Python and Jupyter Fundamentals

In this introductory section we will look at the basics of the python programming language.

This will also give you an opportunity to get a feel for using the Jupyter environment to write and run code.

Getting Started with Jupyter

The "Jupyter Notebook" is a web application that allows you to create and share documents that contain code, narrative text, visualisations and equations.

In Jupyter we use the name "notebook" for a file that we type code and text (markdown) into.

The file that you are reading now is a "notebook".

We use the name "cell" for each box that we type code or text into.

A "notebook" can have any number of "cells"

See https://jupyter.org/ for more info.

The most basic operations in Jupyter are:

- Select a cell by clicking on it.
- Once the cursor is flashing in a code cell you can write code.
- Run a cell by pressing Ctrl-Enter (stays on the current cell) OR Shift-Enter (highlights the next cell).
- If you double click a text cell it will change to edit mode on that cell. Ctrl-Enter will exit edit mode.

We'll explore the Jupyter interface further as we progress through the course.

Jupyter Cells

Jupyter can have different types of cells, some cells have python code, some have "markdown"

This is a markdown cell

You can format the text in markdown cells with markdown syntax, or html.

e.g.

bold text

1. this is
2. a
3. numbered list

1. Python Fundamentals

1-1. Python Operators

We'll run each of the code cells below to see some basic mathematical operations.

1 plus 9 plus 29

In [1]:
```
1 + 9 + 29
```
Out[1]:
```
39
```

2 times 9 times 10

In [2]:
```
2 * 9 * 10
```
Out[2]:
```
180
```

2 times 3 + 12

In [3]:
```
2 * 3 + 12
```
Out[3]:
```
18
```

Python will follow "PEMDAS", but it's good practice to use parentheses for clarity

In [4]:
```
(2 * 3) + 12
```
Out[4]:
```
18
```

spaces help also - see how the brackets change the result

In [5]:
```
2 * (3 + 12)
```
Out[5]:
```
30
```

(3+4)squared – (4 - 2)squared

In [6]:
```
(3 + 4)**2 - (4 - 2)**2
```
Out[6]:
```
45
```

remainder of 7 divided by 4

In [7]:
```
7 % 4
```
Out[7]:
```
3
```

Common Operators

a+b , addition

a-b , subtraction

a*b , multiplication

a/b , division

a//b floor division (e.g. 5//2=2)

a%b , modulo

-a , negation

abs(a) , absolute value

a**b , exponent

math.sqrt(a) , square root

In [8]:
```
# math for square root
import math

math.sqrt(49)
```

Out[8]:
```
7.0
```

Exercise 1

- Write python code to do each of the following simple calculations.
- You can put them all in a single cell.

> - (8 + 15) - 10
> - 7^2 - 7 + 12
> - 12.2 * 5.7 * $\frac{1}{4}$
> - remainder of 12 divided by 5
> - $\sqrt{4^2 + 3^2}$

Notice that by default jupyter will print the result of the last line of code only.

- Everything after a **#** symbol in a line won't be run.
- You can use this to "comment out" lines

In [9]:
```
# Do the exercise here
print((8 + 15) - 10 )
print(7**2 - 7 + 12)
print(12.2 * 5.7 * (1/4))
print( 12 % 5 )
print(math.sqrt(4**2 + 3**2))
```

```
13
54
17.384999999999998
2
5.0
```

1-2. Variables

A "variable" is a name used to store values

You can save the results of a computation into a variable for later use.

In [10]:
```
x = 25 + 13
```

Here the contents of x are displayed because it is the last line of code in the cell

In [11]:
```
x
```

Out[11]:
```
38
```

We can "print" the variable x

Here the contents of x will be displayed even if it is not the last line of code in the cell

The contents of the last line are still displayed

In [12]:
```
# change x
x = x + 10

print(x)

# create more variables e.g. y
y = 'what is this?'

y
```

```
48
```

Out[12]:
```
'what is this?'
```

1-3. Jupyter Keyboard Shortcuts

When there is a **green** bar beside the current cell you are in **Edit** mode. When there is a **blue** bar beside the current cell you are in **Command** mode.

Double click a cell, or press Enter when it's highlighted to enter **Edit** mode. Now you are changing the contents of the cell.

Press Escape to return to **Command** mode, now you can use keyboard shortcuts to add, remove, change cell types etc.

In **Command** mode press 'h' to see a list of all keyboard shortcuts.

Some useful keyboard shortcuts are:

Press "Shift + Enter" to execute the current cell and move to the next one, or create a new one if this is the last cell.

Press "Ctrl + Enter" to execute the current cell and keep the same cell selected.

Add / delete cells:

- "a" to insert a new cell before
- "b" to insert a new cell after
- "d" TWICE to delete a cell

Toggle between code and text:

- "y" for code
- "m" for markdown

1-4. Strings

In Python we can use single quotes to denote a string

In [13]:
```
my_string_var = 'Hello'
```

We can also use double quotes to denote a string

In [14]:
```
another_string_var = "World!"
```

Strings can be concatenated using the + operator

In [15]:
```
hello_world = my_string_var + " " + another_string_var
hello_world
```

Out[15]:
```
'Hello World!'
```

We can find the length of a string

In [16]:
```
len(hello_world)
```

Out[16]:
```
12
```

Print can take an expression as an argument and print the expression's result

e.g. the length of A, " " and B

In [17]:
```
print( len(my_string_var + " " + another_string_var) )
```

```
12
```

Locate the position of the 1st occurrence of a particular character in a string

e.g. 'o'

In [18]:
```
hello_world.find('o')
```

Out[18]:
```
4
```

We can count the occurrences of a letter within a string

how many H's, I's and B's

In [19]:
```
print( hello_world.count('H') )
print( hello_world.count('I') )
print( hello_world.count('B') )
```

```
1
0
0
```

More String Functions

upper()

lower()

startswith()

endswith()

In [20]:
```python
# Create a "Hello World" string - concatenate A and B
hello_world = "Hello World"
hello_world
```

Out[20]:
```
'Hello World'
```

In [21]:
```python
# does lower case end with 'RLD'?
hello_world.lower().endswith('RLD')
```

Out[21]:
```
False
```

In [22]:
```python
# does upper case end with 'RLD'?
hello_world.upper().endswith('RLD')
```

Out[22]:
```
True
```

In [23]:
```python
# does upper case start with 'RLD'?
hello_world.upper().startswith('RLD')
```

Out[23]:
```
False
```

Documentation on common string operations can be found here - https://docs.python.org/3/library/string.html

Exercise 3

Complete the following steps in a single code cell

1. Create a string containing your first name and assign it to a variable called "first_name"
2. Create a string containing your last name and assign it to a variable called "last_name"
3. Print both strings on one line with a space between them
4. Concatenate the strings with a space between them, and store the result in a new variable called "full_name"
5. Print the length of the full_name string
6. Print how many times the first letter of your name appears in the entire full_name string

In [24]:
```python
# Do the exercise here
first_name = 'Guido'
last_name = 'van Rossum'

print(first_name + ' ' + last_name)

full_name = first_name + ' ' + last_name

print(len(full_name))
```

```
print( full_name.lower().count('g'))
```

```
Guido van Rossum
16
1
```

1-5. Slicing Strings

Slicing is a way to select a subsection of a string.

The same slicing notation is used in Python to get a subsection of many other types of data e.g. lists of data, tables of data etc.

Because of this, slicing is an important topic and a rudimentary understanding is needed before we move on to manipulating tables.

In Python, collections of data almost always call the first position 0 (ZERO) i.e. NOT 1

The general syntax for slicing is:

> string_to_slice [start : stop : step]

- **start** is the position to start the slice from
- **stop** is the position AFTER the end of the slice
- **step** is how many steps between each item (steps of 2, 3, 4, etc.)
    - If step is left out then a value of 1 is assumed

In [25]:
```
# check that we still have our "Hello World" string
hello_world
```

Out[25]:
```
'Hello World'
```

Now slice the string to display:

> lo Worl

In [26]:
```
hello_world[3:10]
```

Out[26]:
```
'lo Worl'
```

Now slice the string to display

> l ol

In [27]:
```
hello_world[3:10:2]
```

Out[27]:
```
'l ol'
```

If we leave out "start" then the beginning of the string is assumed. Use this to display

> Hello W

In [28]:
```
hello_world[:7]
```

Out[28]:
```
'Hello W'
```

If we leave out "start" then the beginning of the string is assumed. Use this to display

> lo World

In [29]:
```python
hello_world[3:]
```
Out[29]:
```
'lo World'
```

check if lower case of slice [3:10:2] ends with 'ol'?

In [ ]:
```python

```

Splitting Strings

the 'split' function splits on whitespace by default

In [30]:
```python
C = "Once upon a time."

C.split()
```
Out[30]:
```
['Once', 'upon', 'a', 'time.']
```

In [31]:
```python
# How would you split on ','?
currencies = "EUR, USD, JYP, GBP"

currencies.split(',')
```
Out[31]:
```
['EUR', ' USD', ' JYP', ' GBP']
```

Exercise 4

Complete the following steps in a single cell:

1. Create a string "Wrangling Data" in a variable called test_string
2. Using Slicing, print the following sections of the test_string:
   - Wran
   - Wrangling
   - gDt
   - Dat
3. Print the list of strings produced when you split this string at each space

In [32]:
```python
# Do the exercise here
test_string = "Wrangling Data"

print(test_string[:4])
print(test_string[:9])
print(test_string[8::2])
print(test_string[-4:-1])
```
```
Wran
Wrangling
gDt
Dat
```

2. Collections of Data

So far we have looked at variables representing an individual value e.g. a number, a string

Python has a number of ways to store "collections" of values.

The two most commonly used structures are:

- **List:** A simple ordered collection of values. Values do NOT need to be the same type.
- **Dictionary:** A collection of "key"/"value" pairs, often used as a type of "lookup" table

2-1 Lists

An ordered collection of values

> i.e. the order is maintained

are heterogenous

> i.e. vaues can be different types

are mutable

> i.e. CAN be changed

We create a list using **"square brackets":** [ ]

Empty List

In [33]:
```
# make an empty list called my_list

my_list = list()

my_list = []
```

List of Integers

In [34]:
```
# make my_list a list of integers

my_list = [2, 5, 4, 2, 4, 7]
```

List with mixed data types

In [35]:
```
# make my_list contain the values 1, "Hello" and 3.4
my_list = [1, "Hello", 3.4]
```

Nested Lists

i.e. lists can contain other lists

In [36]:
```
my_list.append(['test', 2.4])

print(my_list)
```

```
[1, 'Hello', 3.4, ['test', 2.4]]
```

List indexing

Similar to strings, we use square brackets to find an element in a list

In [37]:
```
my_list = ['p','r','o','b','e']
# print just the first value i.e. p
```

```
my_list[0]
```

Out[37]: 'p'

In [38]:
```
# print the third value i.e. o

my_list[2]
```

Out[38]: 'o'

In [39]:
```
# print the last value i.e. e

my_list[-1]
```

Out[39]: 'e'

In [40]:
```
# This cell will throw an Error!

# Try to make sense of the error message
# - usually the last line contains the most useful information.

# Only integers can be used for indexing

my_list[4.0]
# Cannot get list[4.5], list[4.22], etc.
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-40-52e6b29b1afc> in <module>
      6 # Only integers can be used for indexing
      7 
----> 8 my_list[4.0]
      9 # Cannot get list[4.5], list[4.22], etc.

TypeError: list indices must be integers or slices, not float
```

Lists can contain more lists, we use **more** square brackets to get a value from the "sub list"

In [41]:
```
# Nested List
n_list = ["Happy", [2,0,1,5]]

# how to access the sub-list? e.g. display 1

n_list[1][2]
```

Out[41]: 1

In [42]:
```
# Nested indexing

# Output: a
n_list[0][1]
```

Out[42]: 'a'

List slicing is the same a string slicing

In [43]:
```
my_list = ['d','a','t','a','f','l','a','m','e']
# Elements 3rd to 5th

my_list[2:5]
```

Out[43]: ['t', 'a', 'f']

In all Python slicing we can use negative numbers to start counting from the end

This works for strings also

In [44]:
```python
# Elements beginning to 4th

my_list[:4]
```

Out[44]:
```
['d', 'a', 't', 'a']
```

In [46]:
```python
# Elements 6th to end
my_list[5:]
```

Out[46]:
```
['l', 'a', 'm', 'e']
```

In [47]:
```python
# Elements beginning to end
my_list[:]
```

Out[47]:
```
['d', 'a', 't', 'a', 'f', 'l', 'a', 'm', 'e']
```

In [48]:
```python
# Length of my_list
len(my_list)
```

Out[48]:
```
9
```

Some more complicated slicing examples

In [49]:
```python
# All but the last?
my_list[:-1]
```

Out[49]:
```
['d', 'a', 't', 'a', 'f', 'l', 'a', 'm']
```

In [51]:
```python
# Just the last element without using negative indexing?
my_list[len(my_list) - 1:]
```

Out[51]:
```
['e']
```

Exercise 5

Complete the following steps in a single code cell:

- Create a list containing four strings: "a", "b", "c" and "d", store it in a variable called my_list
- Print the letter b from the list
- Print the letters b and c from the list
- Print the length of the list
- Print the last 3 elements in the list using negative indexes in your slice
- Create a second list with four numbers: 1, 2, 3, 4, store it in variable called my_numbers
- What happens if you "add" the two lists? E.g.
    - **my_big_list = my_list + my_numbers**
- Print the new variable my_big_list

In [60]:
```python
# Do the exercise here

my_list = ["a", "b", "c", "d"]

print(my_list[1])
print(my_list[1:3])
print(len(my_list))
```

```
print(my_list[-3:])

my_numbers = [1, 2, 3, 4]
print(my_numbers)

my_big_list = my_list + my_numbers
print(my_big_list)
```

```
b
['b', 'c']
4
['b', 'c', 'd']
[1, 2, 3, 4]
['a', 'b', 'c', 'd', 1, 2, 3, 4]
```

Main List Functions

Method	Description
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

2-2. Tuples

A tuple is like a 'light-weight' list i.e. another heterogeneous collection of values

i.e. can be different types - usually are different types

are immutable

i.e. cannot be changed

Python guaratees tuples are WRITE protected.

Create using ()

Empty tuple

In [61]:
```
my_tuple = ()
print(my_tuple)
```

```
()
```

tuple of integers

In [62]:
```
my_tuple = (3,4,5)
```

```
print(my_tuple)
```

```
(3, 4, 5)
```

tuple of mixed data types

In [63]:
```
my_tuple = (1, "Hello", 3.4)
print(my_tuple)
```

```
(1, 'Hello', 3.4)
```

Nested tuple

In [64]:
```
my_tuple = ("mouse", [8, 4, 6], (1, 2, 3))
print(my_tuple)
```

```
('mouse', [8, 4, 6], (1, 2, 3))
```

In [65]:
```
my_tuple = (1,2,3,4)

print (my_tuple)

my_tuple = (4,"five,", "six")
print(my_tuple)
```

```
(1, 2, 3, 4)
(4, 'five,', 'six')
```

Accessing elements inside a tuple - the same as list - use []

In [66]:
```
print(my_tuple[0])
```

```
4
```

Create tuples without ()

In [67]:
```
my_tuple = 3, 4.6, "dog"
print(my_tuple)
```

```
(3, 4.6, 'dog')
```

Unpacking a tuple

- Used when tuples are used as the return values from a function
- Widely used in the Machine Learning libraries

In [68]:
```
# this is a function called 'foo', it returns a tuple
def foo():
    return (3,4,5)

# we can automatically 'unpack' the tuple returned by the function
a, b, c = foo()
print(a)
print(b)
print(c)
```

```
3
4
5
```

Main Tuple Functions

Method	Description
count()	Returns the number of times a specified value occurs in a tuple
index()	Searches the tuple for a specified value and returns the position of where it was found

2-3. Sets

A heterogeneous collection of values i.e. no repeating values

Create using {}

Empty Set

In [69]:
```
my_set = {}
print(my_set)
```

```
{}
```

Set with initial Values

In [70]:
```
my_set = {1, "USD", 2, 'EUR'}
print(my_set)
```

```
{1, 2, 'USD', 'EUR'}
```

Create using set()

In [72]:
```
my_set = set({1, "USD", 2, 'EUR'})
print(my_set)
```

```
{1, 2, 'USD', 'EUR'}
```

Set of tuples

In [73]:
```
my_tuples = [(1,'FB'), (2,'AMZN')]
my_set = set(my_tuples)
```

Main Set Methods

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not
issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set

remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

2-4. Dictionary

A Python dictionary is a heterogeneous collection of key/values pairs

Quite often the key is a simple value e.g. an integer or string

The values are often complex types e.g. lists or more dictionaries

Keys do not need to be of the same type

> - Create a dictionary using **curly brackets:** { }
> - Put a **colon** between keys and values
> - Separate key/value pairs with a **comma**

Empty Dictionary

In [74]:
```
my_dict = dict()

# OR

my_dict = {}
```

A dictionary with integer keys and string values

> 1: 'USD', 2: 'EUR'

In [76]:
```
my_dict = {1: 'USD', 2: 'EUR'}

print(my_dict)
```

```
{1: 'USD', 2: 'EUR'}
```

A dictionary with mixed keys (string and integer) and mixed values (string and list)

> 'FB': 'a_string_value', 1: [2, 3, 4]

In [78]:
```
my_mixed_dict = { 'FB': 'a_string_value', 1: [2, 3, 4]}

print(my_mixed_dict)
```

```
{'FB': 'a_string_value', 1: [2, 3, 4]}
```

Accessing dictionary elements

We access the elements of a dictionary by putting the required key inside square brackets

In [82]:
```
# get the VALUE that corresponds to the KEY 'FB'
```

```
print(my_mixed_dict['FB'])

# get the VALUE that corresponds to the KEY 1
print(my_mixed_dict[1])
```

```
a_string_value
[2, 3, 4]
```

Using name value pairs to describe attributes of a Stock

In [89]:
```
my_dict = {'FB': 'Facebook', 'Adj Close': 123.45}


# Output: Adj Close
print( list(my_dict.keys())[1] )

print(my_dict['Adj Close'])


# Update Adj Close to be 234.56
my_dict['Adj Close'] = 234.56



# Add item 'Vol' = equal to 525000
my_dict['Vol'] = 525000

print(my_dict)


# Remove 'Adj Close'

my_dict.pop('Adj Close')
print(my_dict)
```

```
Adj Close
123.45
{'FB': 'Facebook', 'Adj Close': 234.56, 'Vol': 525000}
{'FB': 'Facebook', 'Vol': 525000}
```

Dictionaries as lookup tables

In [90]:
```
rates = {'EUR': 1.09, 'USD': 1.28 }

# Print the rate for EUR
rates['EUR']
```

Out[90]:
```
1.09
```

Iteration on collections

Here we look at ways to "loop" over the contents of a collection to operate on each individual element

In [91]:
```
# Over a list
wordlist = ["There", "was", "an", "old", "woman", "who", "lived", "in", "a", "shoe."]

# iterate over the list to print each word
for word in wordlist:
    print(word)
```

```
There
was
an
old
woman
who
lived
in
a
shoe.
```

In [92]:

```python
# Over a tuple
wordtuple = ("She", "had", "so", "many", "children", "she", "didn't", "know", "what",

# it works exactly the same for other collections - e.g. tuple
for word in wordlist:
    print(word)
```

```
There
was
an
old
woman
who
lived
in
a
shoe.
```

In [93]:

```python
# Over a set
wordset = {"She", "gave", "them", "some", "broth", "and", "a", "big", "slice", "of",

# same for set
for word in wordlist:
    print(word)
```

```
There
was
an
old
woman
who
lived
in
a
shoe.
```

In [96]:

```python
# Over a dictionary
worddict = { '1':"Then", "Two":"kissed",  3:"them",  'Four':"all",  5:"soundly",
             'VI':"and", "seven":'sent', 8:'them', "9":'to', 'X':'bed.'}


# a dictionary defaults to iterate over keys, but we have options to get values or bo
for word in worddict.values():
    print(word)
```

```
Then
kissed
them
all
soundly
and
sent
them
to
bed.
```

2-6. List comprehensions

List comprehensions provide a concise way to create lists.

It consists of brackets containing an expression followed by a for clause, then zero or more for or if clauses. The expressions can be anything, meaning you can put in all kinds of objects in lists.

The result will be a new list resulting from evaluating the expression in the context of the for and if clauses which follow it.

The list comprehension always returns a result list.

In [97]:

```
# A function to check if a word has more than 5 characters
def len_gt_five(word):
    return len(word) > 5

# A lot of words in a paragraph of text
verse = '''There was an old woman who lived in a shoe.
She had so many children, she didn't know what to do.
She gave them some broth and a big slice of bread.`b
Then kissed them all soundly and sent them to bed.'''
```

In [102…

```
print(len_gt_five(verse))

len_gt_five('shortr')
```

```
True
```

Out[102]:

```
True
```

You could find all words longe than 5 characters like this:

In [103…

```
wordslist = verse.split()

long_words = []
for word in wordslist:
    if len_gt_five(word):
        long_words.append(word)

print(long_words)
```

```
['children,', "didn't", 'bread.', 'kissed', 'soundly']
```

You can obtain the same thing using list comprehension:

In [104…

```
long_words = [word for word in verse.split() if len_gt_five(word)]

print(long_words)
```

```
['children,', "didn't", 'bread.', 'kissed', 'soundly']
```

3. Python Logical Statements

3-1 Conditionals

We often want to test if a condition is True or False.

There are a number of conditional operators in Python:

- **==** : Check if two elements are equal
- **!=** : NOT Equal

- **>** : Greater than
- **>=** : Greater than OR equal to
- **<** : Less than
- **<=** : Less than OR equal to

> Notice the difference between "a single equals" and "double equals"
>
> - A single equals makes a variable equal to something (assignment)
> - Double equals is "asking" if two statements are the same (querying)

Some conditional checks

In [106…

```
x = 10
y = 12

# Output: x == y is False
print(x == y)

# Output: x != y is True
print(x != y)

# Output: x > y is False
print(x > y)

# Output: x < y is True
print(x < y)

# Output: x >= y is False
print(x >= y)


# Output: x<= y is True

print(x <= y)
```

```
False
True
False
True
False
True
```

3-2. Branching with "if"

We can use "if" or "if/else" to execute lines of code based on whether a condition is True or False.

> Notice that Python uses indentation to show that code is "inside" an "if" statement.
> Many other languages use brackets for this.

In [110…

```
num = -5

# Test is num greater than 0, print something if it is, print something else anyway
if num > 0:
    print("Num is greater than 0")

print('This always is printed')
```

```
This always is printed
```

3-3. Branching with "if/else"

We can also add an "else"

In [ ]:
```
num = 3

# Test for either positive or Zero, print something different in each case
# Test is num greater than 0, print something if it is, print something else anyway
if num > 0:
    print("Num is greater than 0")
else:
    print('Small is perfect')
```

Testing if a value is "in" a collection

Python also has easy to read comparisons to check if a value is in a collection

- in
- not in

e.g.

- Check if the letter 'a' is in a list:
    - **'a' in my_list**
- Check if the key 'b' is in a dictionary:
    - **'b' in my_dict**

In [111…
```
rates = {'EUR': 1.09, 'USD': 1.28, 'JPY': 99.25}

# check if the key 'JPY' is in the dictionary 'rates', print something in each case
if 'JPY' in rates:
    print("rates has the correct symbols")
```

```
rates has the correct symbols
```

Exercise 6

Complete the following steps in a single cell

1. Create a dictionary called my_dict, containing the following data:
    - {'first' : 1, 'second': 2, 'third': 3}
2. Print the **value** corresponding to the **key** "second"
3. Print the result of
    - my_dict['third'] >= 3
4. Write an "if" block that prints "Success" if the key 'third' is in the dictionary my_dict

In [115…
```
my_dict = {'first': 1, 'second': 2, 'third': 3}

print(my_dict['second'])

print(my_dict['third'] >= 3)

if 'third' in my_dict:
    print('Success')
```

```
2
True
Success
```

4. Functions

4-1. User Defined functions

What Are User-Defined Functions in Python? Functions that we define ourselves to do a certain task are referred to as user-defined functions. The way in which we define and call functions in Python has already been discussed.

Functions that readily come with Python are called built-in functions. If we use functions written by others via a library they are called library functions.

All the other functions that we write on our own fall under user-defined functions. So, our user-defined function could be a library function to someone else.

Advantages of User-Defined Functions User-defined functions help to decompose a large program into small segments, which makes the program easy to understand, maintain, and debug. If repeated code occurs in a program, the function can be used to include those codes and execute when needed by calling that function. Programmers working on a large project can divide the workload by making different functions.

Syntax

> `def function_name(argument1, argument2, ...) :`
> `statement_1`
> `statement_2`
> `....`

In [117…
```python
# write a function that adds two numbers and then call it
def add_two_numbers(num1, num2):
    return num1 + num2

print(add_two_numbers(345, 6665))
```

```
7010
```

4-2. Python Lambda Functions

What Are Lambda Functions?
In Python, an anonymous function is a function that is defined without a name.

While normal functions are defined using the defkeyword, in Python anonymous functions are defined using the lambda keyword.

Hence, anonymous functions are also called `lambda` functions.

How To Use Lambda Functions In Python?
A Lambda function in Python has the following syntax:

> `lambda arguments: expression`

Lambda functions can have any number of arguments but only one expression.
expression is evaluated and returned.
Lambda functions can be used wherever function objects are required.

In [122…
```python
# create a lambda than multiplies a number by 2

list(map(lambda x: x*2, [25, 430]))
```

Out[122]: [50, 860]