loop, basicaly the opposite of break. It will take time before you realize a good situation to use it in, but here is a simple example:

In [140]:

```
scores = [30,40,50,60]

for num in scores:
    if num == 50:
        continue
    print("Your score is", num)
```

```
Your score is 30
Your score is 40
Your score is 60
```

In [141]:

```
for letter in 'code':
    if letter == 'e':
        continue

    print('Current Letter is:',letter)
```

```
Current Letter is: c
Current Letter is: o
Current Letter is: d
```

Conditional Statements

if , elif , and else statements

Majority of the time when programming, we'll need to control the flow of our logic. Our program will want to perform an action in only certain cases, we can use the **if**, **elif**, and **else** statements to control for these cases. Let's work through some examples:

Simple if Statement

The format for an if statement

```
if some_condition:
    # Do Something
```

In [142]:

```
if 1<2:
    print('One is less than two')
```

```
One is less than two
```

In [143]:

```
if 1>2:
    print("One is greater than two")
```

Notice what happens, the indented block of code only runs when the if condition is True!

if else Statement

Let's now add in an alternate action in case the **if** is not True using the **else** statement.

Notice the format and how the code blocks line up, this is crucial in Python! Code indentation let's Python know what blocks and statements correspond together.

In [144]:

```
if 1==1:
    print("One is equal to One")
else:
    print("First if was not True")
```

```
One is equal to One
```

In [145]:

```
if 1==2:
    print("One is equal to Two")
else:
    print("First if was not True")
```

```
First if was not True
```

In [150]:

```
saved_password = 123456
new_password = input("Enter your password")

if int(new_password) == saved_password:
    print("Password correct")
else:
    print("You're a criminal")
```

```
Enter your password46373
You're a criminal
```

if, elif, else

Now let's imagine we have multiple conditions to check before the final **else** statement, this is where we can use the **elif** keyword to check for as many individual conditions as necessary:

In [151]:

```
if 2 == 0:
    print('First condition True')
elif 2 == 1:
    print("Second condition True")
elif 2 == 100:
    print("Third condition True")
else:
    print("None of the above conditions were True")
```

```
None of the above conditions were True
```

Let's see what happens if we had multiple True conditions:

In [152]:

```
if 2 == 2:
    print('First condition True')
elif 2 == 2:
    print("Second condition True")
elif 2 == 2:
    print("Third condition True")
else:
    print("None of the above conditions were True")
```

```
First condition True
```

Notice how only the first True condition's code block will be executed, keep this in mind when writing your scripts!

Dictionaries, Tuples and Sets

Dictionaries

So far we've only seen how to store data types in sequences like storing characters in a string or items in a list. But what if we want to store information another way? Python support Dictionaries which is a key-item data structure.

The choice of deciding between sequences like a list and mappings like a dictionary often depends on the specific situation. As you become a stronger programmer, choosing the right storage format will become more intuitive.

Let's cover the basics of dictionaries!

Creating a Dictionary

In [153]:

```
new_dict = {"item1": 1,
            "item2" : 2,
            "item3" : "Abuja"}
```

In [154]:

```
new_dict["item3"] = "Kano"
```

In [155]:

```
new_dict
```

Out[155]:

```
{'item1': 1, 'item2': 2, 'item3': 'Kano'}
```

In [156]:

```
# Make a dictionary with {} and : to signify a key and a value
d = {'key1':'value1','key2':'value2'}
```

In [157]:

```
# Call values by their key
d['key1']
```

Out[157]:

```
'value1'
```

In [158]:

```
d['key2']
```

Out[158]:

```
'value2'
```

Adding New Key-Item Pairs

In [159]:

```
d['new_key'] = 'new item'
```

In [160]:

```
d
```

Out[160]:

```
{'key1': 'value1', 'key2': 'value2', 'new_key': 'new item'}
```

Note: Dictionaries are unordered! ***This may not be clear at first with smaller dictionaries, but as dictionaries get larger they won't retain order, which means they can not be sorted!*** **If you need order and the ability to sort, stick with a sequence, like a list!**

In [161]:

```
d = {'a':1,'z':2}
```

In [162]:

```
d
```

Out[162]:

```
{'a': 1, 'z': 2}
```

In [163]:

```
d['new'] = 0
```

In [164]:

```
d
```

Out[164]:

```
{'a': 1, 'z': 2, 'new': 0}
```

In [165]:

```
d['za'] = 'hello'
```

In [166]:

```
d
```

Out[166]:

```
{'a': 1, 'z': 2, 'new': 0, 'za': 'hello'}
```

Dictionaries are very flexible in the data types they can hold, they can hold numbers, strings, lists, and even other dictionaries!

In [167]:

```
d = {'k1':10,
     'k2':'stringy',
     'k3':[1,2,3,],
     'k4':{'inside_key':[20,30,40, {"keyinside": [60,20, "thirty"]}]}}
```

In [168]:

```
d["k4"]["inside_key"][3]["keyinside"][1:]
```

Out[168]:

```
[20, 'thirty']
```

In [169]:

```
d['k1']
```

Out[169]:

```
10
```

In [170]:

```
d['k2']
```

Out[170]:

```
'stringy'
```

In [171]:

```
d['k3']
```

Out[171]:

```
[1, 2, 3]
```

In [172]:

```
d['k3'][0]
```

Out[172]:

```
1
```

In [173]:

```
d['k4']
```

Out[173]:

```
{'inside_key': [20, 30, 40, {'keyinside': [60, 20, 'thirty']}]}
```

In [174]:

```
d['k4']['inside_key']
```

Out[174]:

```
[20, 30, 40, {'keyinside': [60, 20, 'thirty']}]
```

Error if you ask for a key that isn't there!

In [175]:

```
d['oops']
```

```
---------------------------------------------------------------------------
-
KeyError                                  Traceback (most recent call las
t)
<ipython-input-175-cf5dba4e3111> in <module>
----> 1 d['oops']

KeyError: 'oops'
```

Keep dictionaries in mind when you need to create a mapping and don't care about order!

For example:

In [176]:

```
short_names = {"AAU": "Ambrose Alli University",
               "OAU": "Obafemi Awolowo University",
               "UNILAG": "University of Lagos",
               "NDA": "National Defence Academy"}
```

In [177]:

```
short_names["AAU"]
```

Out[177]:

```
'Ambrose Alli University'
```

Another example of using Dictionaries...

In [178]:

```
pop_in_mil = {"Nigeria": 180,
              "USA":323,
              "Germany": 83,
              "India": 1324}
```

In [179]:

```
pop_in_mil["Nigeria"]
```

Out[179]:

```
180
```

Methods

In [180]:

```
short_names.values()
```

Out[180]:

```
dict_values(['Ambrose Alli University', 'Obafemi Awolowo University', 'Uni
versity of Lagos', 'National Defence Academy'])
```

In [181]:

```
short_names.keys()
```

Out[181]:

```
dict_keys(['AAU', 'OAU', 'UNILAG', 'NDA'])
```

In [182]:

```
short_names.items()
```

Out[182]:

```
dict_items([('AAU', 'Ambrose Alli University'), ('OAU', 'Obafemi Awolowo U
niversity'), ('UNILAG', 'University of Lagos'), ('NDA', 'National Defence
Academy')])
```

Tuples

Tuples are ordered sequences just like a list, but have one major difference, they are **immutable**. Meaning you can not *change* them. So in practice what does this actually mean? It means that you can not reassign an item once its in the tuple, unlike a list, where you can do a reassignment.

Let's see this in action:

Creating a Tuple

You use parenthesis and commas for tuples:

In [183]:

```
t = (1,2,3)
```

In [184]:

```
type(t)
```

Out[184]:

```
tuple
```

In [185]:

```
# Mixed data types are fine
t = ('a',1)
```

In [186]:

```
# Indexing works just like a list
t[0]
```

Out[186]:

```
'a'
```

Immutability

In [187]:

```
mylist = [1,2,3]
```

In [188]:

```
type(mylist)
```

Out[188]:

```
list
```

In [189]:

```
# No problem for a list!
mylist[0] = 'new'
```

In [190]:

```
mylist
```

Out[190]:

```
['new', 2, 3]
```

In [191]:

```
t = (1,2,3)
```

In [192]:

```
t[0] = 'new'
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-192-f469ab99125e> in <module>
----> 1 t[0] = 'new'

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

You also can't add items to a tuple:

In [193]:

```
t.append('NOPE!')
```

```
---------------------------------------------------------------------------
-
AttributeError                            Traceback (most recent call las
t)
<ipython-input-193-8b6ac6fc5b45> in <module>
----> 1 t.append('NOPE!')

AttributeError: 'tuple' object has no attribute 'append'
```

Tuple Methods

Tuples only have two methods available .index() and count()

In [194]:

```
t = ('a','b','c','a')
```

In [195]:

```
# Returns index of first instance!
t.index('a')
```

Out[195]:

```
0
```

In [196]:

```
t.count('b')
```

Out[196]:

```
1
```

Why use tuples?

Lists and tuples are very similar, so you may find yourself exchanging use cases for either one. However, you should use a tuple for collections or sequences that shouldn't be changed, such as the dates of the year, or user information such as an address,street, city , etc.

Sets

Another fundamental Data Structure is The Set!

Sets are an unordered collection of unique elements. We can construct them by using the set() function. Let's go ahead and make a set to see how it works:

Constructing Sets

In [197]:

```
x = set()
```

In [198]:

```
x.add(1)
```

In [199]:

```
x
```

Out[199]:

```
{1}
```

In [200]:

```
x.add(2)
```

In [201]:

```
x
```

Out[201]:

```
{1, 2}
```

Note the curly brackets. This does not indicate a dictionary!

A set has only unique entries. So what happens when we try to add something that is already in a set?

In [202]:

```
x.add(1)
```

In [203]:

```
x
```

Out[203]:

```
{1, 2}
```

Notice how it won't place another 1 there. That's because a set is only concerned with unique elements! We can cast a list with multiple repeat elements to a set to get the unique elements. For example:

In [204]:

```
mylist = [1,1,1,1,1,2,2,2,2,2,3,3,3,3,3]
```

In [205]:

```
set(mylist)
```

Out[205]:

```
{1, 2, 3}
```

You can also quickly create a set with just {}

In [206]:

```
myset = {1,2,3,3,3,3,3,3}
```

In [207]:

```
myset
```

Out[207]:

```
{1, 2, 3}
```

In [208]:

```
type(myset)
```

Out[208]:

```
set
```

Functions In Python

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

Defining a Function

You can define functions by following the rules below:

- Function blocks begin with the keyword **def** followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
- The first statement of a function can be an optional statement - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

```
def functionname( parameters ):
   "function_docstring descrinng what the function does"

   # list of expressions to be executed

   return [expression]
```

Example

The following function takes a two numbers as input parameters and prints out the sum:

In [209]:

```
def print_sum(a, b):
    "This prints the sum of two numbers a and b"
    print(a + b)
    return
```

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code. Inorder to use the function, you must call the name and pass in the required parameters.

To demonstrate this, we call the **print_sum** function we created above:

In [210]:

```
print_sum(5,6)
```

```
11
```

Passing values

Pass by reference

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example:

In [211]:

```
def change_name(name):
    print("The passed name is {}".format(name))
    #change the name
    name = "Harry Potter"
    print("The new name in this function is {}".format(name))
    return

#call the function
name = "John Stone"
change_name(name)
```

```
The passed name is John Stone
The new name in this function is Harry Potter
```

Function Arguments

You can call a function by using the following types of formal arguments:

- Required arguments
- Keyword arguments
- Default arguments
- Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition. For example, the function change_name above requires one argument for it to work. If you call the function without the required argument it throws an error.

In [212]:

```
change_name()
```

```
---------------------------------------------------------------------------
-
TypeError                                 Traceback (most recent call las
t)
<ipython-input-212-25bf586f7e90> in <module>
----> 1 change_name()

TypeError: change_name() missing 1 required positional argument: 'name'
```

In [213]:

```
change_name("Opeyemi")
```

```
The passed name is Opeyemi
The new name in this function is Harry Potter
```

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

In [214]:

```
change_name(name="Mercy")
```

```
The passed name is Mercy
The new name in this function is Harry Potter
```

In [215]:

```
def printinfo(name,age):
    "This prints the info about a person"
    print("Name: ", name)
    print("Age ", age)
    return

printinfo(name="Jesse", age=20)
```

```
Name:  Jesse
Age  20
```

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed.

In [216]:

```
def printinfo(age=25):
    "This prints the info about a person"
    print("Age ", age)
    return


printinfo()
```

```
Age  25
```

In [217]:

```
printinfo(45)
```

```
Age  45
```

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called variable-length arguments and are not named in the function definition, unlike required and default arguments.

```
    def functionname([formal_args,] *var_args_tuple ):
       "function_docstring"
       function_expressions
       return [expression]
```

An asterisk (*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example::

In [218]:

```
def printargs(name, *scores):
   "This prints a variable passed arguments"
   print("Name is: {}".format(name))
   #print info in variable leanght arguments
   for var in scores:
        print(var)
```

In [219]:

```
printargs("John", 80,90,98,97)
```

```
Name is: John
80
90
98
97
```

Return Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

In [220]:

```
def sum(a, b ):
   # Add both the parameters and return total"
   total = a + b
   return total

# Now you can call sum function and pass the result to a variable
total = sum(10,20)
print(total)
```

```
30
```

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope.

In [221]:

```
sum = 0 # This is global variable.

def add_num(a, b):
   # Add both the parameters and return total"
   sum = a + b
   print("The total inside the function is: {}".format(sum))
   return sum

# Now you can call sum function
add_num(10,20);
print("The total outside the function is: {}".format(sum))
```

```
The total inside the function is: 30
The total outside the function is: 0
```