PROGRAM STUDI INFORMATIKA
FAKULTAS TEKNIK DAN INFORMATIKA
UNIVERSITAS MULTIMEDIA NUSANTARA
SEMESTER GENAP TAHUN AJARAN 2021/2022

IF420 – ANALISIS NUMERIK

Pertemuan ke 1 – Python Basics

Seng Hansun, S.Si., M.Cs.

Capaian Pembelajaran Mingguan Mata Kuliah (Sub-CPMK):

Sub-CPMK 1: Mahasiswa mampu menerapkan dasar-dasar pemrograman Python – C3

Outlines

- Backgrounds
- Python Basics
- Variables and Basic Data Structures
- Functions
- Branching Statements
- Iteration

Backgrounds

- The outline of this course follows the standard material taught at the University of California, Berkeley, which can be found at https://pythonnumericalmethods.berkeley.edu/notebooks/Index.html
- This course has two fundamental goals:
  1. Teach **Python programming** to science and engineering students who do not have prior exposure to programming
  2. Introduce a variety of **numerical analysis** tools that are useful for solving science and engineering problems
- Therefore, we will cover a wide range of topics, but no topic is covered in great depth.
- Prerequisite knowledges:
  - Understanding of the computer monitor and keyboard/mouse input devices
  - Understanding of the folder structure used to store files in most operating systems
  - High school level algebra and trigonometry
  - Introductory, college-level calculus

Python Basics

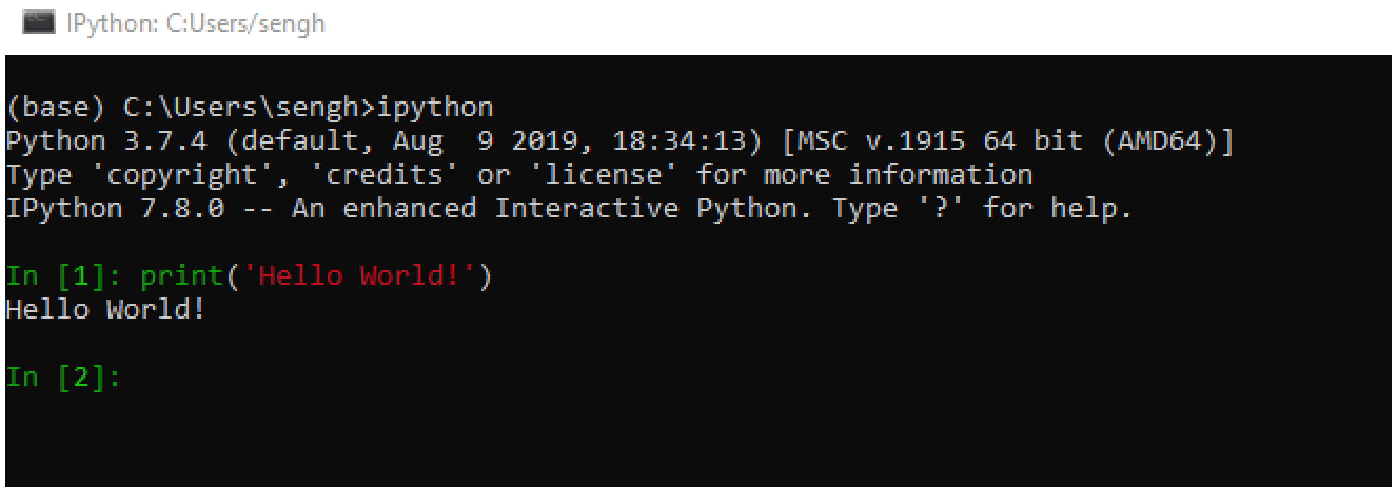
- Before we start to use Python, we need to set up our Python working environment on the computer.
- There are different ways to install Python and related packages, here we recommend to use **Anaconda** or **Miniconda** to install and manage your packages.
- Anaconda is a complete distribution framework that includes the Python interpreter, package manager as well as the commonly used packages in scientific computing (https://www.anaconda.com/products/individual).
- Miniconda is a light version of Anaconda that does not include the common packages, therefore, you need to install all the different packages by yourself. But it does have the Python interpreter and package manager (https://docs.conda.io/en/latest/miniconda.html).

Running the Python Codes

- There are different ways to run Python code, they all have different usages.

1. Using **Python shell** or **Ipython shell**

- The easiest way to run Python code is through the Python shell or Ipython Shell (which stands for Interactive Python).
- The Ipython shell is richer than Python shell, such as Tab autocompletion, color-highlighted error messages, basic UNIX shell integration and so on.


IPython: C:Users/sengh

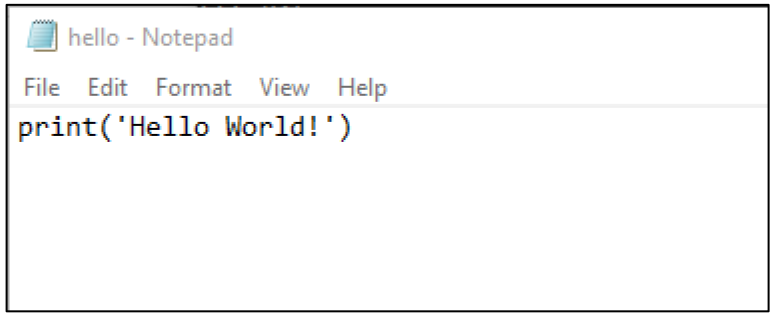
```
(base) C:\Users\sengh>ipython
Python 3.7.4 (default, Aug  9 2019, 18:34:13) [MSC v.1915 64 bit (AMD64)]
Type 'copyright', 'credits' or 'license' for more information
IPython 7.8.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]: print('Hello World!')
Hello World!

In [2]:
```

Running the Python Codes

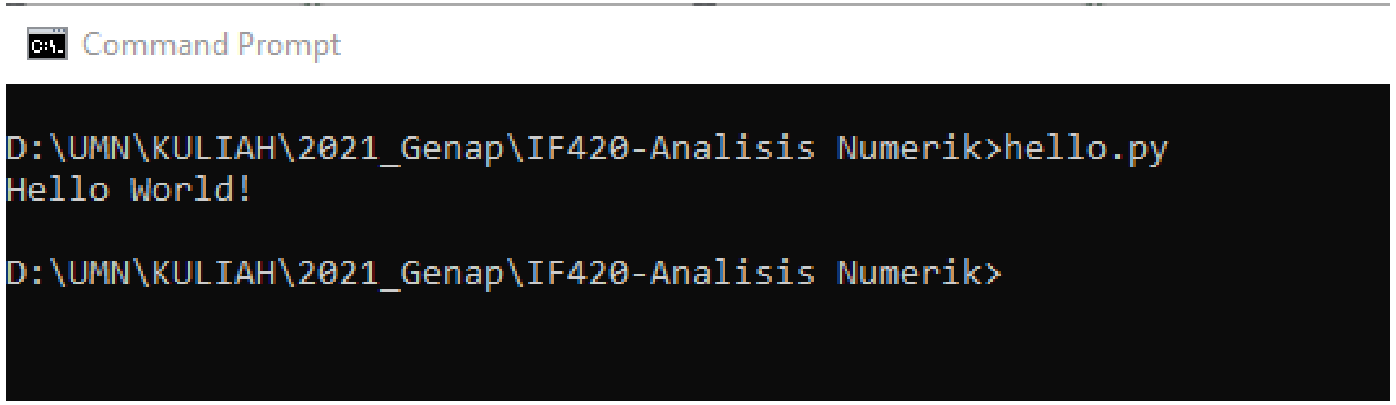
- There are different ways to run Python code, they all have different usages.

2. Run Python script/file from **command line**

- The second way to run Python code is to put all the commands into a file and save it as a file with extension **.py** (the extension of the file could be anything, but by convention, it is usually .py).
- Then just run it from terminal.



```
print('Hello World!')
```



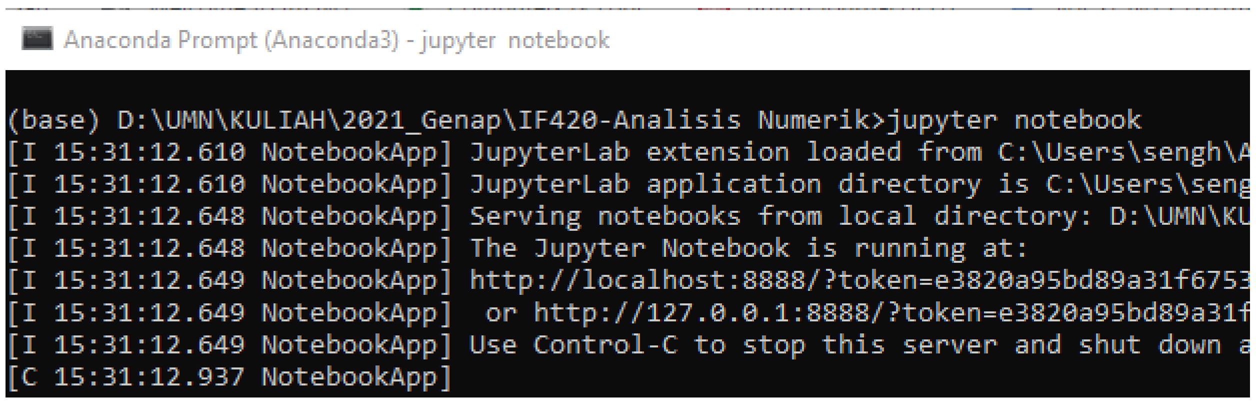
```
D:\UMN\KULIAH\2021_Genap\IF420-Analisis Numerik>hello.py
Hello World!

D:\UMN\KULIAH\2021_Genap\IF420-Analisis Numerik>
```

Running the Python Codes

- There are different ways to run Python code, they all have different usages.

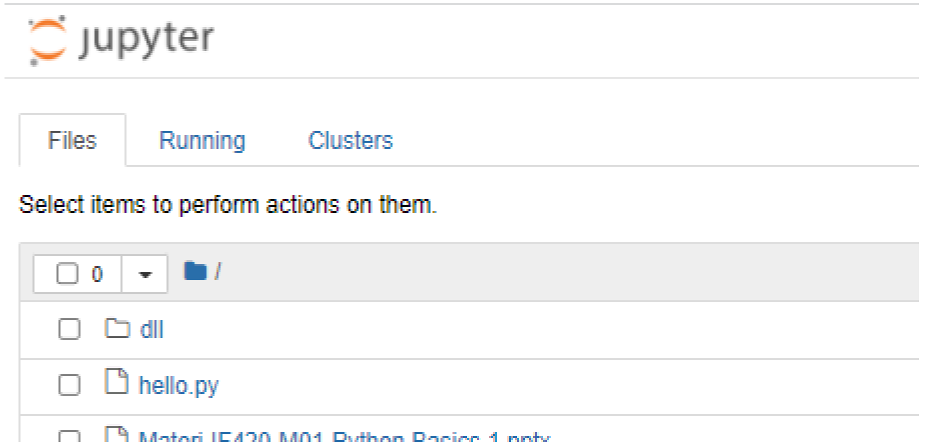
3. Using **Jupyter Notebook**

- The third way to run Python is through Jupyter notebook.
- It is a very powerful browser-based Python environment.
- Run the Jupyter notebook in the bash command line:



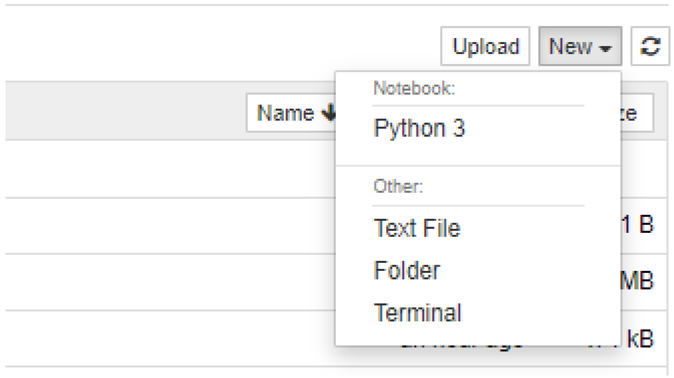
Anaconda Prompt (Anaconda3) - jupyter notebook

```
(base) D:\UMN\KULIAH\2021_Genap\IF420-Analisis Numerik>jupyter notebook
[I 15:31:12.610 NotebookApp] JupyterLab extension loaded from C:\Users\sengh\A
[I 15:31:12.610 NotebookApp] JupyterLab application directory is C:\Users\seng
[I 15:31:12.648 NotebookApp] Serving notebooks from local directory: D:\UMN\KU
[I 15:31:12.648 NotebookApp] The Jupyter Notebook is running at:
[I 15:31:12.649 NotebookApp] http://localhost:8888/?token=e3820a95bd89a31f6753
[I 15:31:12.649 NotebookApp]  or http://127.0.0.1:8888/?token=e3820a95bd89a31f
[I 15:31:12.649 NotebookApp] Use Control-C to stop this server and shut down a
[C 15:31:12.937 NotebookApp]
```

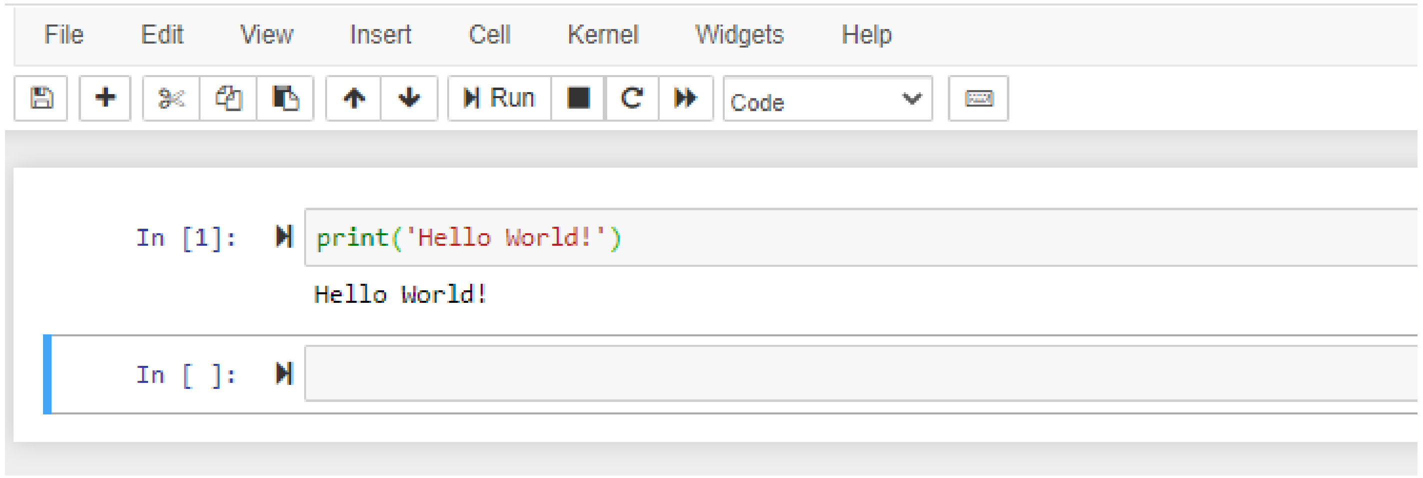


Running the Python Codes

- From the upper right button to create a new **Python3** notebook:

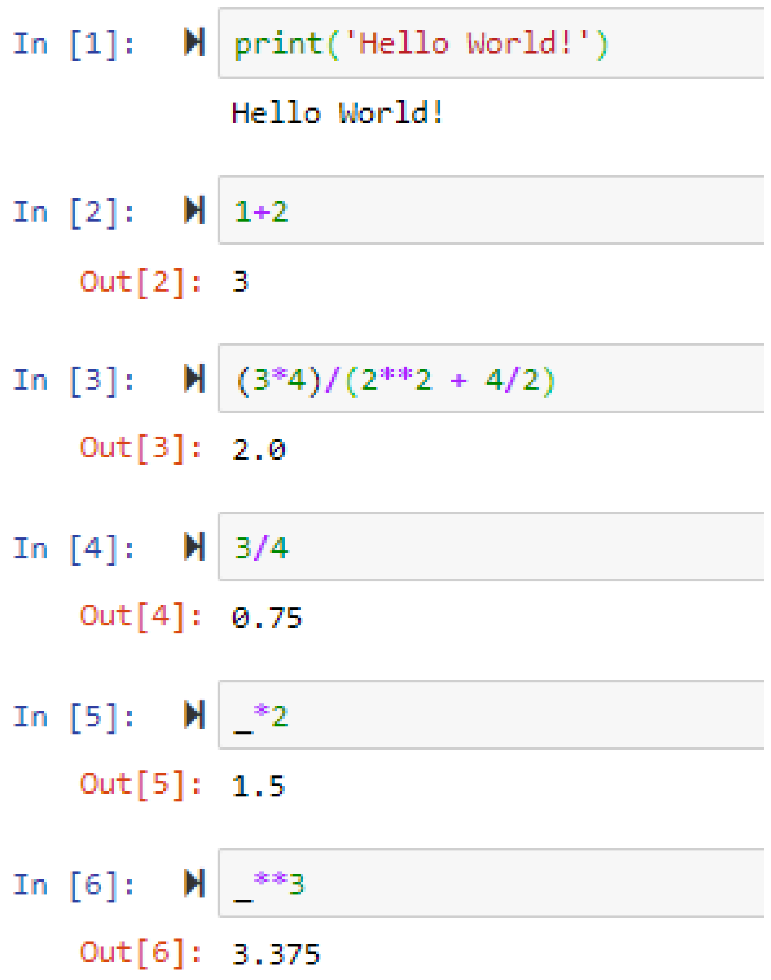


- Running code in Jupyter notebook is easy, you type your code in the cell, and press **shift + enter** to run the cell, the results will be shown below the code.



```
In [1]:  print('Hello World!')

         Hello World!

In [ ]:
```

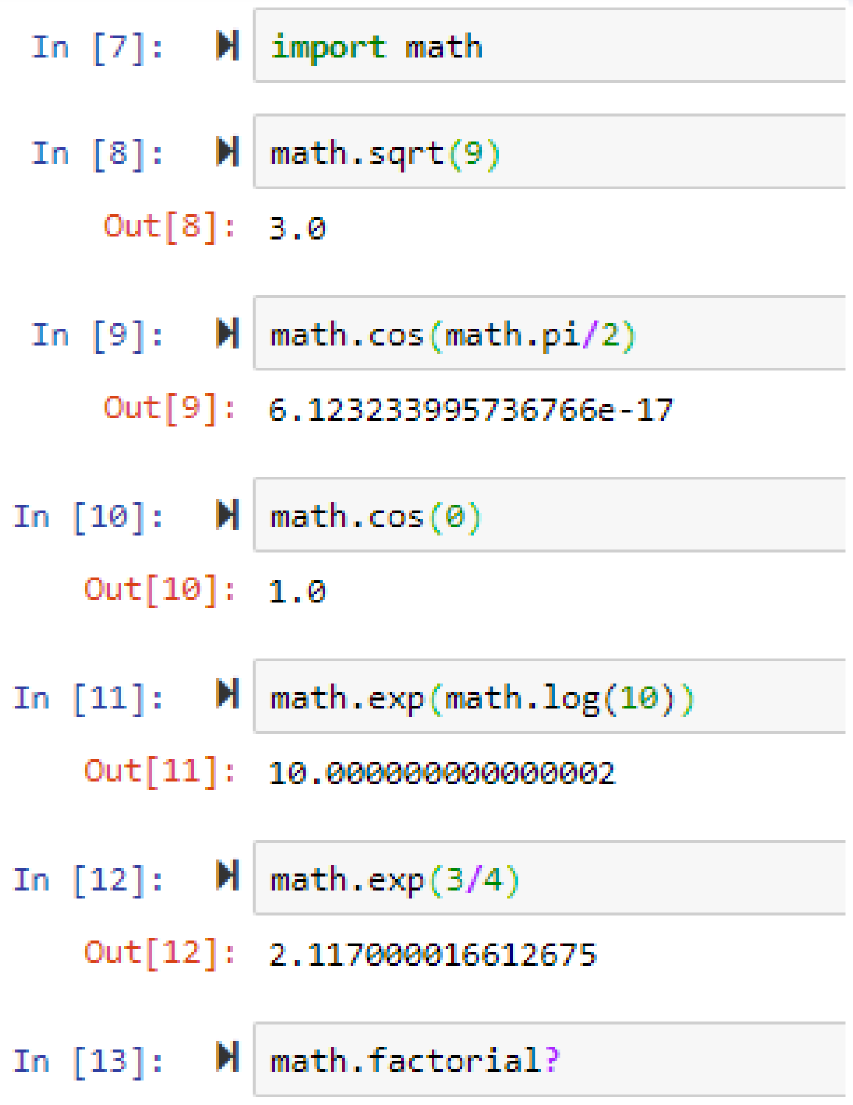
Python as a Calculator



```
In [1]:  print('Hello World!')

         Hello World!

In [2]:  1+2

Out[2]:  3

In [3]:  (3*4)/(2**2 + 4/2)

Out[3]:  2.0

In [4]:  3/4

Out[4]:  0.75

In [5]:  _*2

Out[5]:  1.5

In [6]:  _**3

Out[6]:  3.375
```

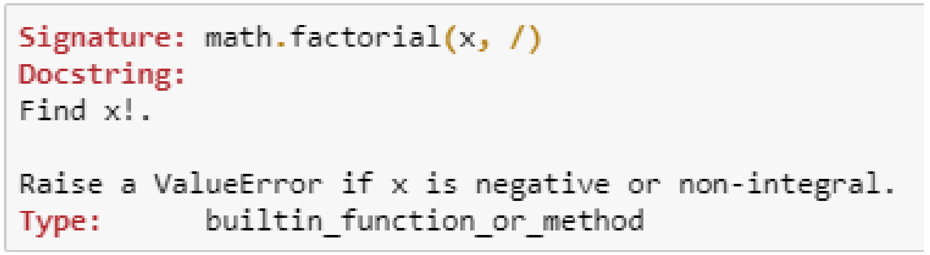
- An **arithmetic operation** is either addition, subtraction, multiplication, division, or powers between two numbers.
- An **arithmetic operator** is a symbol that Python has reserved to mean one of the aforementioned operations.
- These symbols are **+** for addition, **-** for subtraction, ***** for multiplication, **/** for division, and ****** for exponentiation.
- An **order** of operations is a standard order of precedence that different operations have in relationship to one another.
- Powers are executed before multiplication and division, which are executed before addition and subtraction.
- Parentheses, **()**, can also be used to supersede the standard order of operations.

Python as a Calculator



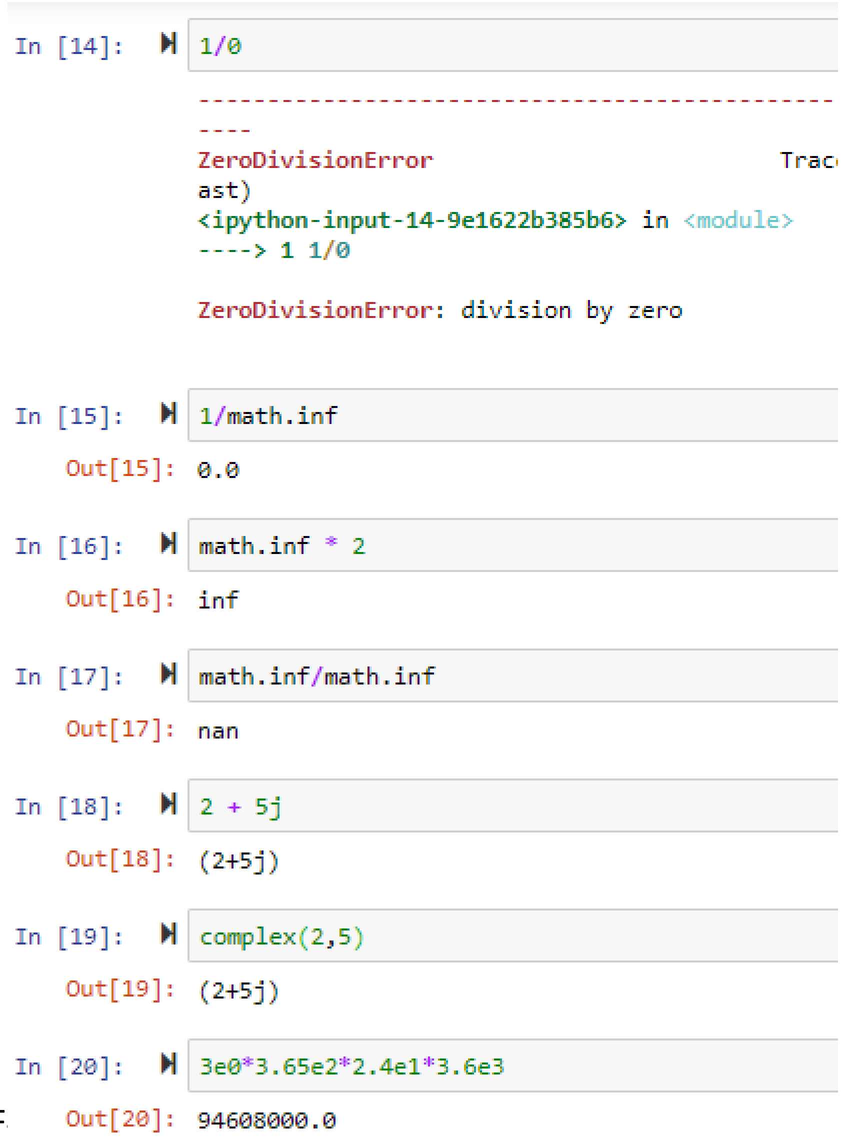
```
In [7]:  import math

In [8]:  math.sqrt(9)
Out[8]:  3.0

In [9]:  math.cos(math.pi/2)
Out[9]:  6.123233995736766e-17

In [10]: math.cos(0)
Out[10]: 1.0

In [11]: math.exp(math.log(10))
Out[11]: 10.000000000000002

In [12]: math.exp(3/4)
Out[12]: 2.117000016612675

In [13]: math.factorial?
```

- Python has many basic arithmetic functions like sin, cos, tan, asin, acos, atan, exp, log, log10 and sqrt stored in a module called **math**.
- We can import this module first to get access to these functions.
- The way we use these mathematical functions is **module.function**, the inputs to them are always placed inside of parentheses that are connected to the function name.



```
Signature: math.factorial(x, /)
Docstring:
Find x!.

Raise a ValueError if x is negative or non-integral.
Type:      builtin_function_or_method
```

Python as a Calculator



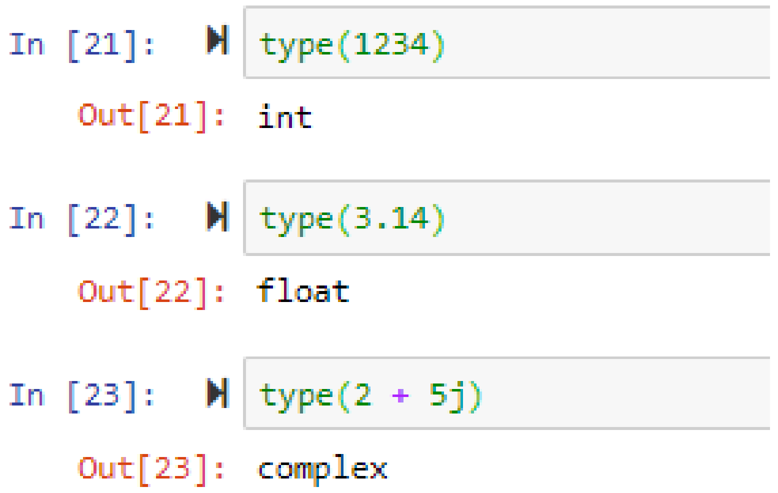
```
In [14]:  1/0

          ----------------------------------------------
          ----
          ZeroDivisionError                          Trac
          ast)
          <ipython-input-14-9e1622b385b6> in <module>
          ----> 1 1/0

          ZeroDivisionError: division by zero

In [15]:  1/math.inf
  Out[15]: 0.0

In [16]:  math.inf * 2
  Out[16]: inf

In [17]:  math.inf/math.inf
  Out[17]: nan

In [18]:  2 + 5j
  Out[18]: (2+5j)

In [19]:  complex(2,5)
  Out[19]: (2+5j)

In [20]:  3e0*3.65e2*2.4e1*3.6e3
  Out[20]: 94608000.0
```

- Python will raise an ZeroDivisionError when you have expression 1/0, which is **infinity**.
- You can type **math.inf** at the command prompt to denote infinity or **math.nan** to denote something that is not a number that you wish to be handled as a number.
- Finally, Python can also handle the **imaginary** number (in Python imaginary part is using j instead of i).
- Another way to represent complex number in Python is to use the **complex function**.
- Python can also handle scientific notation using the letter **e** between two numbers.
- For example, 1e6=1000000 and 1e−3=0.001.

Basic Data Types

- In Python, there are a few data types we need to know, for numerical values, **int**, **float**, and **complex** are the types associated with the values.

  - int: Integers, such as 1, 2, 3, …
  - float: Floating-point numbers, such as 3.2, 6.4, …
  - complex: Complex numbers, such as 2 + 5j, 3 + 2j, …



```
In [21]:  type(1234)
Out[21]:  int

In [22]:  type(3.14)
Out[22]:  float

In [23]:  type(2 + 5j)
Out[23]:  complex
```

- You can use function **type** to check the data type for different values.
- Of course, there are other different data types, such as **boolean**, **string** and so on.

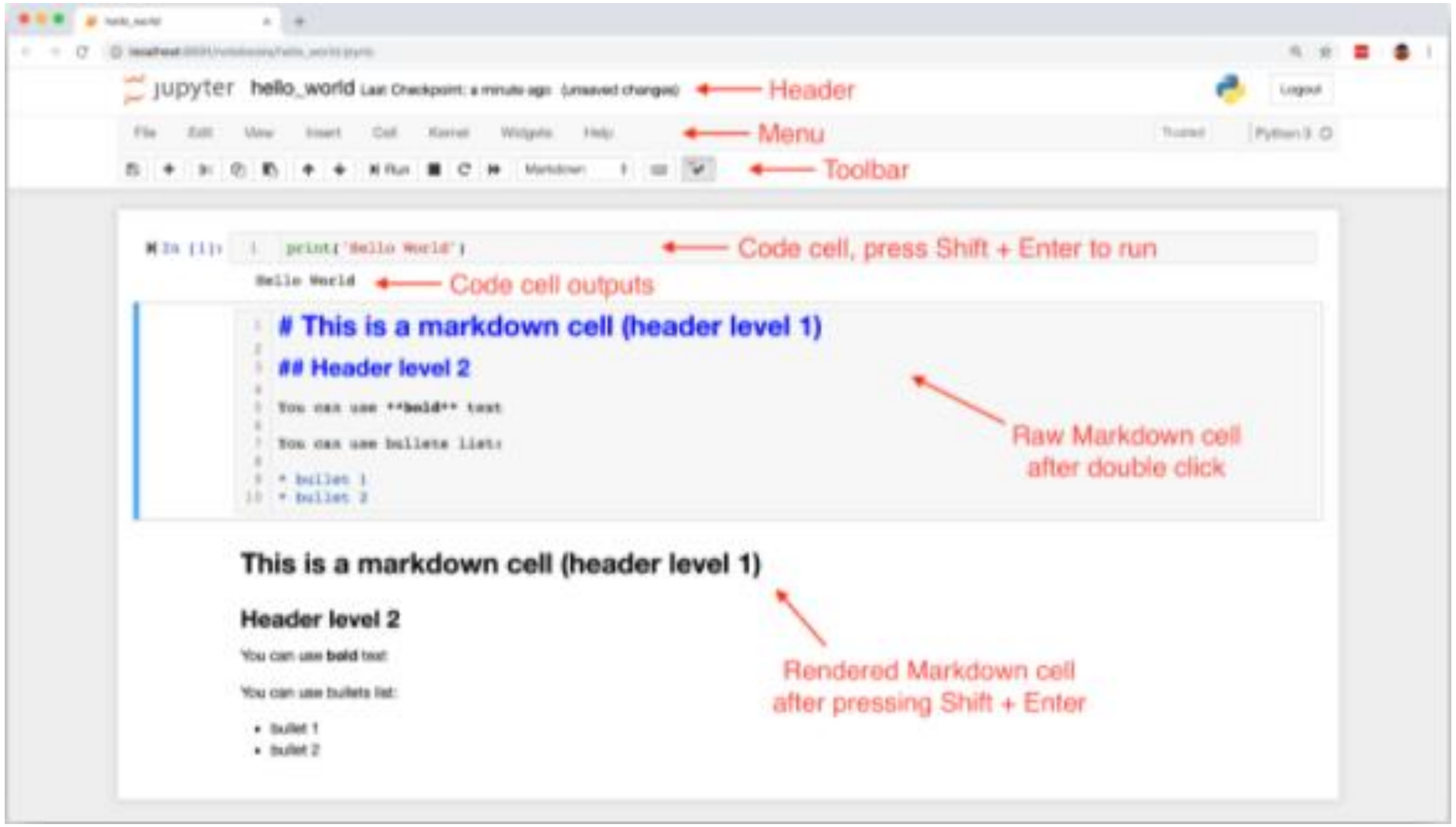
Managing Packages

- One feature makes Python really great is the various **packages/ modules** developed by the community.
- Most of the time, when you want to use some functions or algorithms, you will find there maybe already multiple packages from the community coded that for you, and all you need to do is to install the packages and use them in your code.
- Therefore, managing packages is one of the most important skills you need to learn to fully take advantage of Python.
- **Pip** is a package manager that automates the process of installing, updating, and removing the packages.
- It could install packages that published on Python Package Index (PyPI).

Introduction to Jupyter Notebook

- The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text.
- Uses include: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.
- Jupyter notebook is running using your browser, it could run locally on your machine as a local server or remotely on a server.
- The reason it is called **notebook** is because it can contain live code, rich text elements such as equations, links, images, tables, and so on.
- Therefore, you could have a very nice notebook to describe your idea and the live code all in one document.

Introduction to Jupyter Notebook



jupyter
hello_world Last Checkpoint: a minute ago (unsaved changes)
Header
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3
Logout
Menu
Run
Markdown
Toolbar
Code cell, press Shift + Enter to run
Code cell outputs
Raw Markdown cell after double click
Rendered Markdown cell after pressing Shift + Enter

```
In [1]:  print('Hello World')
```

```
Hello World
```

```
# This is a markdown cell (header level 1)

## Header level 2

You can use **bold** text

You can use bullets list:

* bullet 1
* bullet 2
```

This is a markdown cell (header level 1)

Header level 2

You can use **bold** text

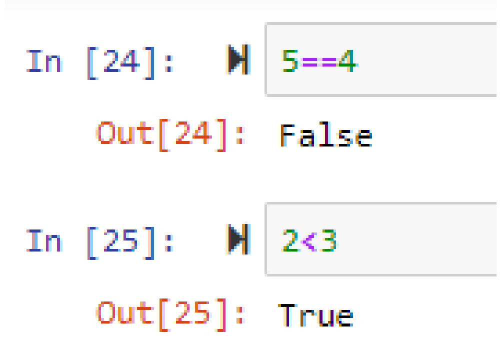
You can use bullets list:

- bullet 1
- bullet 2

https://pythonnumericalmethods.berkeley.edu/notebooks/chapter01.04-Introduction-to-Jupyter-Notebook.html

Logical Expressions and Operators

- A **logical expression** is a statement that can either be true or false. For example, $a < b$ is a logical expression. It can be true or false depending on what values of a and b are given. Note that this differs from a **mathematical expression** which denotes a truth statement. In the previous example, the mathematical expression $a < b$ means that a is less than b, and values of a and b where $a \geq b$ are not permitted.
- In Python, a logical expression that is true will compute to the value "**True**". A false expression will compute to the value "**False**". This is a new data type we come across - **boolean**, which has the built-in values True and False.

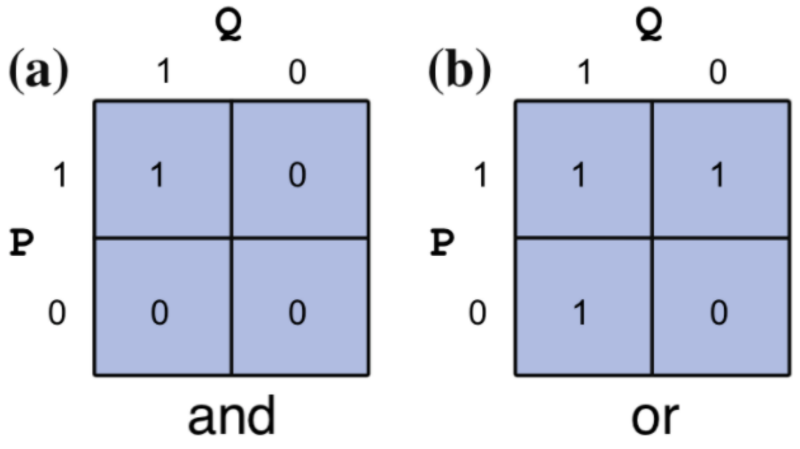


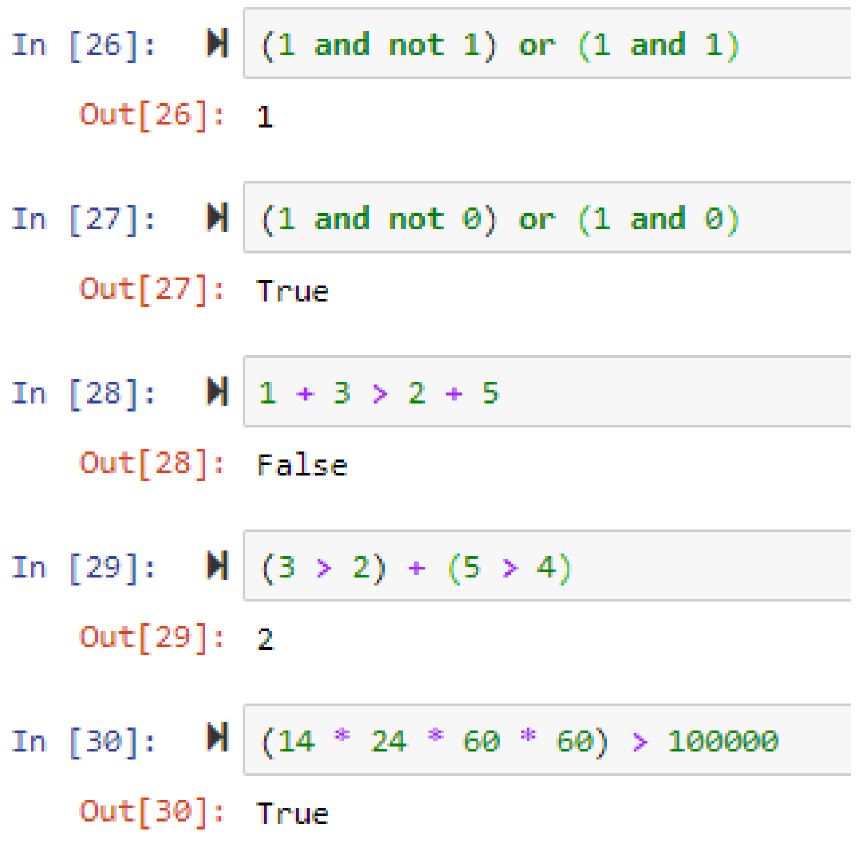
```
In [24]:  5==4
  Out[24]: False

In [25]:  2<3
  Out[25]: True
```

- **Comparison operators** compare the value of two numbers, and they are used to build logical expressions. Python reserves the symbols **>**, **>=**, **<**, **<=**, **!=**, **==**, to denote "greater than", "greater than or equal", "less than", "less than or equal", "not equal", and "equal", respectively.

Logical Expressions and Operators

- Logical operators are operations between two logical expressions that, for the sake of discussion, we call **P** and **Q**. The fundamental logical operators we will use herein are **and**, **or**, and **not**.
- The **truth table** of a logical operator or expression gives the result of every truth combination of P and Q.
- The truth tables for "and" and "or" are given in the following figure.





```
In [26]:  (1 and not 1) or (1 and 1)
Out[26]: 1

In [27]:  (1 and not 0) or (1 and 0)
Out[27]: True

In [28]:  1 + 3 > 2 + 5
Out[28]: False

In [29]:  (3 > 2) + (5 > 4)
Out[29]: 2

In [30]:  (14 * 24 * 60 * 60) > 100000
Out[30]: True
```

https://pythonnumericalmethods.berkeley.edu/notebooks/chapter01.05-Logial-Expressions-and-Operators.html

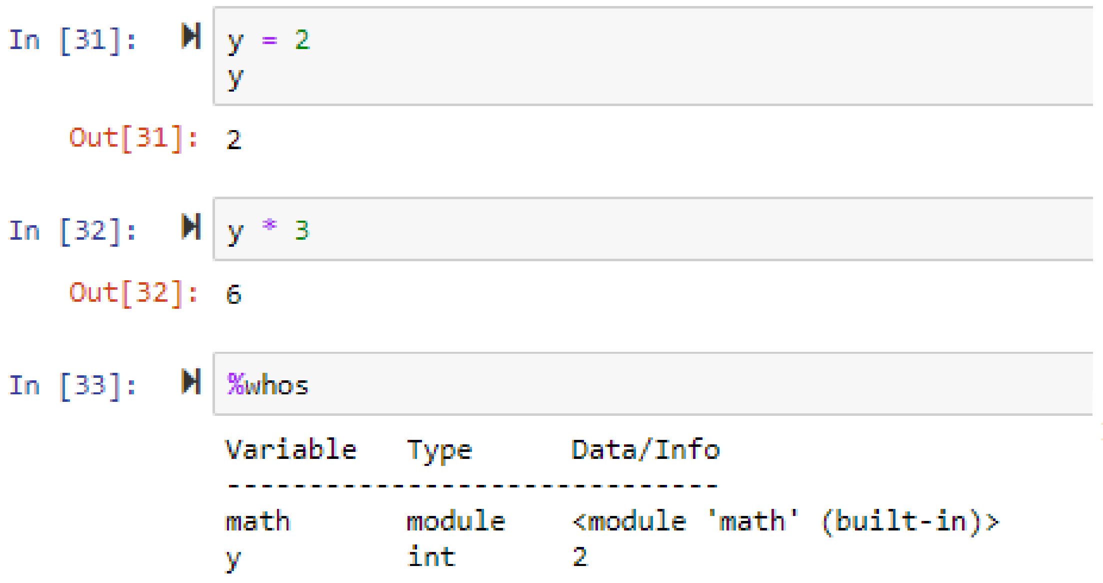
Variables and Basic Data Structures

- Currently, technology can acquire information from the physical world at an enormous rate. For example, there are sensors that can take tens of thousands of pressure, temperature, and acceleration readings per second.
- To make sense of all this data and process it in a way that will help solve science and engineering problems requires storing information in data structures that you and Python can easily work with.
- Variables are used in Python to **store and work** with data.
- However, data can take **many forms**, such as numbers, words, or have a more complicated structure.
- It is only natural that Python would have different kinds of variables to hold different kinds of data.

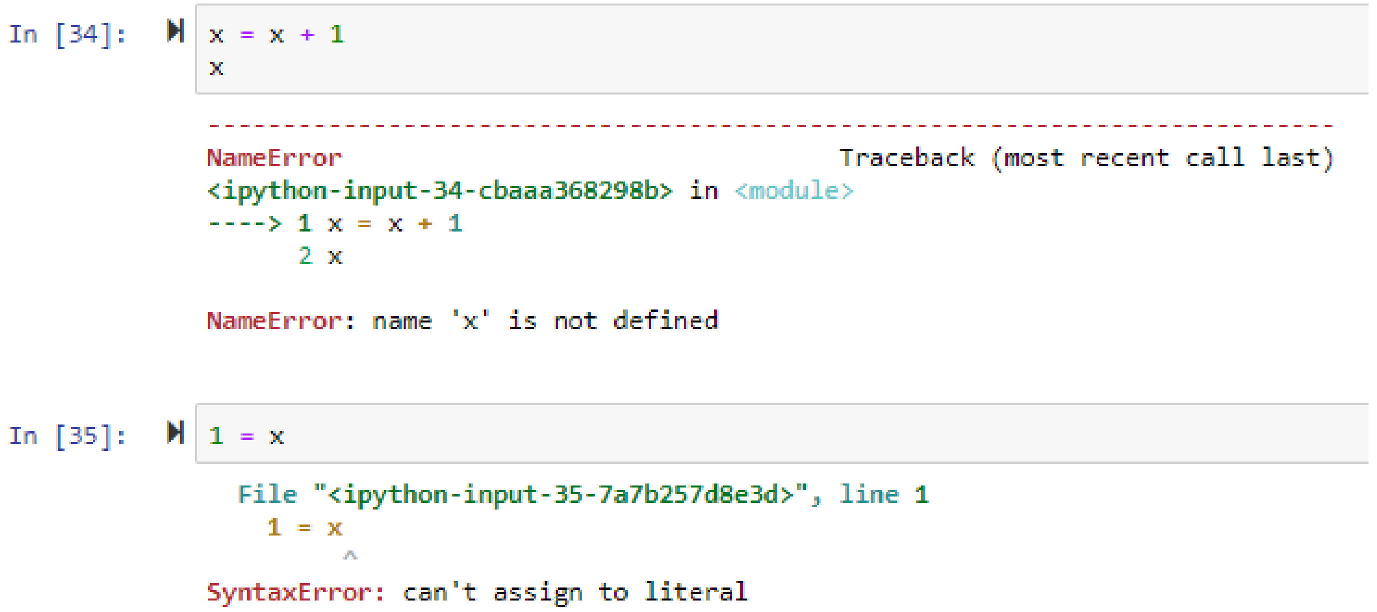
Variables and Assignment

- A **variable** is a string of characters and numbers associated with a piece of information.
- The **assignment operator**, denoted by the "**=**" symbol, is the operator that is used to assign values to variables in Python.
- You can view a list of all the variables in the notebook using the magic command **%whos**.
- Although it is perfectly valid to say $1 = x$ in mathematics, assignments in Python always **go left**: meaning the value to the right of the equal sign is assigned to the variable on the left of the equal sign.
- Variables can only contain alphanumeric characters (letters and numbers) as well as underscores. However, the first character of a variable name must be a letter or underscores. Spaces within a variable name are not permitted, and the variable names are case-sensitive.

Variables and Assignments



```
In [31]:  y = 2
          y

Out[31]:  2

In [32]:  y * 3

Out[32]:  6

In [33]:  %whos

          Variable   Type      Data/Info
          ------------------------------
          math       module    <module 'math' (built-in)>
          y          int       2
```



```
In [34]:  x = x + 1
          x

          ---------------------------------------------------------------------------
          NameError                                 Traceback (most recent call last)
          <ipython-input-34-cbaaa368298b> in <module>
          ----> 1 x = x + 1
                2 x

          NameError: name 'x' is not defined

In [35]:  1 = x

            File "<ipython-input-35-7a7b257d8e3d>", line 1
              1 = x
                 ^
          SyntaxError: can't assign to literal
```

Variables and Assignment

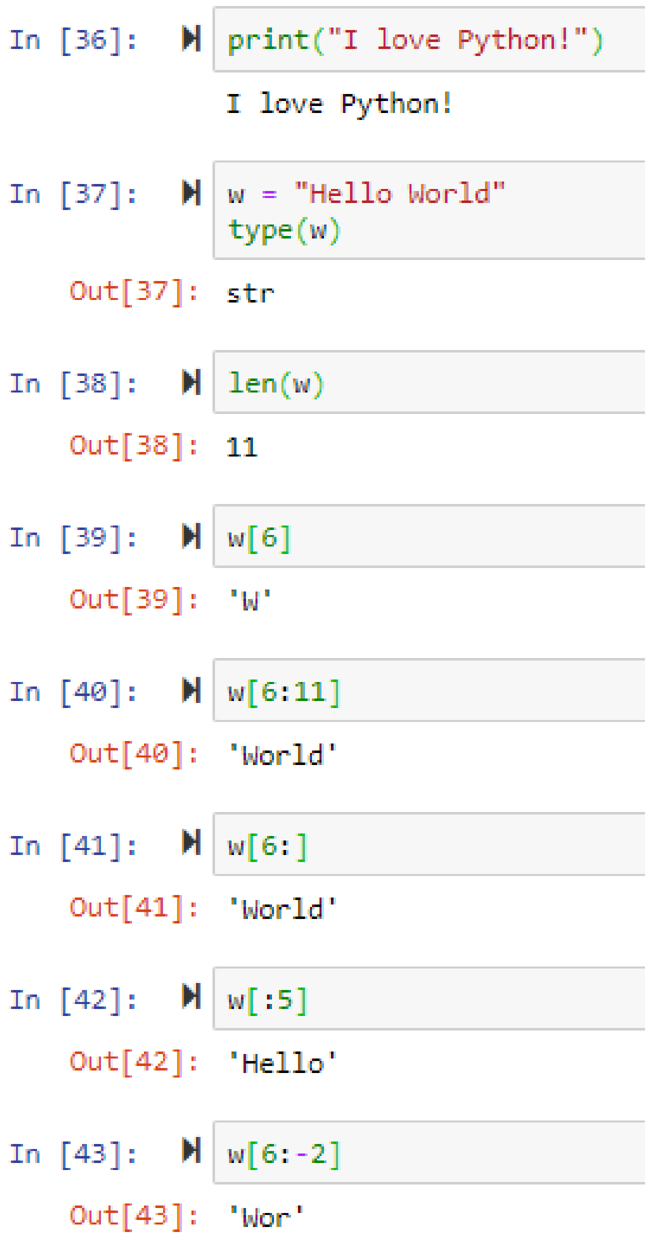
- Now that you know how to assign variables, it is important that you learn to never leave **unassigned commands**. An unassigned command is an operation that has a result, but that result is not assigned to a variable.
- You can clear a variable from the notebook using the **del** function. Typing **del x** will clear the variable **x** from the workspace. If you want to remove all the variables in the notebook, you can use the magic command **%reset**.
- In programming, variables are associated with a value of a certain type. There are many data types that can be assigned to variables.
- A **data type** is a classification of the type of information that is being stored in a variable.
- The basic data types that you will utilize throughout this lesson are **boolean**, **int**, **float**, **string**, **list**, **tuple**, **dictionary**, **set**.

Strings

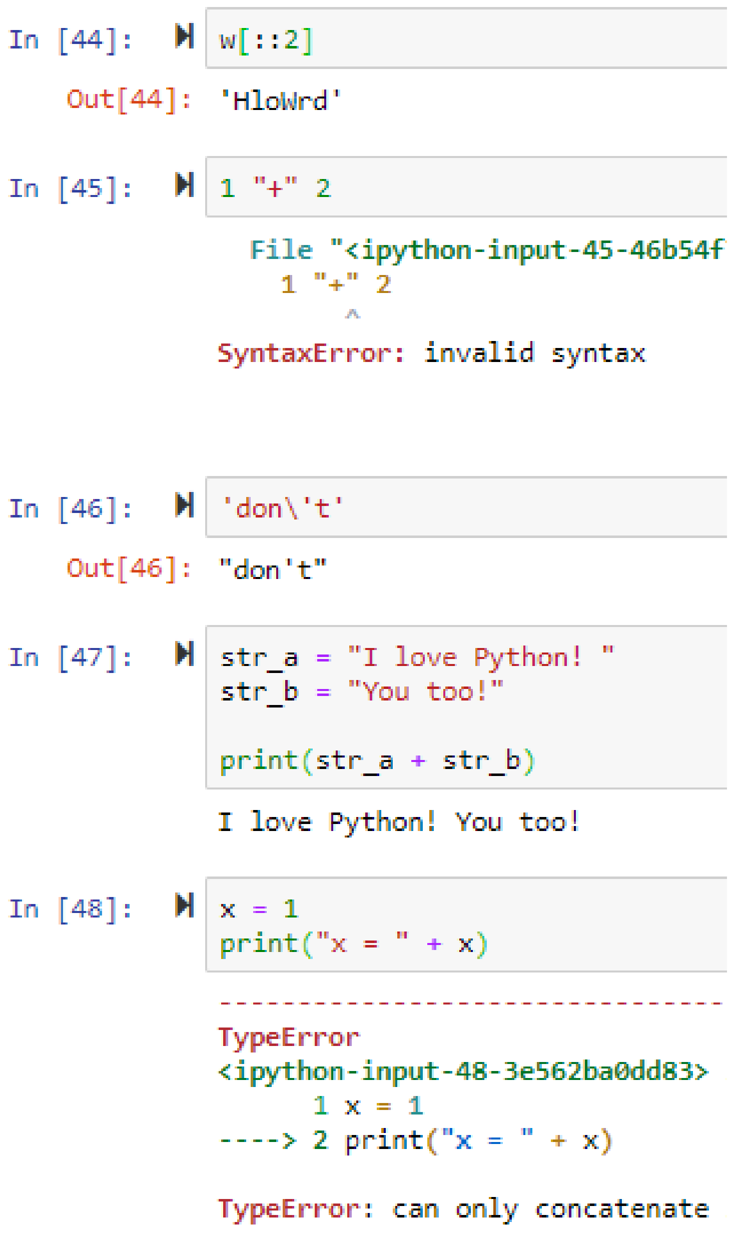
- A **string** is a sequence of characters, such as “Hello World” we saw before. Strings are surrounded by either single or double quotation marks. We could use **print** function to output the strings to the screen.
- A string is an array of characters, therefore it has length to indicate the size of the string. For example, we could check the size of the string by using the built-in function **len**.
- Strings also have **indexes** to indicate the location of each character, so that we could easily find out some character.
- We could get access to any character by using a **bracket** and the **index** of the position.
- We could also select a sequence as well using string **slicing**.
- You can also use **negative index** when slice the strings, which means counting from the end of the string.

Strings

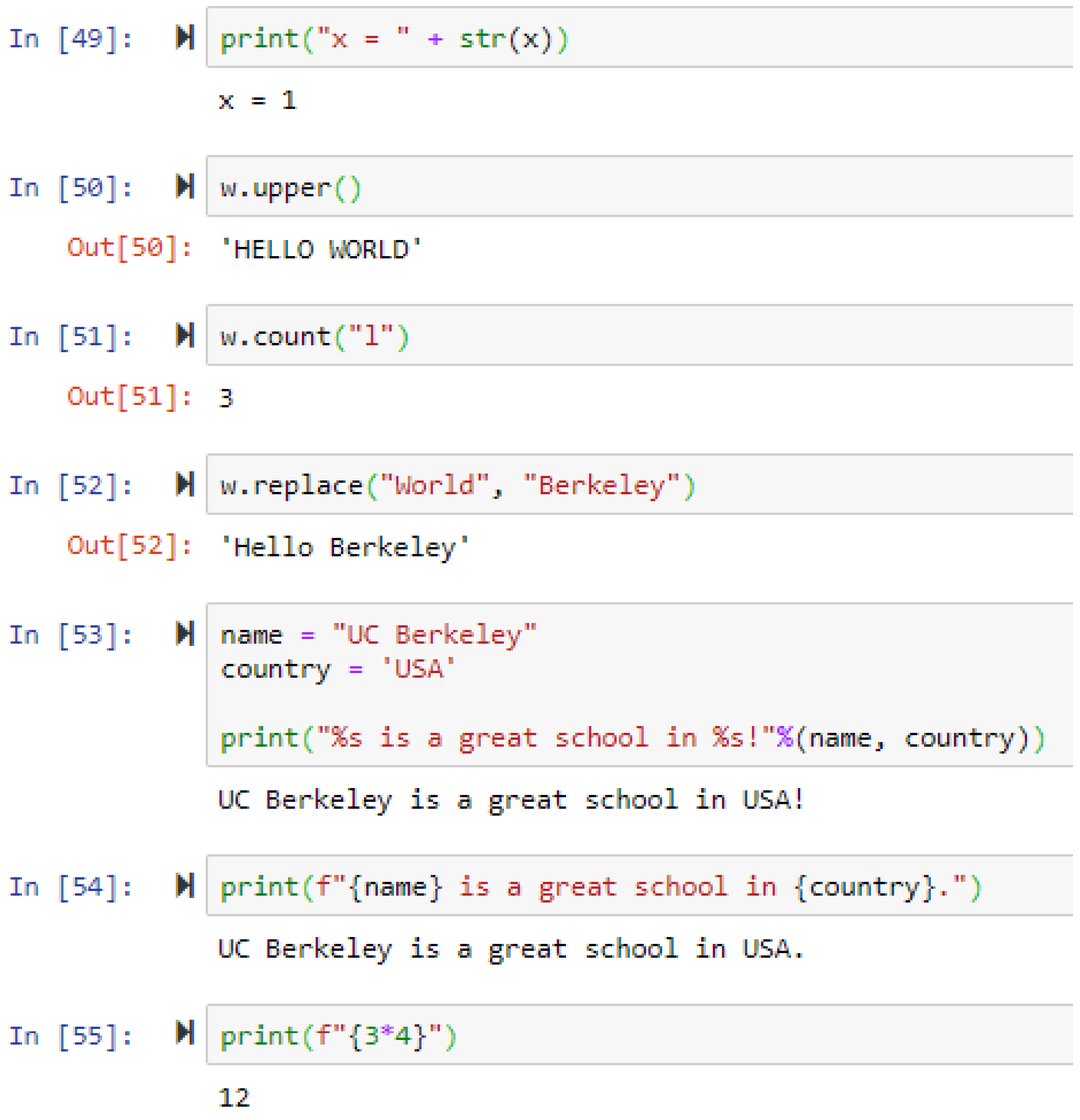
- Strings can **not be used** in the Mathematical operations.
- The backslash (\) is a way to tell Python this is part of the string, not to denote strings. The backslash character is used to **escape** characters that otherwise have a special meaning, such as **newline**, **backslash** itself, or the **quote** character.
- One string could be **concatenated** to another string, by using '**+**'.
- We could convert other data types to strings as well using the built-in function **str**.
- In Python, string as an object that has various methods that could be used to manipulate it (we will talk more about object-oriented programming later). They way to get access to the various methods is to use this patter "**string.method_name**".
- There are different ways to **pre-format** a string (see the last two examples in the next slide).



```
In [36]:  print("I love Python!")

          I love Python!

In [37]:  w = "Hello World"
          type(w)

Out[37]:  str

In [38]:  len(w)

Out[38]:  11

In [39]:  w[6]

Out[39]:  'W'

In [40]:  w[6:11]

Out[40]:  'World'

In [41]:  w[6:]

Out[41]:  'World'

In [42]:  w[:5]

Out[42]:  'Hello'

In [43]:  w[6:-2]

Out[43]:  'Wor'
```



```
In [44]:  w[::2]

Out[44]:  'HloWrd'

In [45]:  1 "+" 2

            File "<ipython-input-45-46b54f
              1 "+" 2
                  ^
          SyntaxError: invalid syntax

In [46]:  'don\'t'

Out[46]:  "don't"

In [47]:  str_a = "I love Python! "
          str_b = "You too!"

          print(str_a + str_b)

          I love Python! You too!

In [48]:  x = 1
          print("x = " + x)

          ---------------------------------
          TypeError
          <ipython-input-48-3e562ba0dd83>
                1 x = 1
          ----> 2 print("x = " + x)

          TypeError: can only concatenate
```

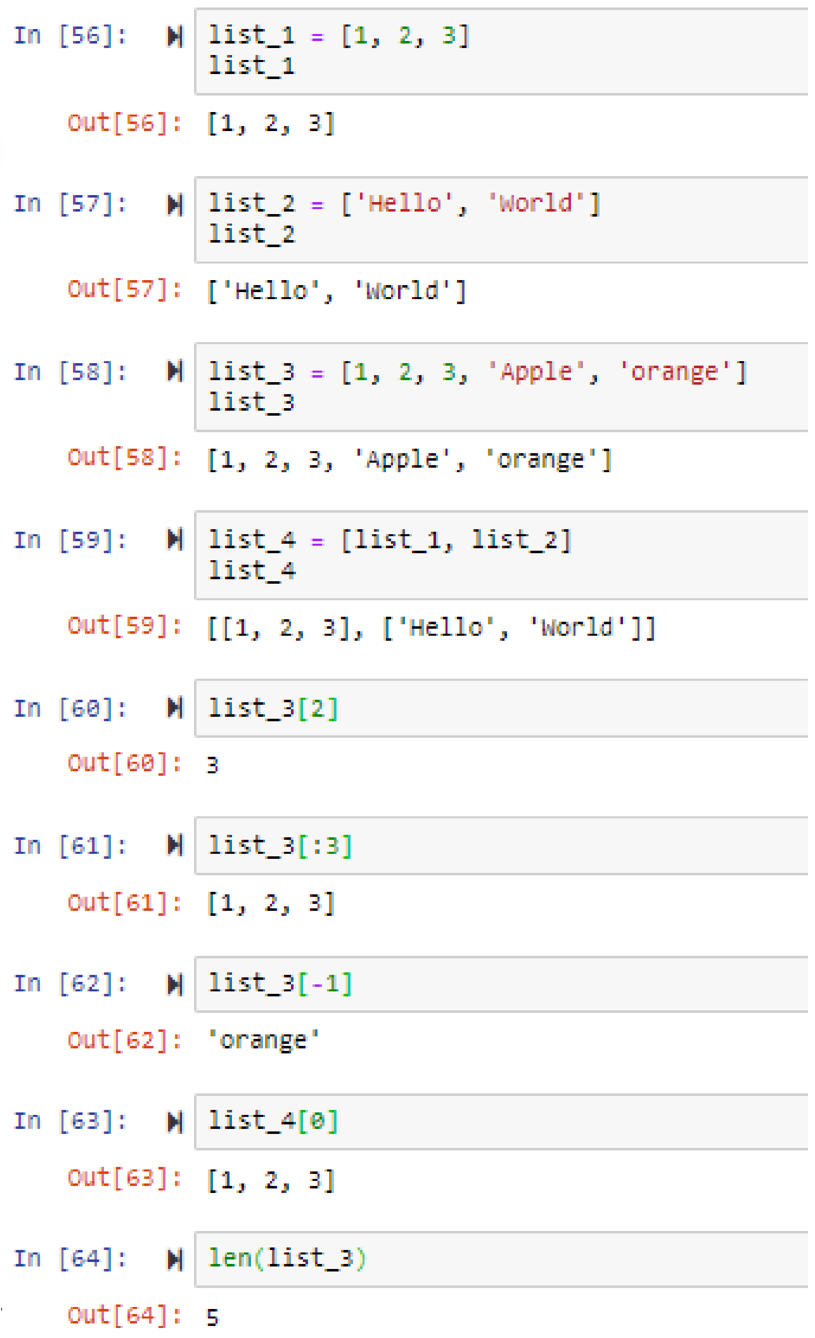


```
In [49]:  print("x = " + str(x))

          x = 1

In [50]:  w.upper()

Out[50]:  'HELLO WORLD'

In [51]:  w.count("l")

Out[51]:  3

In [52]:  w.replace("World", "Berkeley")

Out[52]:  'Hello Berkeley'

In [53]:  name = "UC Berkeley"
          country = 'USA'

          print("%s is a great school in %s!"%(name, country))

          UC Berkeley is a great school in USA!

In [54]:  print(f"{name} is a great school in {country}.")

          UC Berkeley is a great school in USA.

In [55]:  print(f"{3*4}")

          12
```

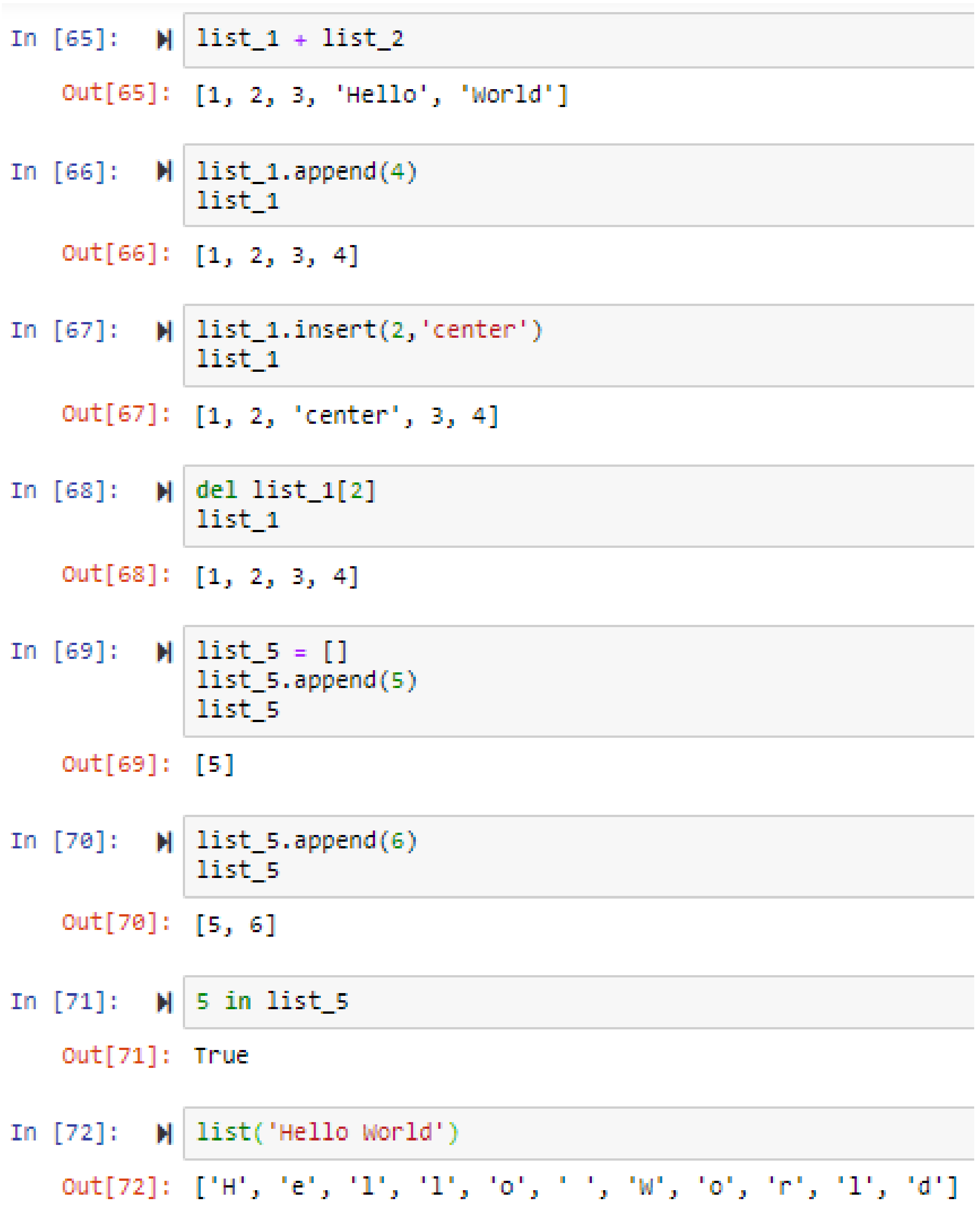
Lists

- The way to define a **List** is to use a pair of brackets **[ ]**, and the elements within it are separated by commas. A list could hold any type of data: numerical, or strings or other types.
- The way to retrieve the element in the list is very similar to the strings.
- New items could be added to an existing list by using the **append** method from the list.
- We could also insert or remove element from the list by using the methods **insert** and **remove**, but they are also operating on the list directly.
- We could also define an **empty** list and add in new element later using **append** method.
- We could also quickly check if an element is in the list using the operator **in**.
- Using the **list** function, we could turn other sequence items into a list.

UMN UNIVERSITAS MULTIMEDIA NUSANTARA



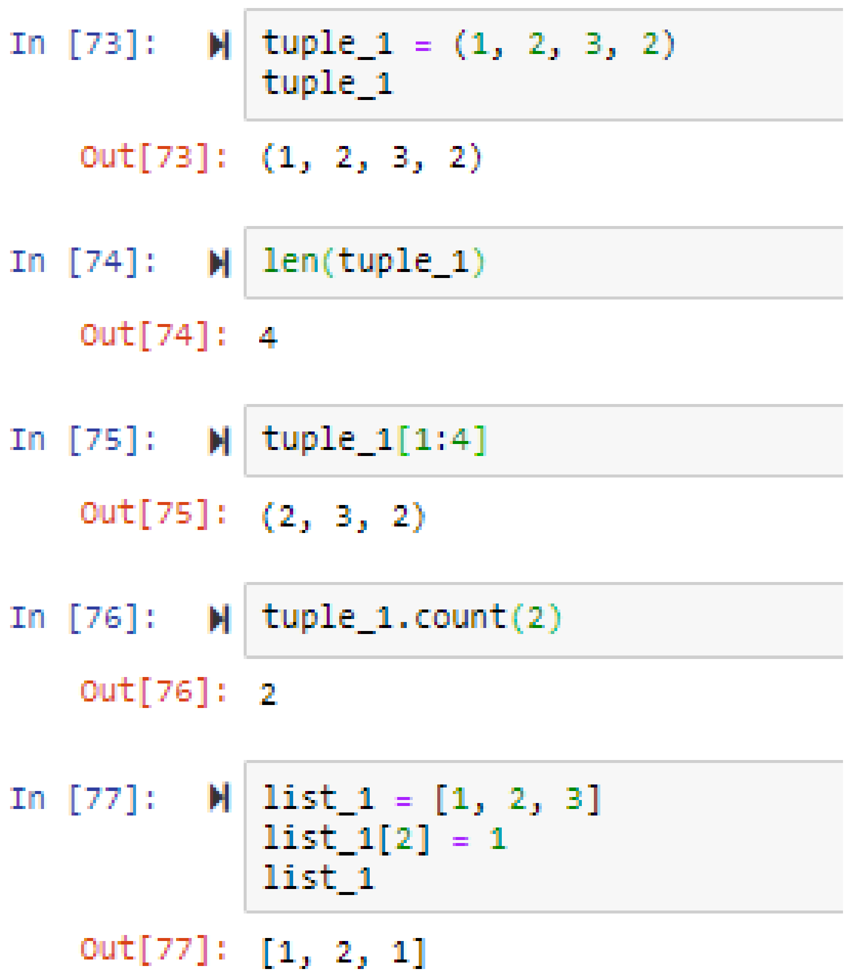
```
In [56]:  list_1 = [1, 2, 3]
          list_1
Out[56]:  [1, 2, 3]

In [57]:  list_2 = ['Hello', 'World']
          list_2
Out[57]:  ['Hello', 'World']

In [58]:  list_3 = [1, 2, 3, 'Apple', 'orange']
          list_3
Out[58]:  [1, 2, 3, 'Apple', 'orange']

In [59]:  list_4 = [list_1, list_2]
          list_4
Out[59]:  [[1, 2, 3], ['Hello', 'World']]

In [60]:  list_3[2]
Out[60]:  3

In [61]:  list_3[:3]
Out[61]:  [1, 2, 3]

In [62]:  list_3[-1]
Out[62]:  'orange'

In [63]:  list_4[0]
Out[63]:  [1, 2, 3]

In [64]:  len(list_3)
Out[64]:  5
```



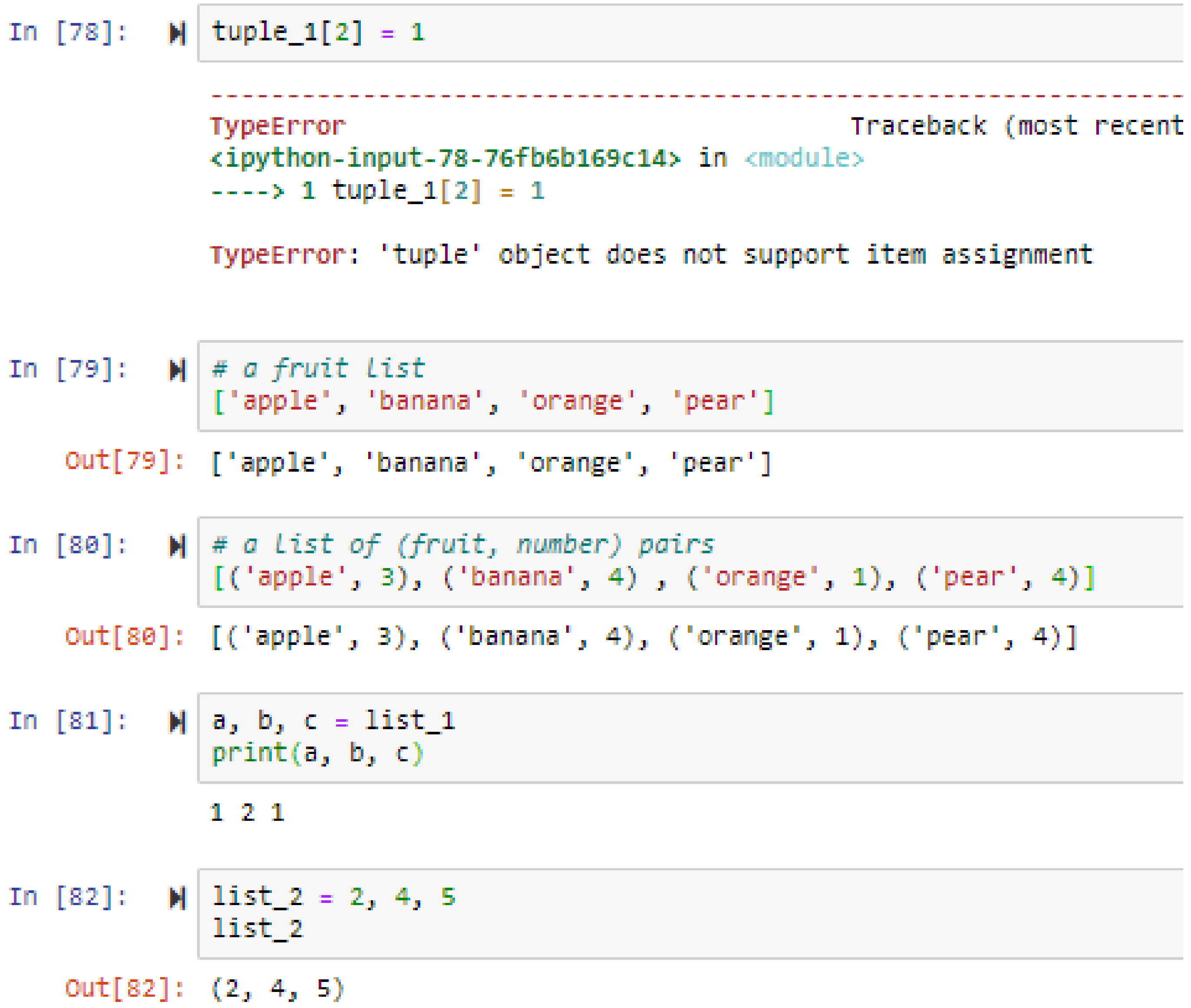
```
In [65]:  list_1 + list_2
Out[65]:  [1, 2, 3, 'Hello', 'World']

In [66]:  list_1.append(4)
          list_1
Out[66]:  [1, 2, 3, 4]

In [67]:  list_1.insert(2,'center')
          list_1
Out[67]:  [1, 2, 'center', 3, 4]

In [68]:  del list_1[2]
          list_1
Out[68]:  [1, 2, 3, 4]

In [69]:  list_5 = []
          list_5.append(5)
          list_5
Out[69]:  [5]

In [70]:  list_5.append(6)
          list_5
Out[70]:  [5, 6]

In [71]:  5 in list_5
Out[71]:  True

In [72]:  list('Hello World')
Out[72]:  ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd']
```

Tuples

- **Tuple** is usually defined by using a pair of parentheses **( )**, and its elements are separated by commas.
- As strings and lists, they way for indexing the tuples, slicing the elements, and even some methods are very similar.
- You may ask, what's the difference between lists and tuples?
- **Tuples** are **immutable**, and usually contain a **heterogeneous** sequence of elements that are accessed via **unpacking** (see later in this section) or **indexing** (or even by attribute in the case of named tuples).
- **Lists** are **mutable**, and their elements are usually **homogeneous** and are accessed by **iterating** over the list.



```
In [73]:  tuple_1 = (1, 2, 3, 2)
          tuple_1

Out[73]:  (1, 2, 3, 2)

In [74]:  len(tuple_1)

Out[74]:  4

In [75]:  tuple_1[1:4]

Out[75]:  (2, 3, 2)

In [76]:  tuple_1.count(2)

Out[76]:  2

In [77]:  list_1 = [1, 2, 3]
          list_1[2] = 1
          list_1

Out[77]:  [1, 2, 1]
```

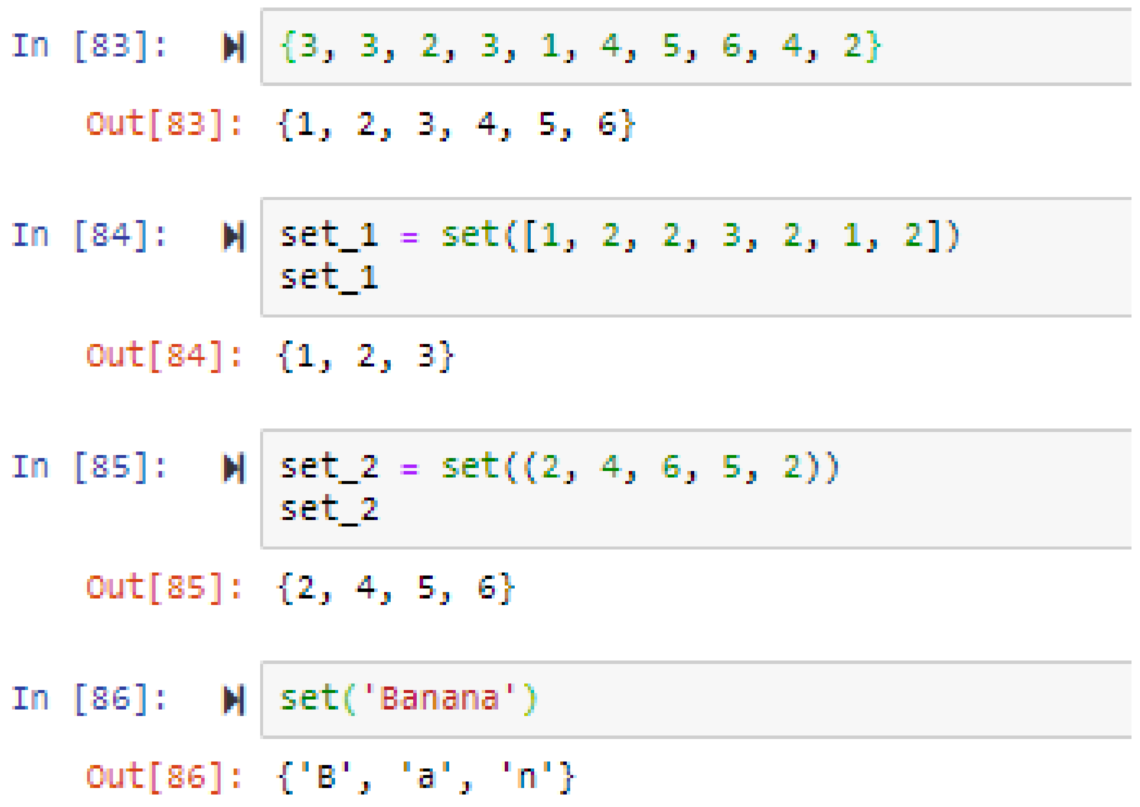


```
In [78]:  tuple_1[2] = 1

          ---------------------------------------------------------------------------
          TypeError                                 Traceback (most recent
          <ipython-input-78-76fb6b169c14> in <module>
          ----> 1 tuple_1[2] = 1

          TypeError: 'tuple' object does not support item assignment

In [79]:  # a fruit list
          ['apple', 'banana', 'orange', 'pear']

Out[79]:  ['apple', 'banana', 'orange', 'pear']

In [80]:  # a list of (fruit, number) pairs
          [('apple', 3), ('banana', 4) , ('orange', 1), ('pear', 4)]

Out[80]:  [('apple', 3), ('banana', 4), ('orange', 1), ('pear', 4)]

In [81]:  a, b, c = list_1
          print(a, b, c)

          1 2 1

In [82]:  list_2 = 2, 4, 5
          list_2

Out[82]:  (2, 4, 5)
```

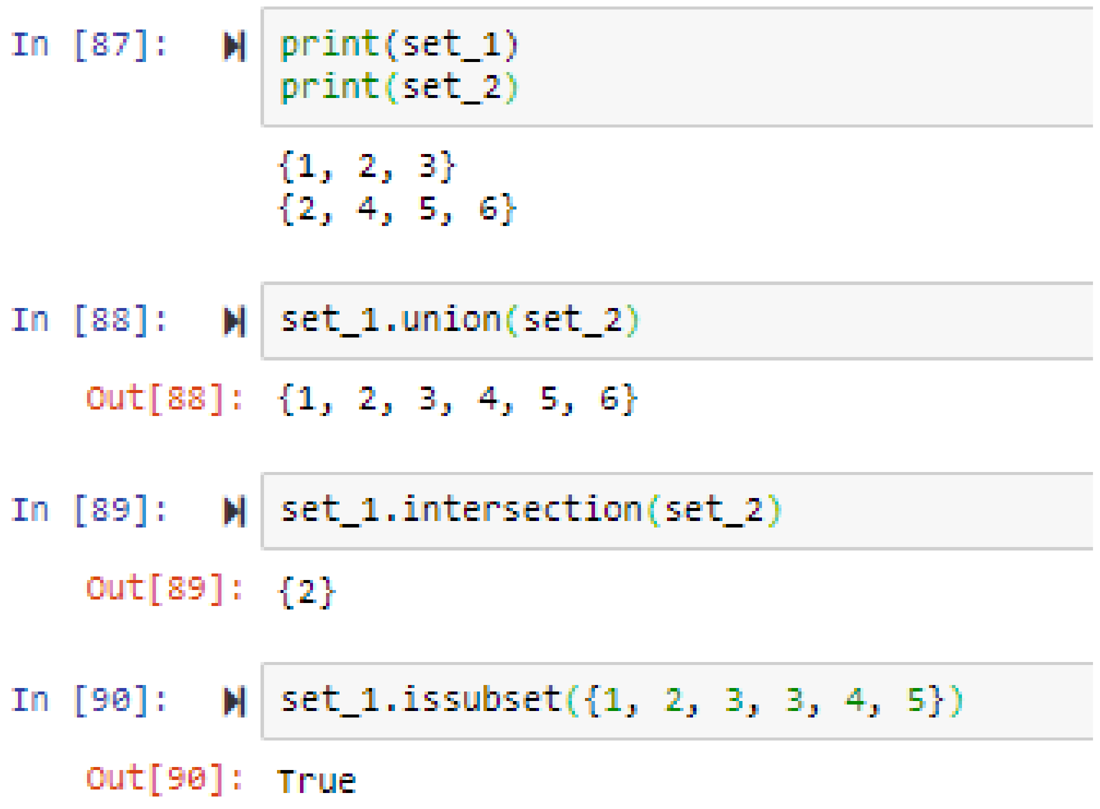
Sets

- Another data type in Python is **sets**.
- It is a type that could store an **unordered collection** with **no duplicate** elements.
- It is also support the mathematical operations like **union**, **intersection**, **difference**, and **symmetric difference**.
- It is defined by using a pair of braces **{ }**, and its elements are separated by commas.
- One quick usage of this is to find out the **unique elements** in a string, list, or tuple.

Sets



```
In [83]:  {3, 3, 2, 3, 1, 4, 5, 6, 4, 2}

Out[83]: {1, 2, 3, 4, 5, 6}

In [84]:  set_1 = set([1, 2, 2, 3, 2, 1, 2])
          set_1

Out[84]: {1, 2, 3}

In [85]:  set_2 = set((2, 4, 6, 5, 2))
          set_2

Out[85]: {2, 4, 5, 6}

In [86]:  set('Banana')

Out[86]: {'B', 'a', 'n'}
```

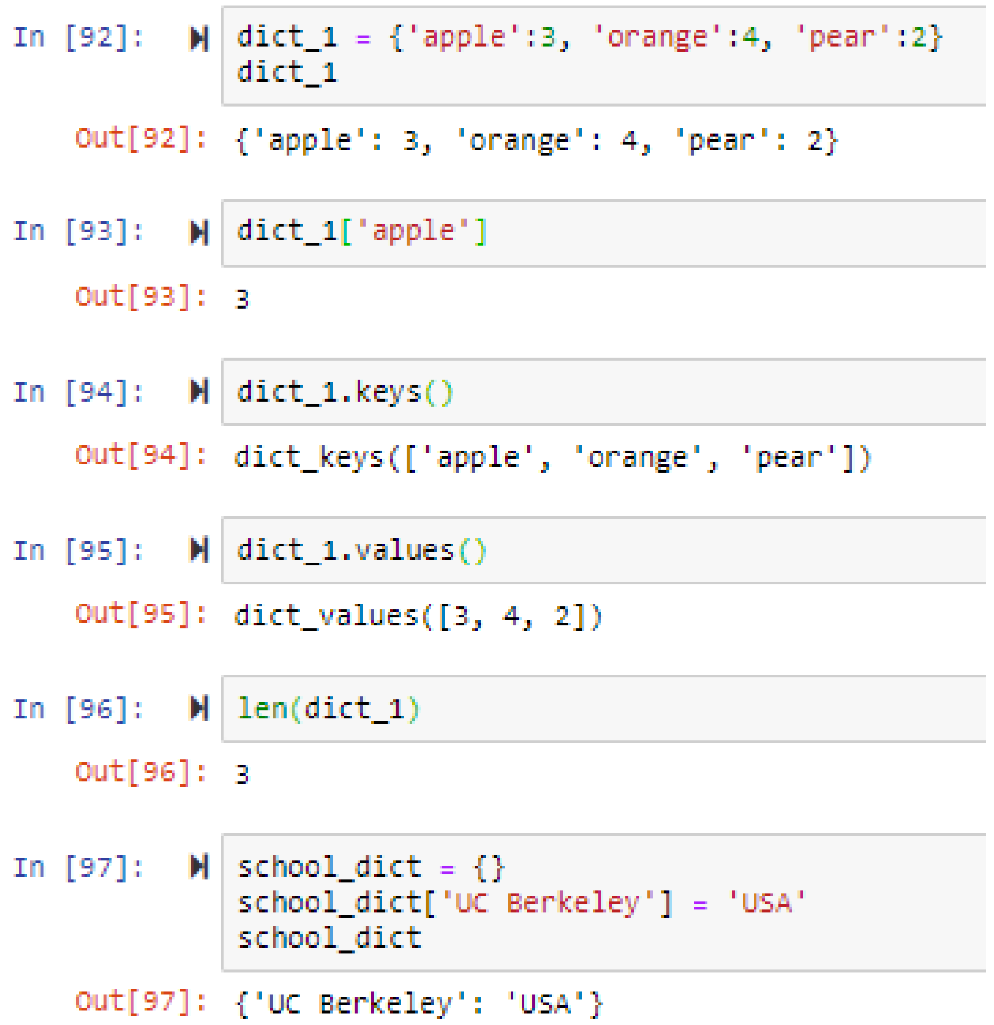


```
In [87]:  print(set_1)
          print(set_2)

          {1, 2, 3}
          {2, 4, 5, 6}

In [88]:  set_1.union(set_2)

Out[88]: {1, 2, 3, 4, 5, 6}

In [89]:  set_1.intersection(set_2)

Out[89]: {2}

In [90]:  set_1.issubset({1, 2, 3, 3, 4, 5})

Out[90]: True
```

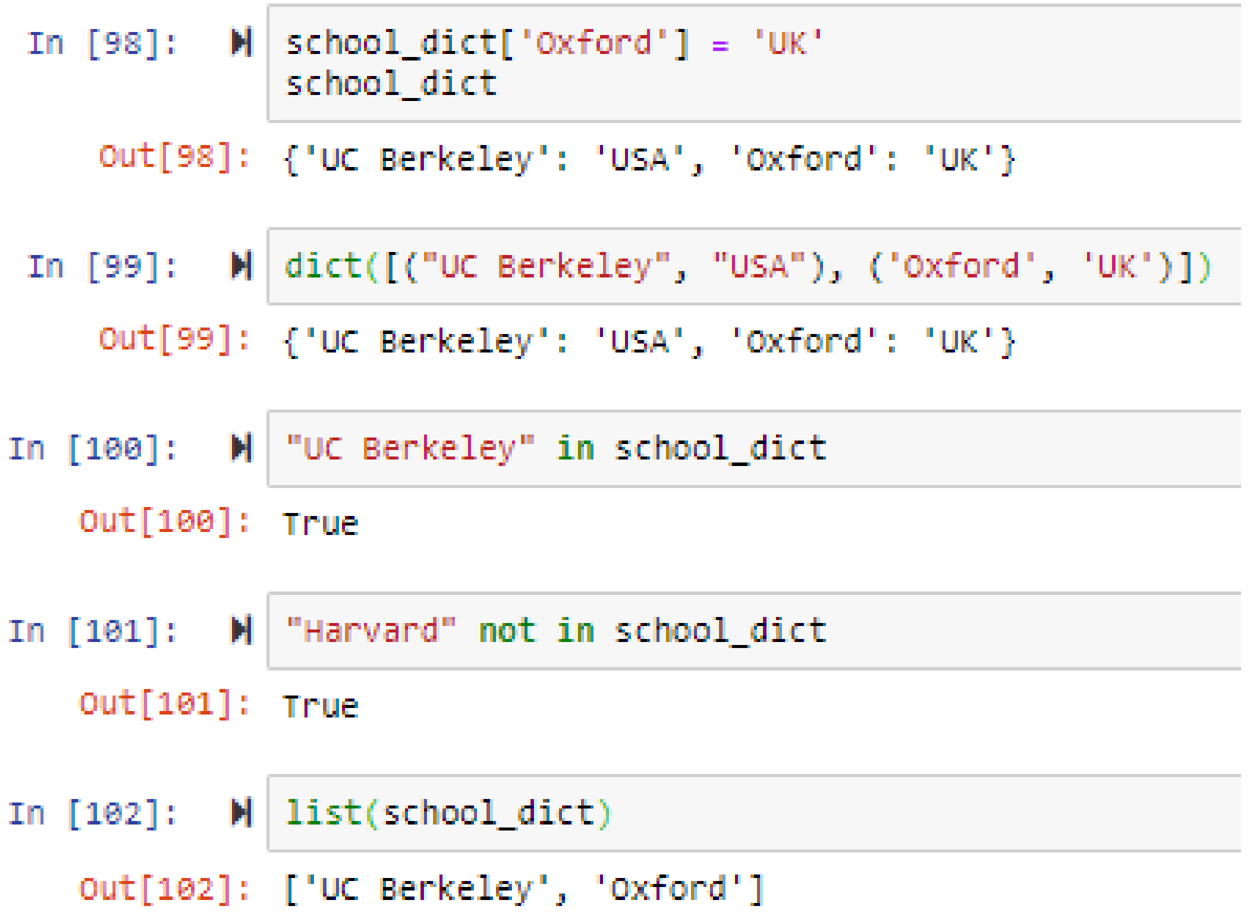
Dictionaries

- A **dictionary** is a key-value pairs, and each key maps to a corresponding value. It is defined by using a pair of braces **{ }**, while the elements are a list of comma separated **key:value** pairs (note the key:value pair is separated by the colon, with key at front and value at the end).
- Within a dictionary, elements are stored **without order**, therefore, you can not access a dictionary based on a sequence of index numbers. To get access to a dictionary, we need to use the key of the element - **dictionary[key]**.
- We could get all the keys in a dictionary by using the **keys** method, or all the values by using the method **values**.
- We could define an **empty** dictionary and then fill in the element later. Or we could turn a list of tuples with **(key, value)** pairs to a dictionary.
- We could also check if an element belong to a dictionary using the operator **in**.

Dictionaries



```
In [92]:  dict_1 = {'apple':3, 'orange':4, 'pear':2}
          dict_1

Out[92]: {'apple': 3, 'orange': 4, 'pear': 2}

In [93]:  dict_1['apple']

Out[93]: 3

In [94]:  dict_1.keys()

Out[94]: dict_keys(['apple', 'orange', 'pear'])

In [95]:  dict_1.values()

Out[95]: dict_values([3, 4, 2])

In [96]:  len(dict_1)

Out[96]: 3

In [97]:  school_dict = {}
          school_dict['UC Berkeley'] = 'USA'
          school_dict

Out[97]: {'UC Berkeley': 'USA'}
```



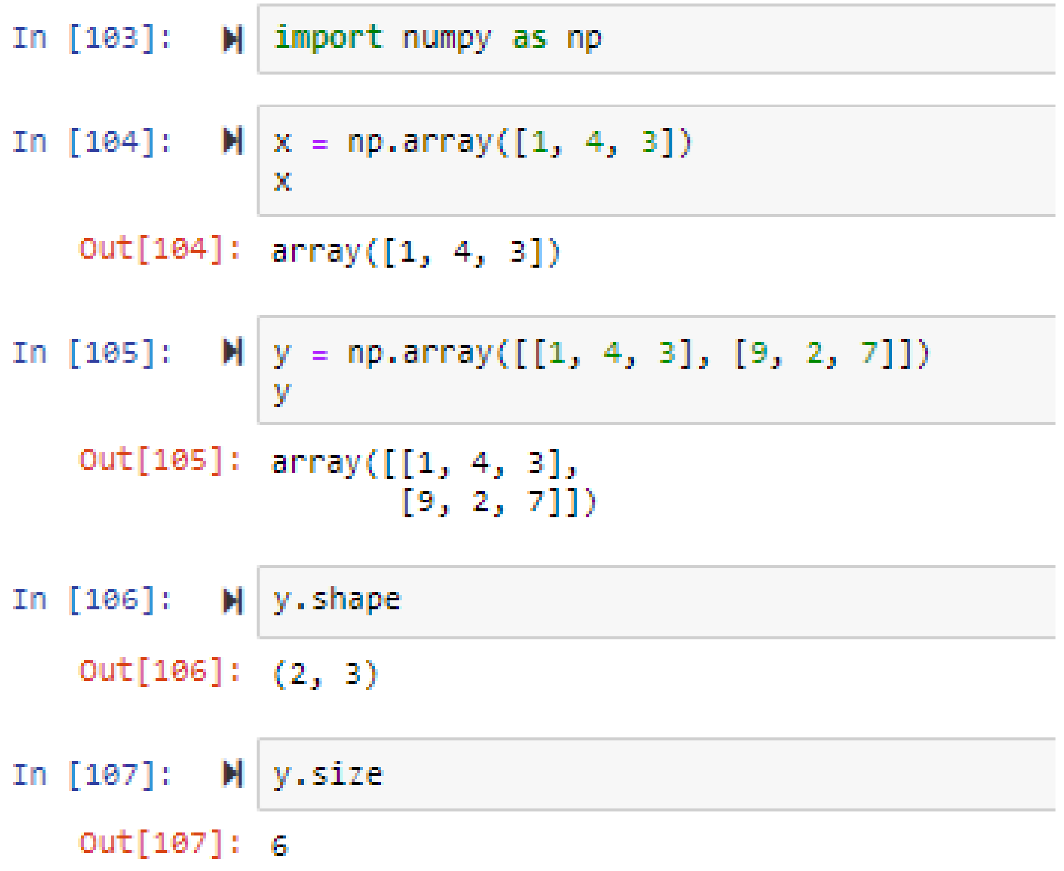
```
In [98]:  school_dict['Oxford'] = 'UK'
          school_dict

Out[98]: {'UC Berkeley': 'USA', 'Oxford': 'UK'}

In [99]:  dict([("UC Berkeley", "USA"), ('Oxford', 'UK')])

Out[99]: {'UC Berkeley': 'USA', 'Oxford': 'UK'}

In [100]:  "UC Berkeley" in school_dict

Out[100]: True

In [101]:  "Harvard" not in school_dict

Out[101]: True

In [102]:  list(school_dict)

Out[102]: ['UC Berkeley', 'Oxford']
```

Numpy Arrays

- Here we are going to introduce the most common way to handle arrays in Python using the **Numpy** module. Numpy is probably the most fundamental numerical computing module in Python.
- NumPy is important in scientific computing, it is coded both in Python and C (for speed). On its website, a few important features for Numpy is listed:
    - a powerful N-dimensional array object
    - sophisticated (broadcasting) functions
    - tools for integrating C/C++ and Fortran code
    - useful linear algebra, Fourier transform, and random number capabilities
- In order to use Numpy module, we need to **import** it first. A conventional way to import it is to use “**np**” as a shortened name.

Numpy Arrays

- To define an array in Python, you could use the **np.array** function to convert a list.

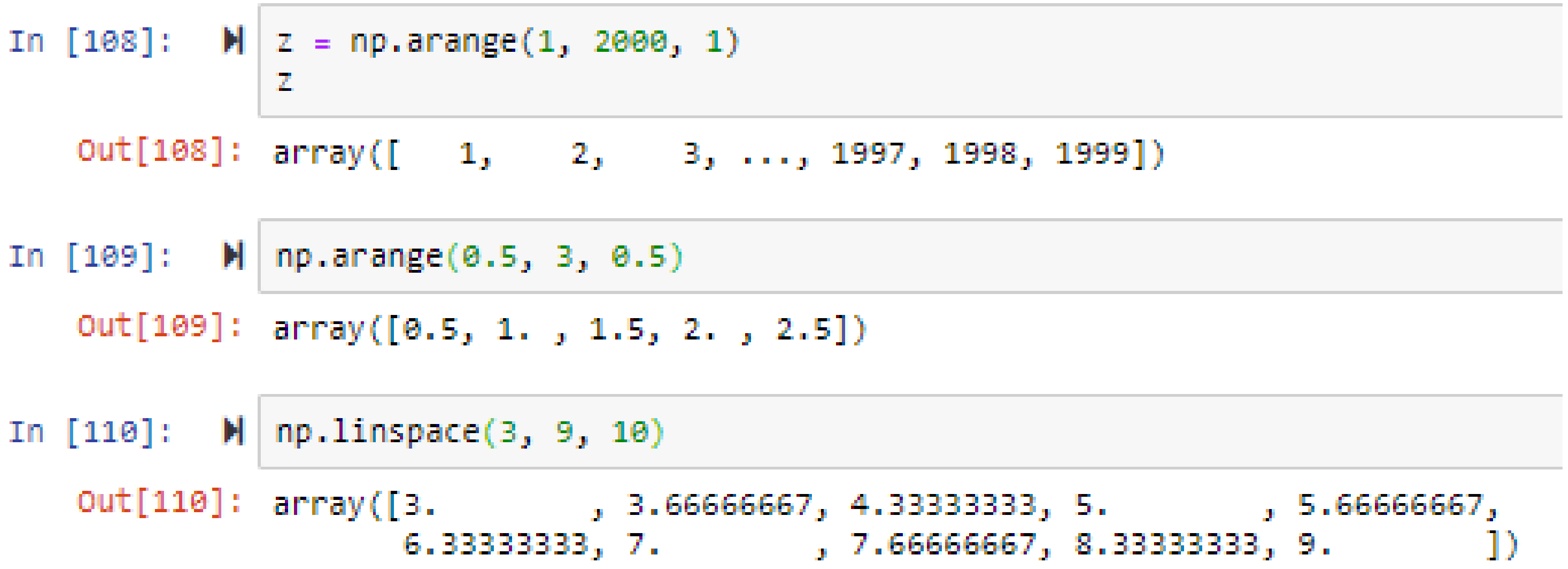


```
In [103]:  import numpy as np

In [104]:  x = np.array([1, 4, 3])
           x

Out[104]:  array([1, 4, 3])

In [105]:  y = np.array([[1, 4, 3], [9, 2, 7]])
           y

Out[105]:  array([[1, 4, 3],
                  [9, 2, 7]])

In [106]:  y.shape

Out[106]:  (2, 3)

In [107]:  y.size

Out[107]:  6
```

- Many times we would like to know the size or length of an array.
- The array **shape** attribute is called on an array M and returns a 2 × 3 array where the first element is the number of rows in the matrix M and the second element is the number of columns in M. Note that the output of the shape attribute is a **tuple**.
- The **size** attribute is called on an array M and returns the total number of elements in matrix M.

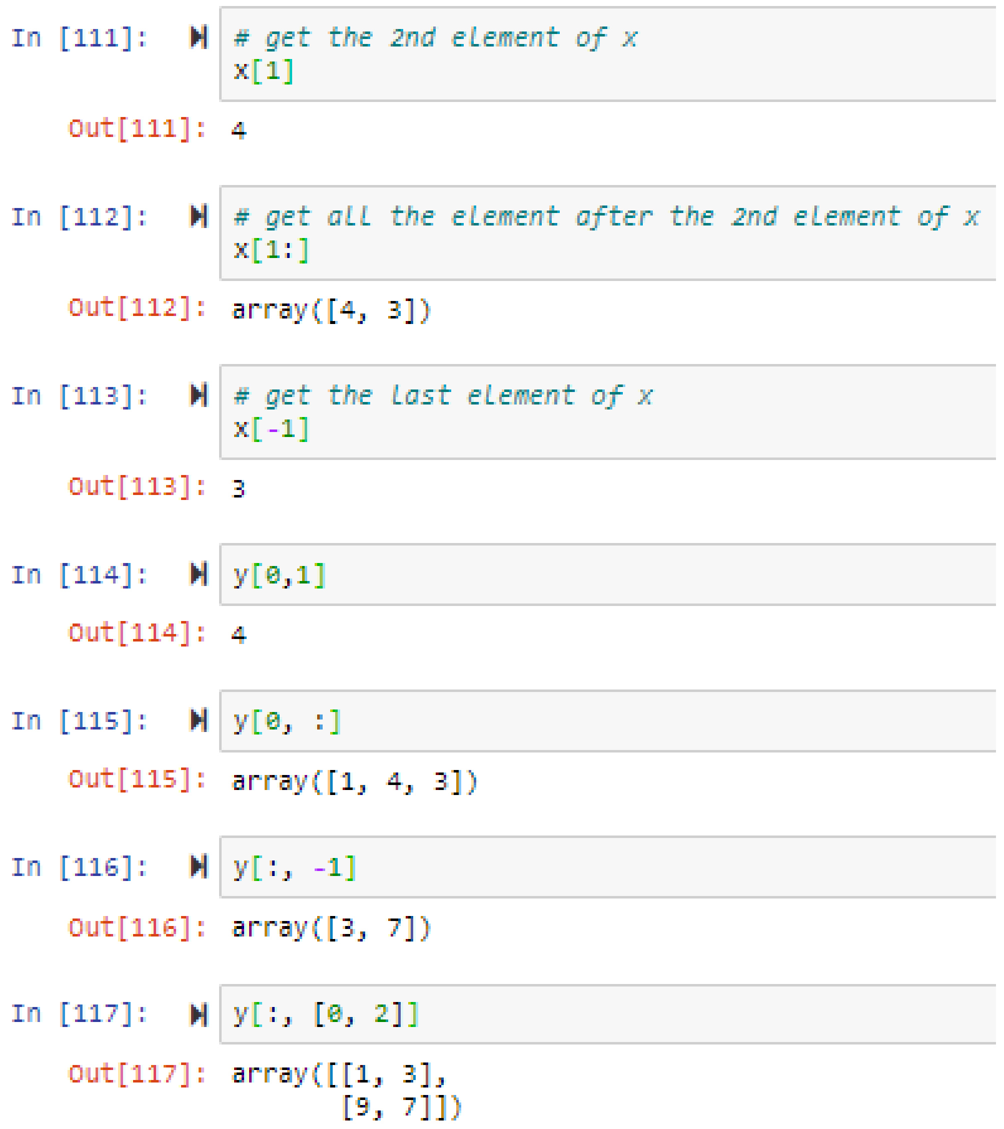
Numpy Arrays

- For generating arrays that are in order and evenly spaced, it is useful to use the **arange** function in Numpy.
- Sometimes we want to guarantee a start and end point for an array but still have evenly spaced elements. For this purpose you can use the function **np.linspace**. linspace takes three input values separated by commas.



```
In [108]:   z = np.arange(1, 2000, 1)
            z

Out[108]: array([   1,    2,    3, ..., 1997, 1998, 1999])

In [109]:   np.arange(0.5, 3, 0.5)

Out[109]: array([0.5, 1. , 1.5, 2. , 2.5])

In [110]:   np.linspace(3, 9, 10)

Out[110]: array([3.        , 3.66666667, 4.33333333, 5.        , 5.66666667,
                 6.33333333, 7.        , 7.66666667, 8.33333333, 9.        ])
```

Numpy Arrays

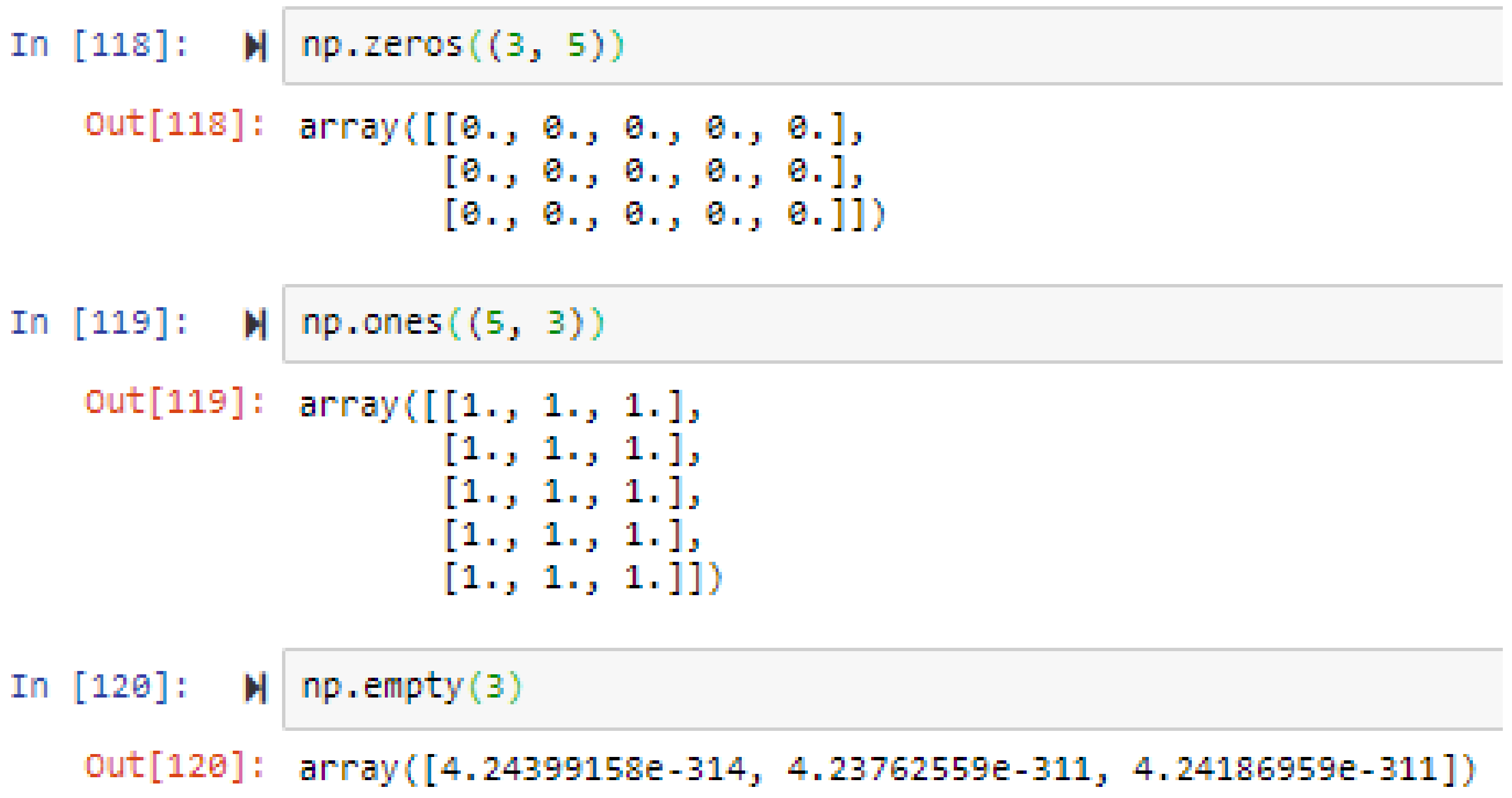


```
In [111]:  # get the 2nd element of x
           x[1]

Out[111]:  4

In [112]:  # get all the element after the 2nd element of x
           x[1:]

Out[112]:  array([4, 3])

In [113]:  # get the last element of x
           x[-1]

Out[113]:  3

In [114]:  y[0,1]

Out[114]:  4

In [115]:  y[0, :]

Out[115]:  array([1, 4, 3])

In [116]:  y[:, -1]

Out[116]:  array([3, 7])

In [117]:  y[:, [0, 2]]

Out[117]:  array([[1, 3],
                  [9, 7]])
```

- Getting access to the 1D numpy array is similar to what we described for lists or tuples, it has an index to indicate the location.
- For 2D arrays, it is slightly different, since we have rows and columns.
- To get access to the data in a 2D array M, we need to use **M[r, c]**, that the row **r** and column **c** are separated by comma. This is referred to as array indexing. The **r** and **c** could be single number, a list and so on.

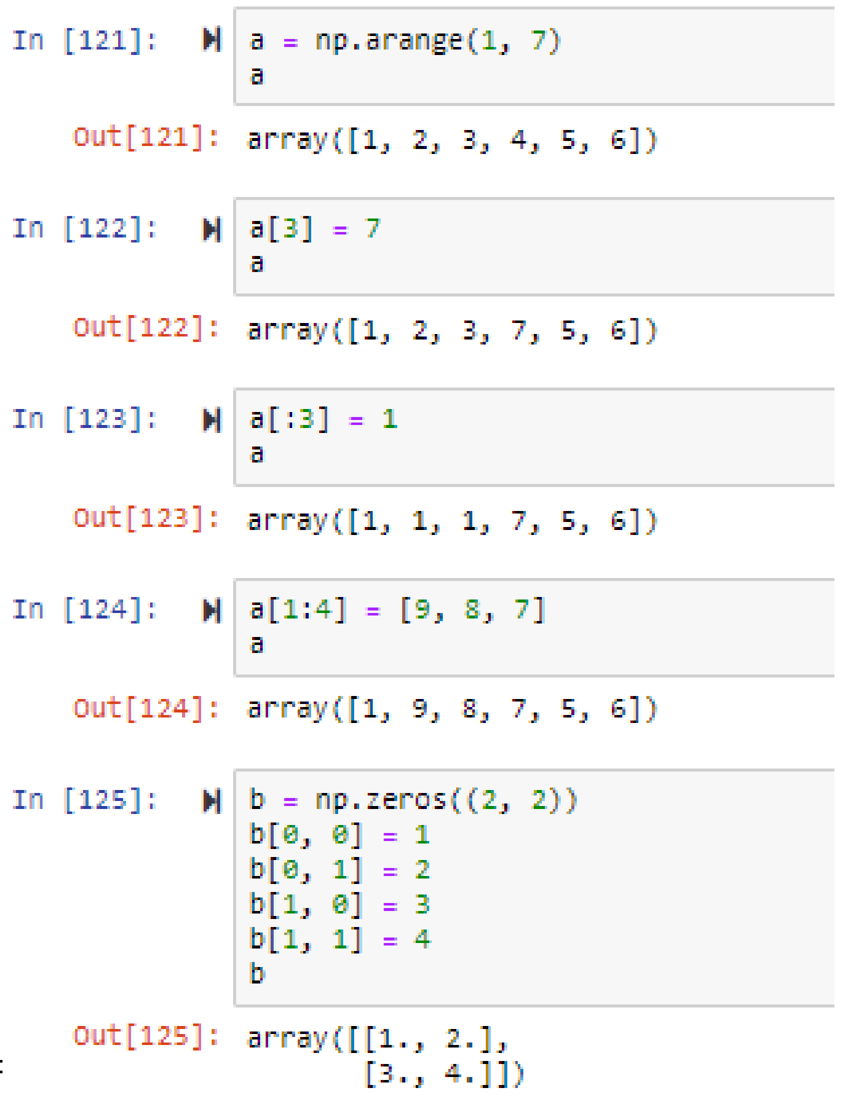
Numpy Arrays

- There are some predefined arrays that are really useful.
- For example, the **np.zeros**, **np.ones**, and **np.empty** are 3 useful functions.



```
In [118]:  np.zeros((3, 5))

Out[118]: array([[0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0.],
                 [0., 0., 0., 0., 0.]])

In [119]:  np.ones((5, 3))

Out[119]: array([[1., 1., 1.],
                 [1., 1., 1.],
                 [1., 1., 1.],
                 [1., 1., 1.],
                 [1., 1., 1.]])

In [120]:  np.empty(3)

Out[120]: array([4.24399158e-314, 4.23762559e-311, 4.24186959e-311])
```

Numpy Arrays



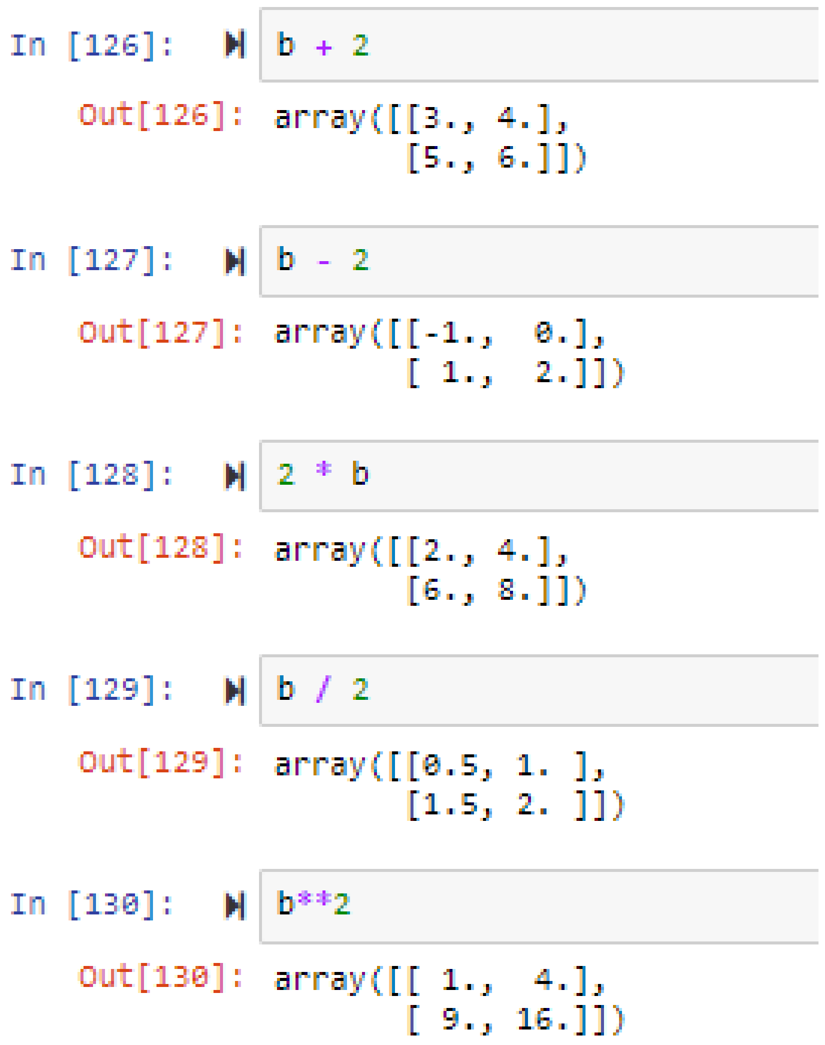
```
In [121]:  a = np.arange(1, 7)
           a

Out[121]: array([1, 2, 3, 4, 5, 6])

In [122]:  a[3] = 7
           a

Out[122]: array([1, 2, 3, 7, 5, 6])

In [123]:  a[:3] = 1
           a

Out[123]: array([1, 1, 1, 7, 5, 6])

In [124]:  a[1:4] = [9, 8, 7]
           a

Out[124]: array([1, 9, 8, 7, 5, 6])

In [125]:  b = np.zeros((2, 2))
           b[0, 0] = 1
           b[0, 1] = 2
           b[1, 0] = 3
           b[1, 1] = 4
           b

Out[125]: array([[1., 2.],
                 [3., 4.]])
```

- You can reassign a value of an array by using array indexing and the assignment operator. You can reassign multiple elements to a single number using array indexing on the left side. You can also reassign multiple elements of an array as long as both the number of elements being assigned and the number of elements assigned is the same. You can create an array using array indexing.
- Although you can create an array from scratch using indexing, we do not advise it. It can confuse you and errors will be harder to find in your code later.

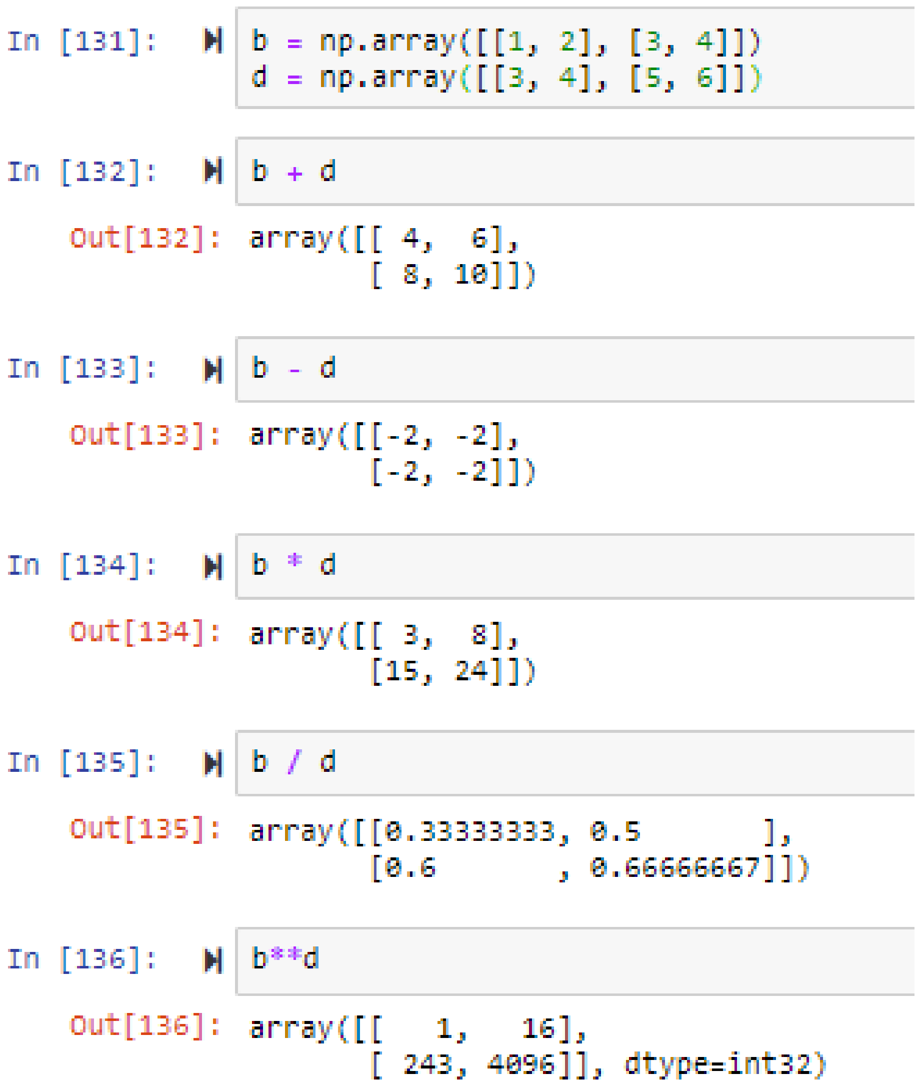
Numpy Arrays

- Basic arithmetic is defined for arrays. However, there are operations between a scalar (a single number) and an array and operations between two arrays. We will start with operations between a scalar and an array.
- To illustrate, let c be a scalar, and b be a matrix. $b + c$, $b - c$, $b * c$ and b / c adds a to every element of b, subtracts c from every element of b, multiplies every element of b by c, and divides every element of b by c, respectively.
- Describing operations between two matrices is more complicated. Let b and d be two matrices of the same size. $b - d$ takes every element of b and subtracts the corresponding element of d. Similarly, $b + d$ adds every element of d to the corresponding element of b.
- There are two different kinds of matrix multiplication (and division). There is **element-by-element** matrix multiplication and **standard** matrix multiplication. For this section, we will only show how element-by-element matrix multiplication and division work.

Numpy Arrays



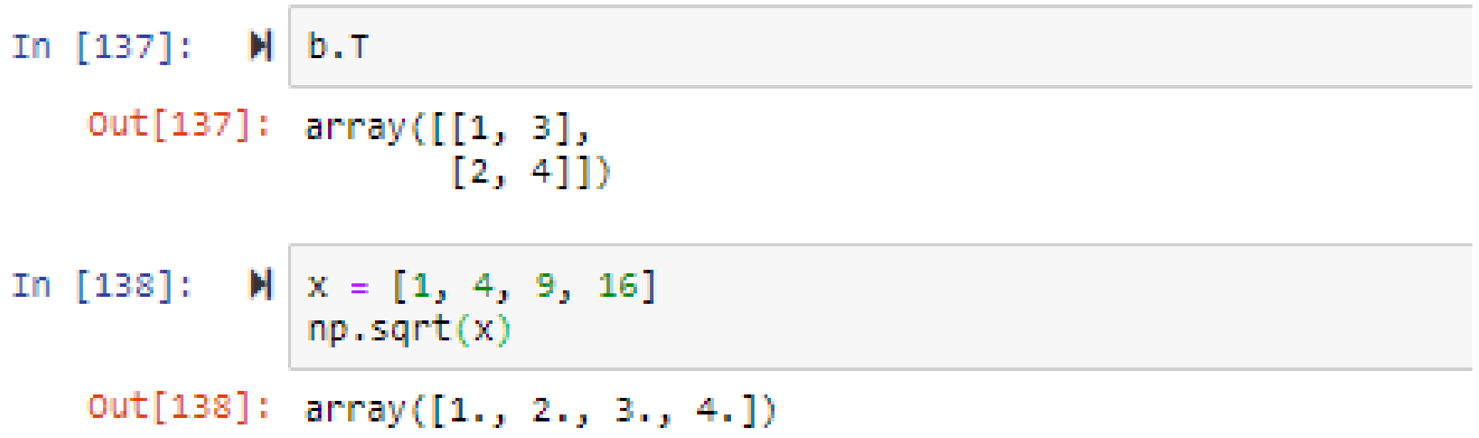
```
In [126]:  b + 2

Out[126]: array([[3., 4.],
                 [5., 6.]])

In [127]:  b - 2

Out[127]: array([[-1.,  0.],
                 [ 1.,  2.]])

In [128]:  2 * b

Out[128]: array([[2., 4.],
                 [6., 8.]])

In [129]:  b / 2

Out[129]: array([[0.5, 1. ],
                 [1.5, 2. ]])

In [130]:  b**2

Out[130]: array([[ 1.,  4.],
                 [ 9., 16.]])
```



```
In [131]:  b = np.array([[1, 2], [3, 4]])
           d = np.array([[3, 4], [5, 6]])

In [132]:  b + d

Out[132]: array([[ 4,  6],
                 [ 8, 10]])

In [133]:  b - d

Out[133]: array([[-2, -2],
                 [-2, -2]])

In [134]:  b * d

Out[134]: array([[ 3,  8],
                 [15, 24]])

In [135]:  b / d

Out[135]: array([[0.33333333, 0.5       ],
                 [0.6       , 0.66666667]])

In [136]:  b**d

Out[136]: array([[   1,   16],
                 [ 243, 4096]], dtype=int32)
```

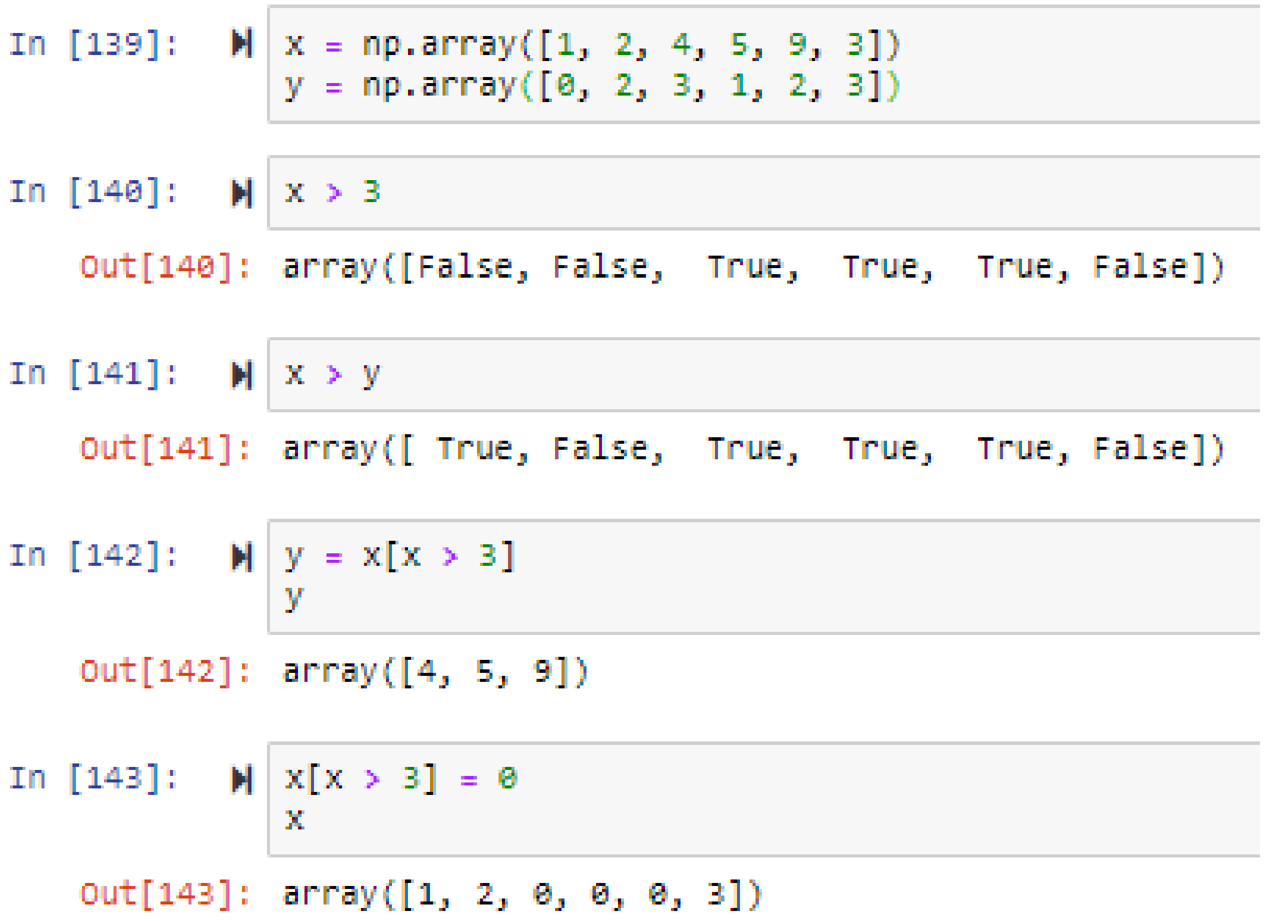
Numpy Arrays

- The transpose of an array, b, is an array, d, where $b[i,j] = d[j,i]$.
- In other words, the transpose switches the rows and the columns of b. You can transpose an array in Python using the array method T.
- Numpy has many arithmetic functions, such as sin, cos, etc., can take arrays as input arguments. The output is the function evaluated for every element of the input array.
- A function that takes an array as input and performs the function on it is said to be **vectorized**.



```
In [137]:  b.T

Out[137]: array([[1, 3],
                 [2, 4]])

In [138]:  x = [1, 4, 9, 16]
           np.sqrt(x)

Out[138]: array([1., 2., 3., 4.])
```

Numpy Arrays



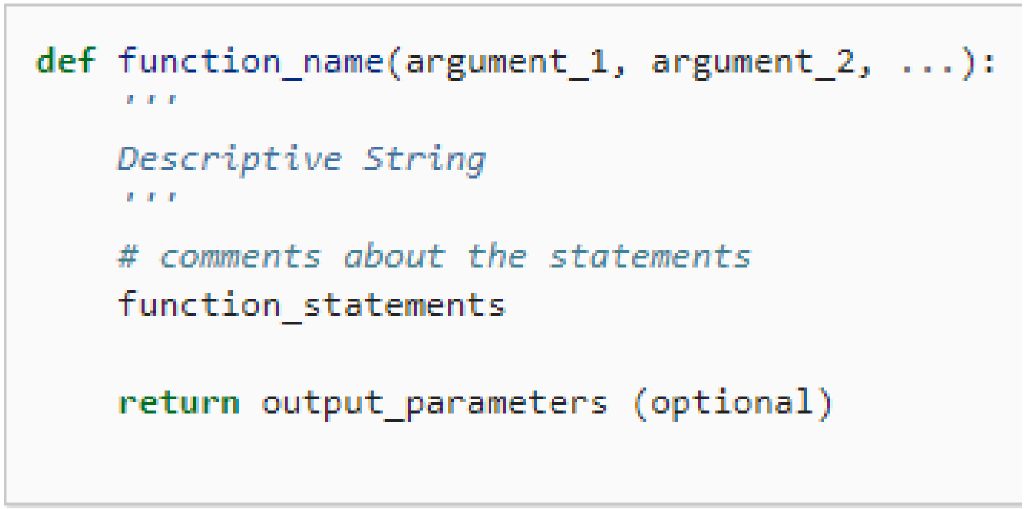
```
In [139]:  x = np.array([1, 2, 4, 5, 9, 3])
           y = np.array([0, 2, 3, 1, 2, 3])

In [140]:  x > 3

Out[140]: array([False, False,  True,  True,  True, False])

In [141]:  x > y

Out[141]: array([ True, False,  True,  True,  True, False])

In [142]:  y = x[x > 3]
           y

Out[142]: array([4, 5, 9])

In [143]:  x[x > 3] = 0
           x

Out[143]: array([1, 2, 0, 0, 0, 3])
```

- Logical operations are only defined between a scalar and an array and between two arrays of the same size. Between a scalar and an array, the logical operation is conducted between the scalar and each element of the array. Between two arrays, the logical operation is conducted **element-by-element**.
- Python can index elements of an array that satisfy a logical expression.

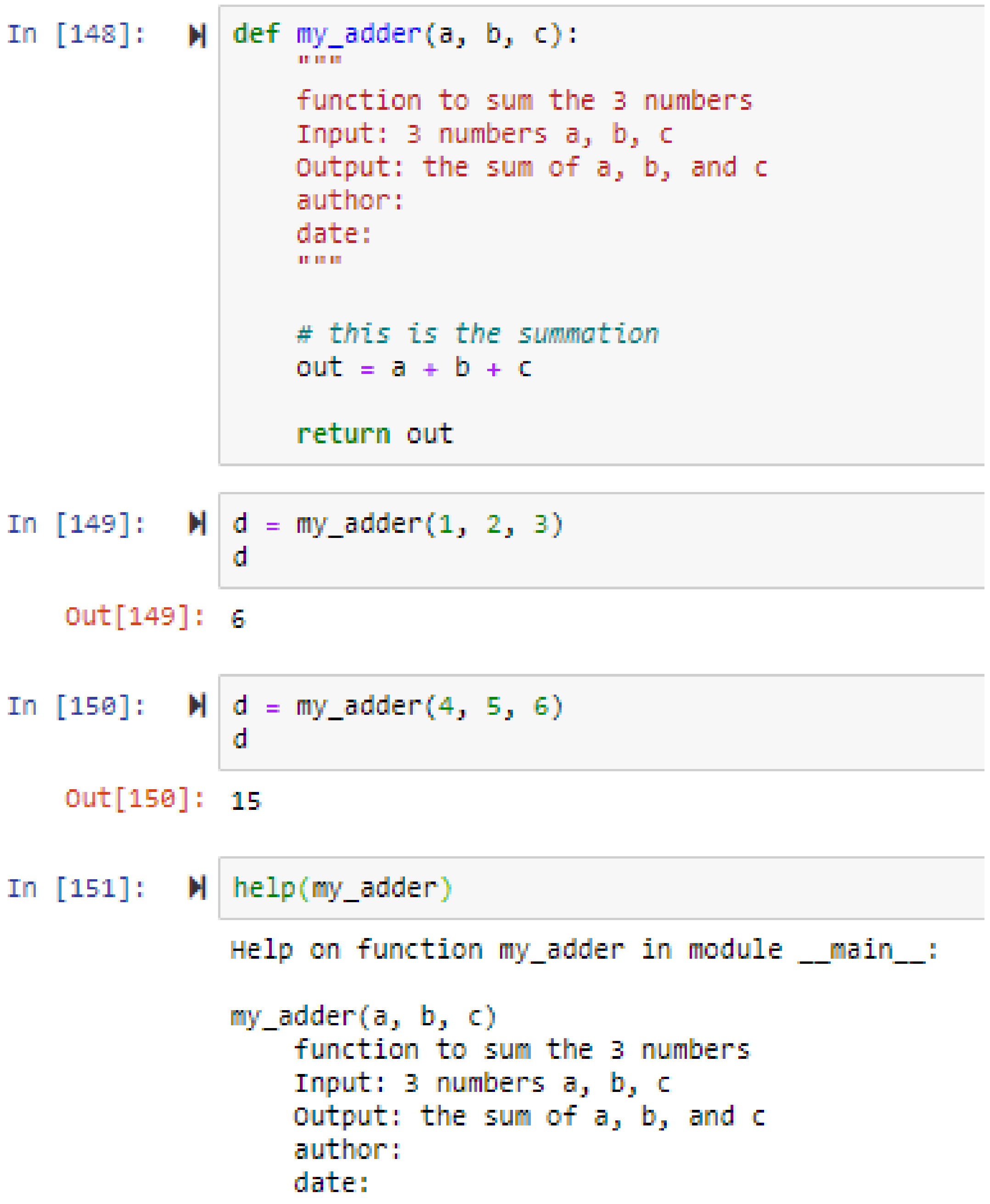
Functions

- In programming, a **function** is a sequence of instructions that performs a specific task. A function is a block of code that can run when it is called. A function can have input arguments, which are made available to it by the user, the entity calling the function. Functions also have output parameters, which are the results of the function that the user expects to receive once the function has completed its task.
- We saw many built-in Python functions already, such as **type**, **len**, and so on. Also we saw the functions from some packages, for example, **math.sin**, **np.array** and so on.
- We can define our own functions. A function can be specified in several ways. Here we will introduce the most common way to define a function which can be specified using the keyword **def**.
- Functions must conform to a naming scheme similar to variables. They can only contain alphanumeric characters and underscores, and the first character must be a letter.

Functions



```
def function_name(argument_1, argument_2, ...):
    '''
    Descriptive String
    '''
    # comments about the statements
    function_statements

    return output_parameters (optional)
```



```
In [148]:  def my_adder(a, b, c):
               """
               function to sum the 3 numbers
               Input: 3 numbers a, b, c
               Output: the sum of a, b, and c
               author:
               date:
               """

               # this is the summation
               out = a + b + c

               return out
```

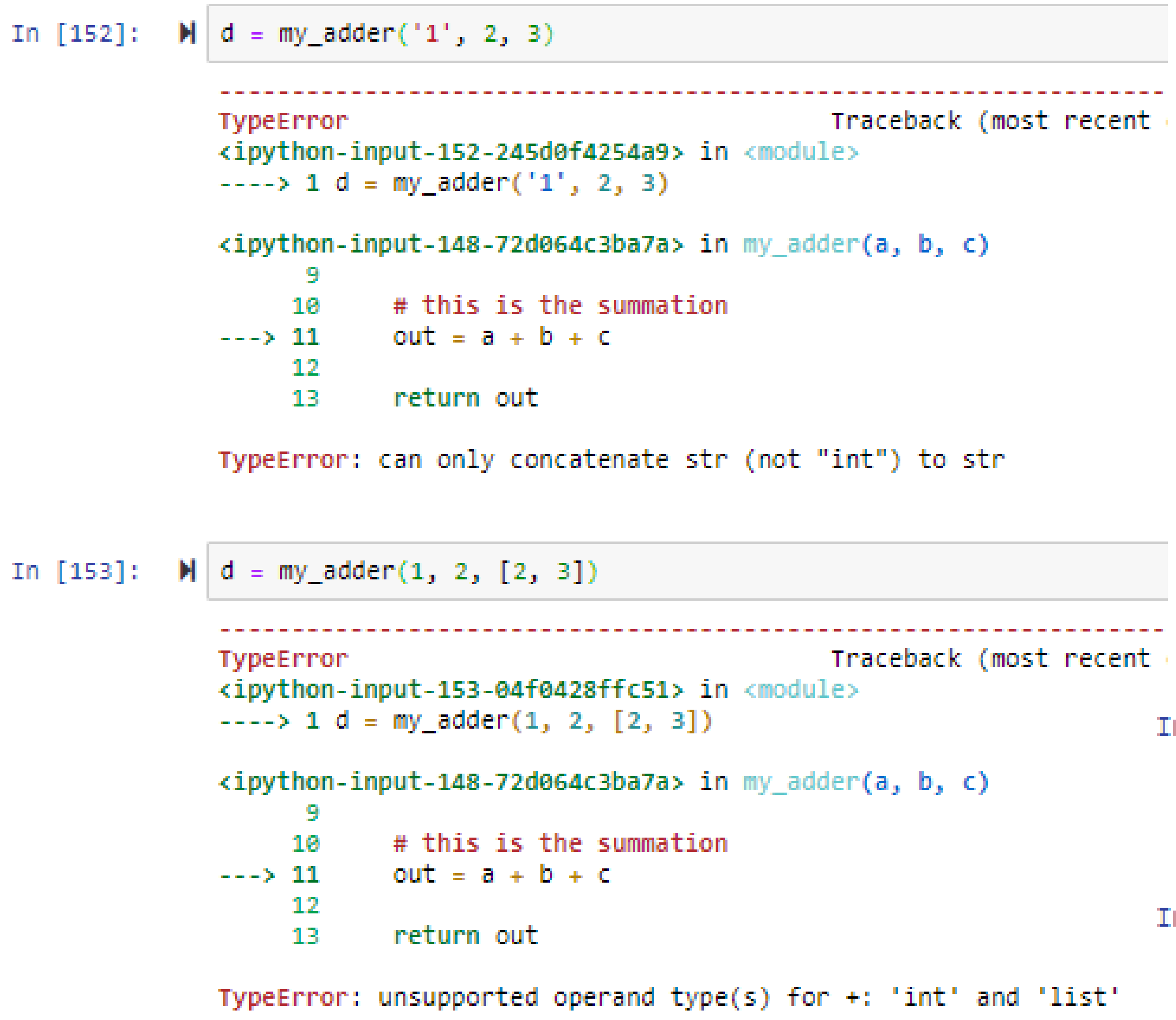
```
In [149]:  d = my_adder(1, 2, 3)
           d

Out[149]:  6
```

```
In [150]:  d = my_adder(4, 5, 6)
           d

Out[150]:  15
```

```
In [151]:  help(my_adder)

           Help on function my_adder in module __main__:

           my_adder(a, b, c)
               function to sum the 3 numbers
               Input: 3 numbers a, b, c
               Output: the sum of a, b, and c
               author:
               date:
```

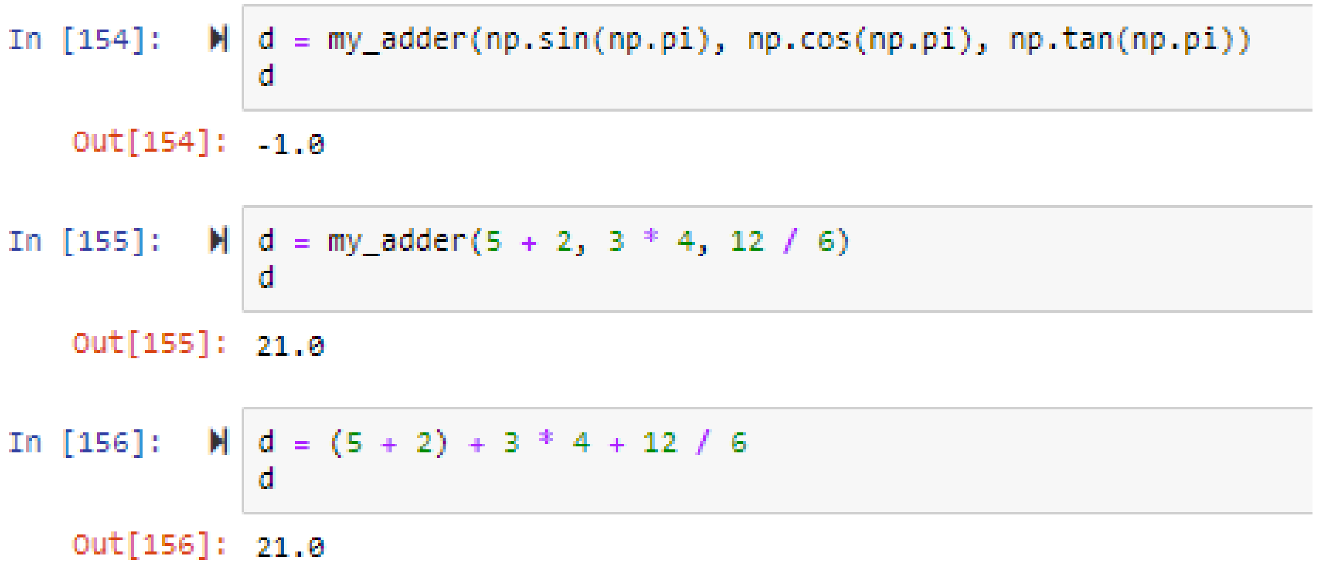
Functions

- Python gives the user tremendous **freedom to assign variables** to different data types. For example, it is possible to give the variable x a dictionary value or a float value. In other programming languages this is not always the case, you must declare at the beginning of a session whether x will be a dictionary or a float type, and then you're stuck with it. This can be both a benefit and a drawback.
- Remember to read the **errors** that Python gives you. They usually tell you exactly where the problem was.
- You can compose functions by assigning function calls as the input to other functions. In the order of operations, Python will execute the **innermost** function call first. You can also assign mathematical expressions as the input to functions. In this case, Python will execute the mathematical expressions first.



```
In [152]:   d = my_adder('1', 2, 3)

            ---------------------------------------------------------------------------
            TypeError                                 Traceback (most recent
            <ipython-input-152-245d0f4254a9> in <module>
            ----> 1 d = my_adder('1', 2, 3)

            <ipython-input-148-72d064c3ba7a> in my_adder(a, b, c)
                  9
                 10     # this is the summation
            ---> 11     out = a + b + c
                 12
                 13     return out

            TypeError: can only concatenate str (not "int") to str
```

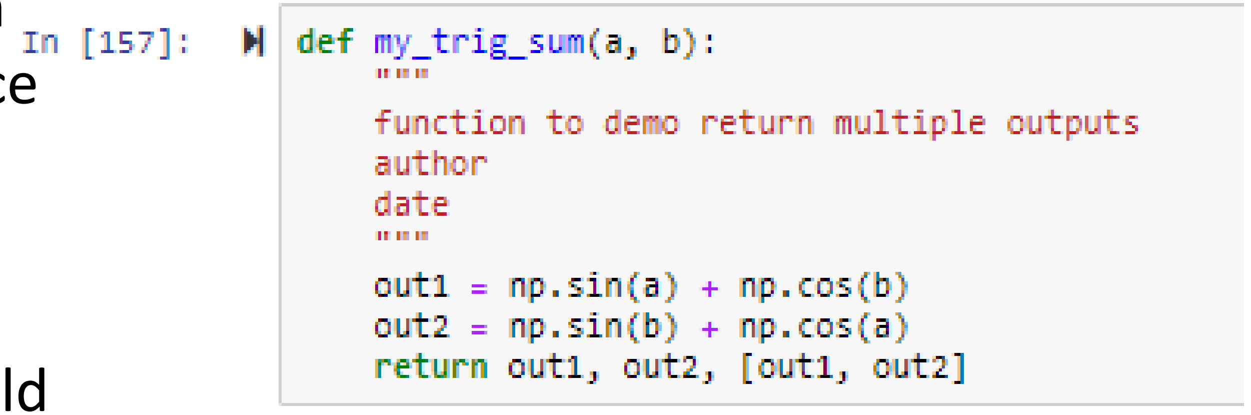
```
In [153]:   d = my_adder(1, 2, [2, 3])

            ---------------------------------------------------------------------------
            TypeError                                 Traceback (most recent
            <ipython-input-153-04f0428ffc51> in <module>
            ----> 1 d = my_adder(1, 2, [2, 3])

            <ipython-input-148-72d064c3ba7a> in my_adder(a, b, c)
                  9
                 10     # this is the summation
            ---> 11     out = a + b + c
                 12
                 13     return out

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



```
In [154]:   d = my_adder(np.sin(np.pi), np.cos(np.pi), np.tan(np.pi))
            d

Out[154]:   -1.0

In [155]:   d = my_adder(5 + 2, 3 * 4, 12 / 6)
            d

Out[155]:   21.0

In [156]:   d = (5 + 2) + 3 * 4 + 12 / 6
            d

Out[156]:   21.0
```

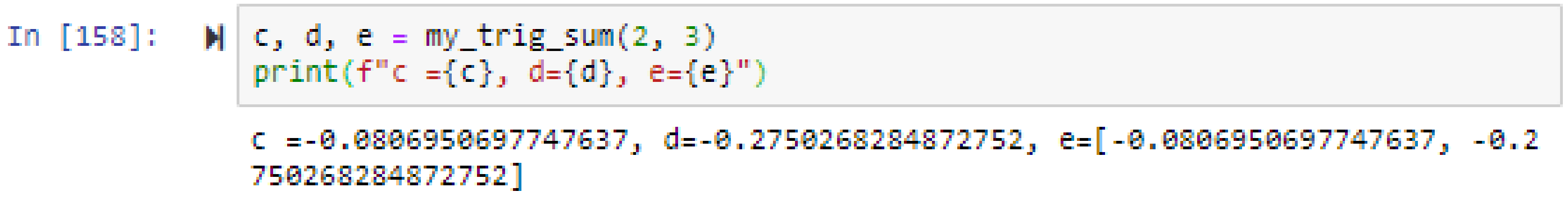
Functions

- Python functions can have **multiple output** parameters. When calling a function with multiple output parameters, you can place the multiple variables you want assigned separated by commas. The function essentially will return the multiple result parameters in a tuple, therefore, you could unpack the returned tuple.

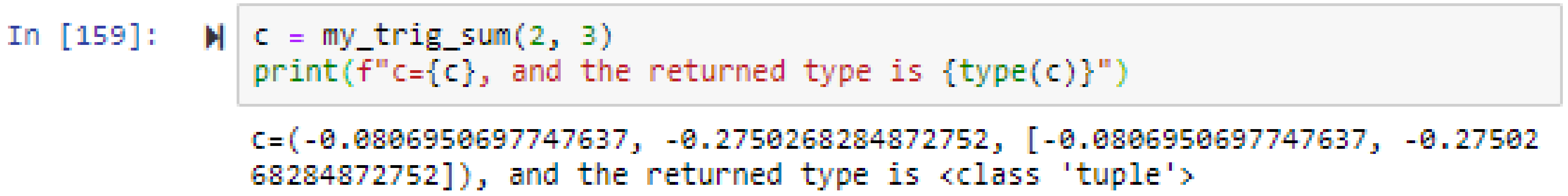


```
In [157]:  def my_trig_sum(a, b):
               """
               function to demo return multiple outputs
               author
               date
               """
               out1 = np.sin(a) + np.cos(b)
               out2 = np.sin(b) + np.cos(a)
               return out1, out2, [out1, out2]
```

- If you assign the results to one variable, you will get a **tuple** that has all the output parameters.



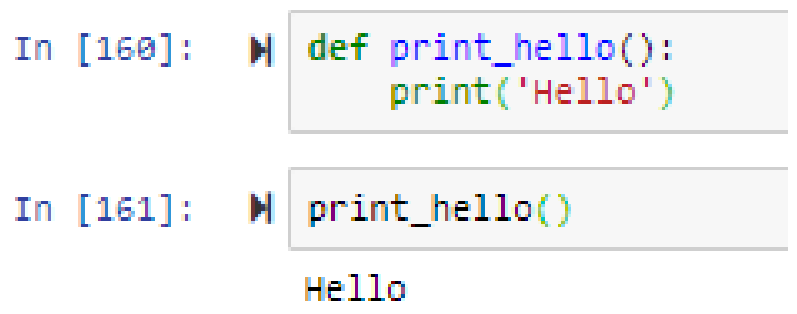
```
In [158]:  c, d, e = my_trig_sum(2, 3)
           print(f"c ={c}, d={d}, e={e}")

           c =-0.0806950697747637, d=-0.2750268284872752, e=[-0.0806950697747637, -0.2
           750268284872752]
```



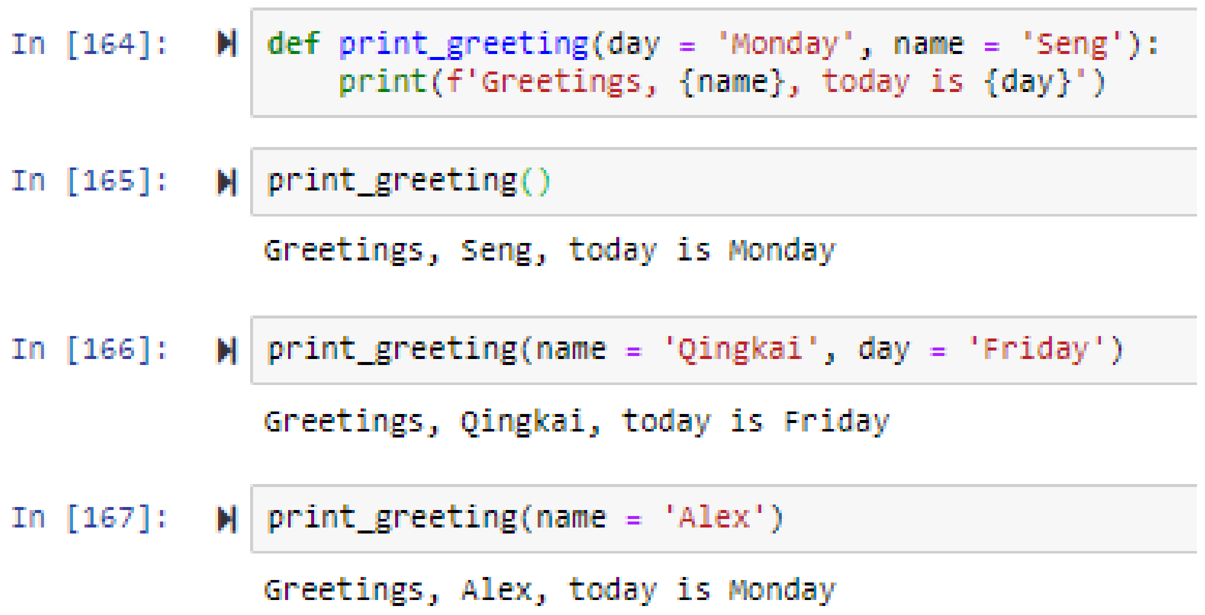
```
In [159]:  c = my_trig_sum(2, 3)
           print(f"c={c}, and the returned type is {type(c)}")

           c=(-0.0806950697747637, -0.2750268284872752, [-0.0806950697747637, -0.27502
           68284872752]), and the returned type is <class 'tuple'>
```

Functions

- A function could be defined **without an input** argument and **returning any value**.



```
In [160]:  def print_hello():
               print('Hello')

In [161]:  print_hello()

           Hello
```

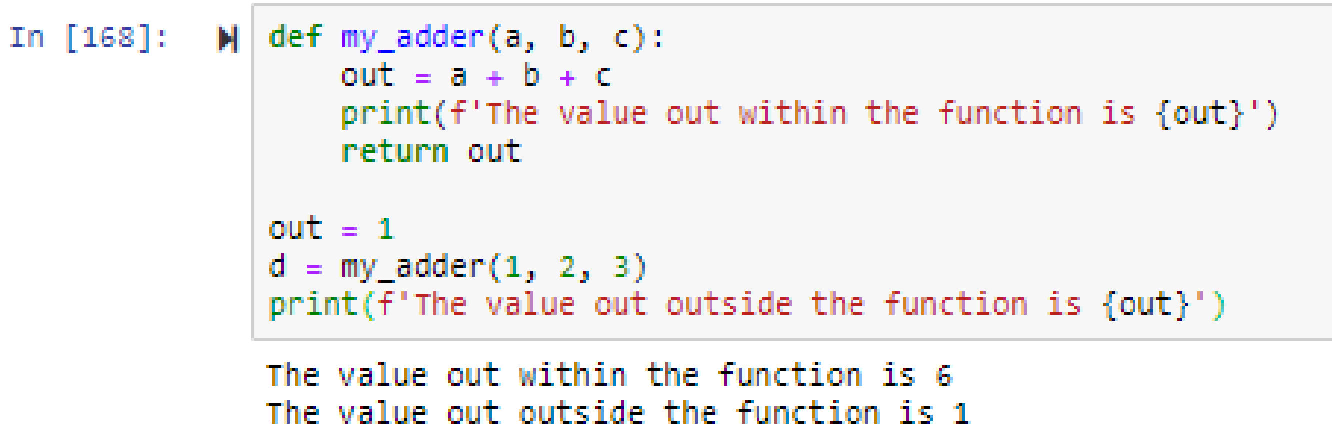


```
In [164]:  def print_greeting(day = 'Monday', name = 'Seng'):
               print(f'Greetings, {name}, today is {day}')

In [165]:  print_greeting()

           Greetings, Seng, today is Monday

In [166]:  print_greeting(name = 'Qingkai', day = 'Friday')

           Greetings, Qingkai, today is Friday

In [167]:  print_greeting(name = 'Alex')

           Greetings, Alex, today is Monday
```

- For the input of the argument, we can have the **default value** as well.
- We can see that, if we give a value to the argument when we define the function, this value will be the default value of the function. If the user doesn’t provide an input to this argument, then this default value will be used during calling of the function. Besides, the order of the argument is not important when calling the function if you provide the name of the argument.

Local and Global Variables

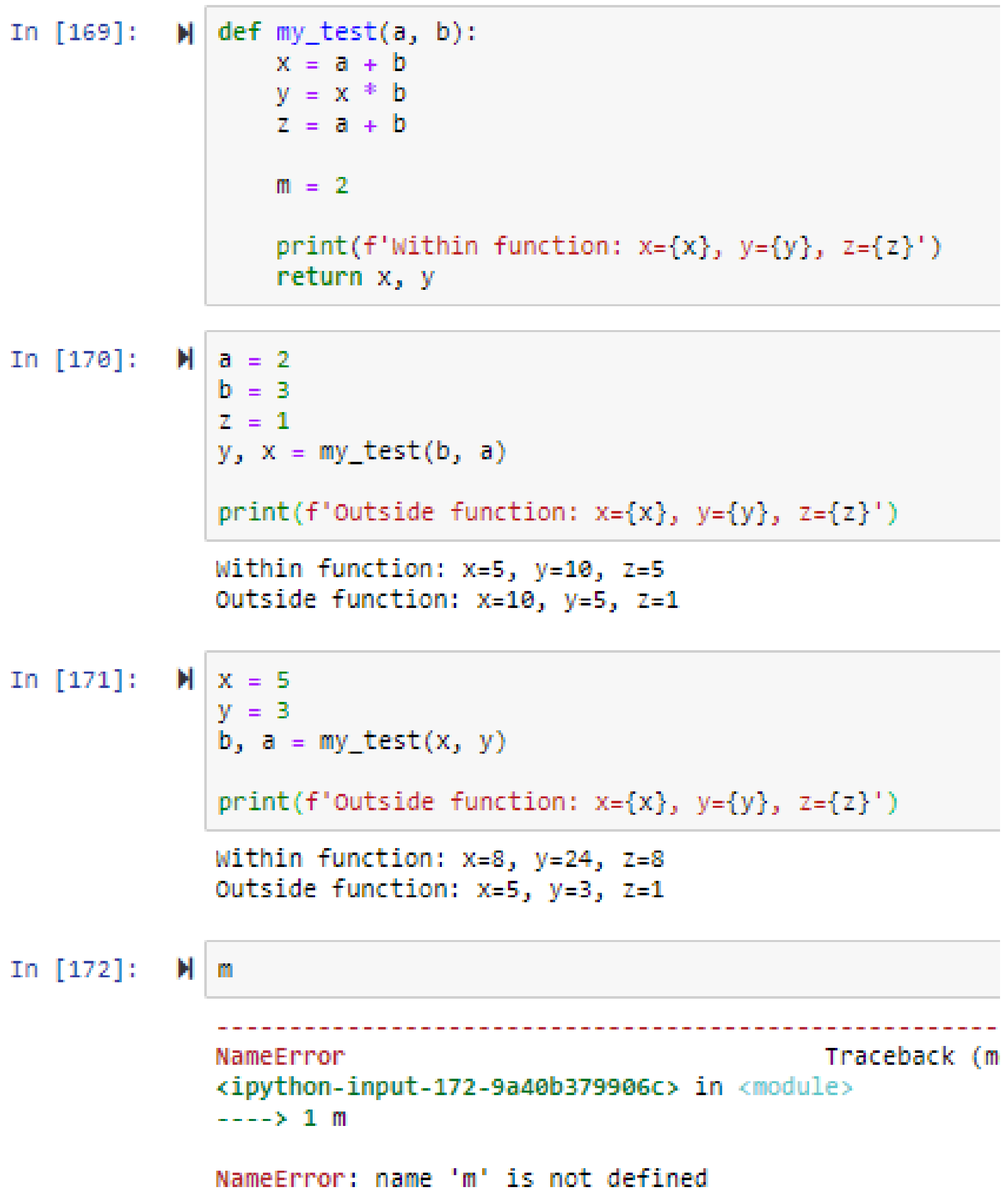
- A function has its own **memory block** that is reserved for variables created within that function. This block of memory is not shared with the whole notebook memory block. Therefore, a variable with a given name can be assigned within a function without changing a variable with the same name outside of the function. The memory block associated with the function is opened every time a function is used.



```
In [168]:  def my_adder(a, b, c):
               out = a + b + c
               print(f'The value out within the function is {out}')
               return out

           out = 1
           d = my_adder(1, 2, 3)
           print(f'The value out outside the function is {out}')

           The value out within the function is 6
           The value out outside the function is 1
```

- In **my_adder**, the variable out is a local variable. That is, it is only defined in the function of my_adder. Therefore, it cannot affect variables outside of the function, and actions taken in the notebook outside the function cannot affect it, even if they have the same name.

- Consider the following function.



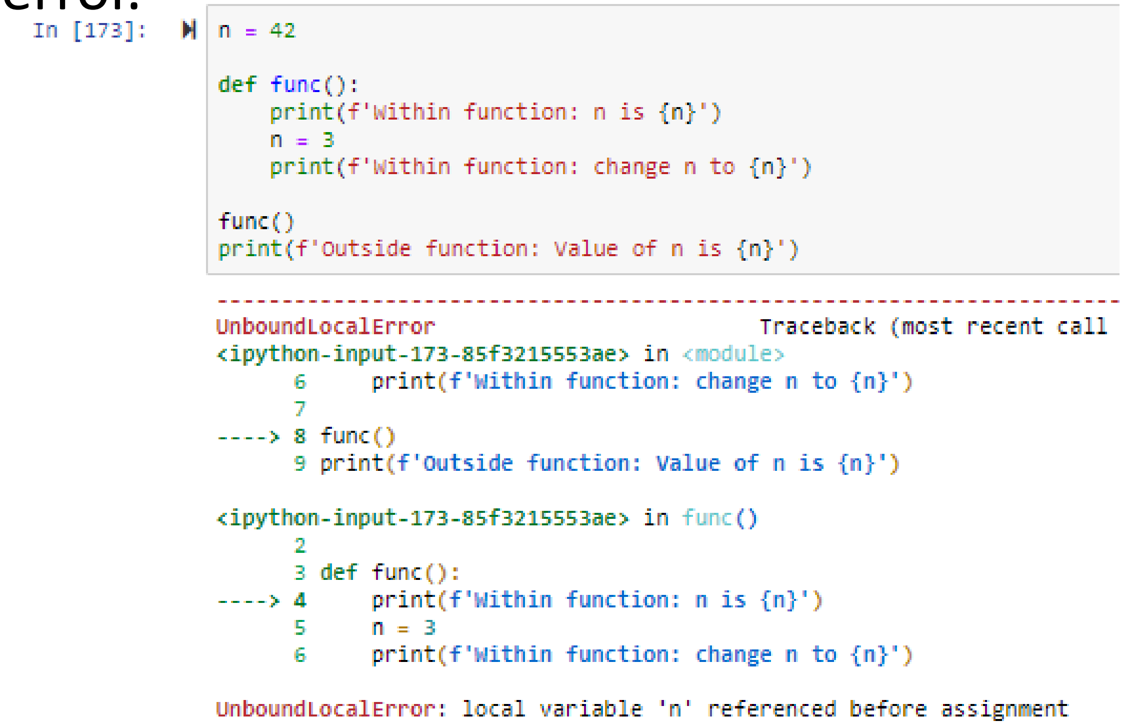
```
In [169]:  def my_test(a, b):
               x = a + b
               y = x * b
               z = a + b

               m = 2

               print(f'Within function: x={x}, y={y}, z={z}')
               return x, y
```

```
In [170]:  a = 2
           b = 3
           z = 1
           y, x = my_test(b, a)

           print(f'Outside function: x={x}, y={y}, z={z}')

           Within function: x=5, y=10, z=5
           Outside function: x=10, y=5, z=1
```

```
In [171]:  x = 5
           y = 3
           b, a = my_test(x, y)

           print(f'Outside function: x={x}, y={y}, z={z}')

           Within function: x=8, y=24, z=8
           Outside function: x=5, y=3, z=1
```

```
In [172]:  m

           ---------------------------------------------------------------
           NameError                                 Traceback (m
           <ipython-input-172-9a40b379906c> in <module>
           ----> 1 m

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

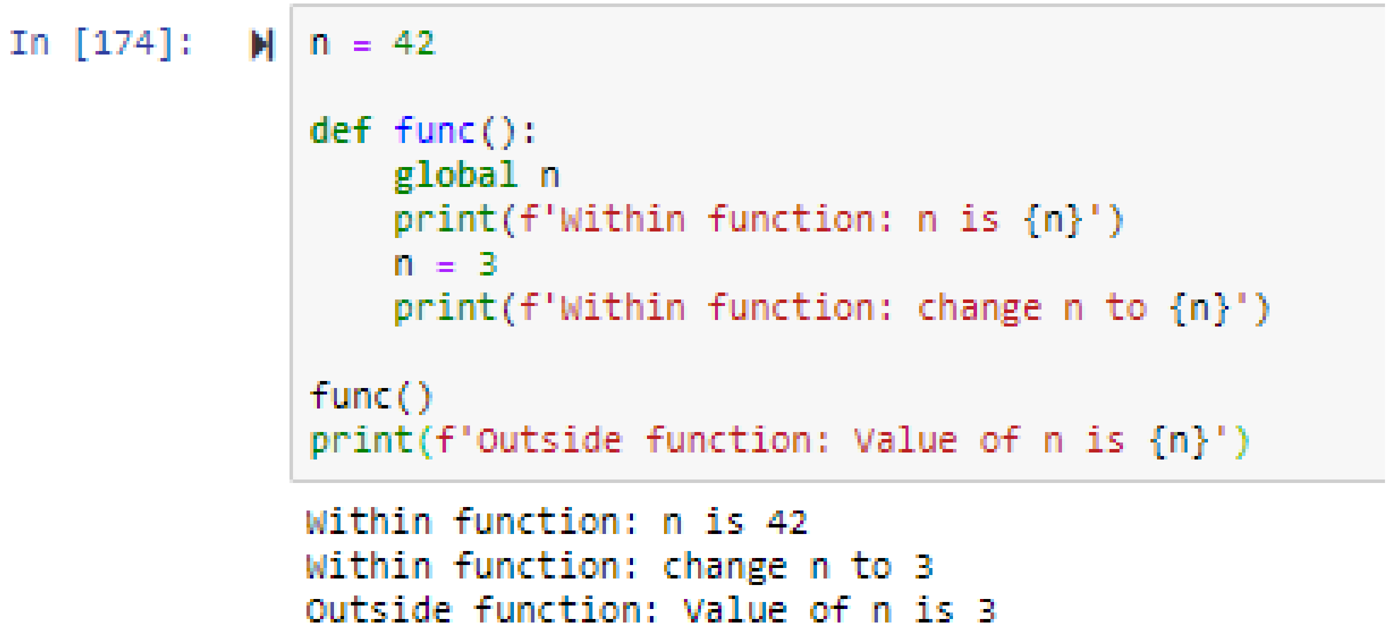
- We can see the value m is not defined outside of the function, since it is defined within the function.
- The opposite is similar, for example, if you define a variable outside a function, but you want to use it inside the function and change the value, you will get the same error.



```
In [173]:  n = 42

           def func():
               print(f'Within function: n is {n}')
               n = 3
               print(f'Within function: change n to {n}')

           func()
           print(f'Outside function: Value of n is {n}')

           ---------------------------------------------------------------------------
           UnboundLocalError                         Traceback (most recent call
           <ipython-input-173-85f3215553ae> in <module>
                 6     print(f'Within function: change n to {n}')
                 7
           ----> 8 func()
                 9 print(f'Outside function: Value of n is {n}')

           <ipython-input-173-85f3215553ae> in func()
                 2
                 3 def func():
           ----> 4     print(f'Within function: n is {n}')
                 5     n = 3
                 6     print(f'Within function: change n to {n}')

           UnboundLocalError: local variable 'n' referenced before assignment
```

Local and Global Variables

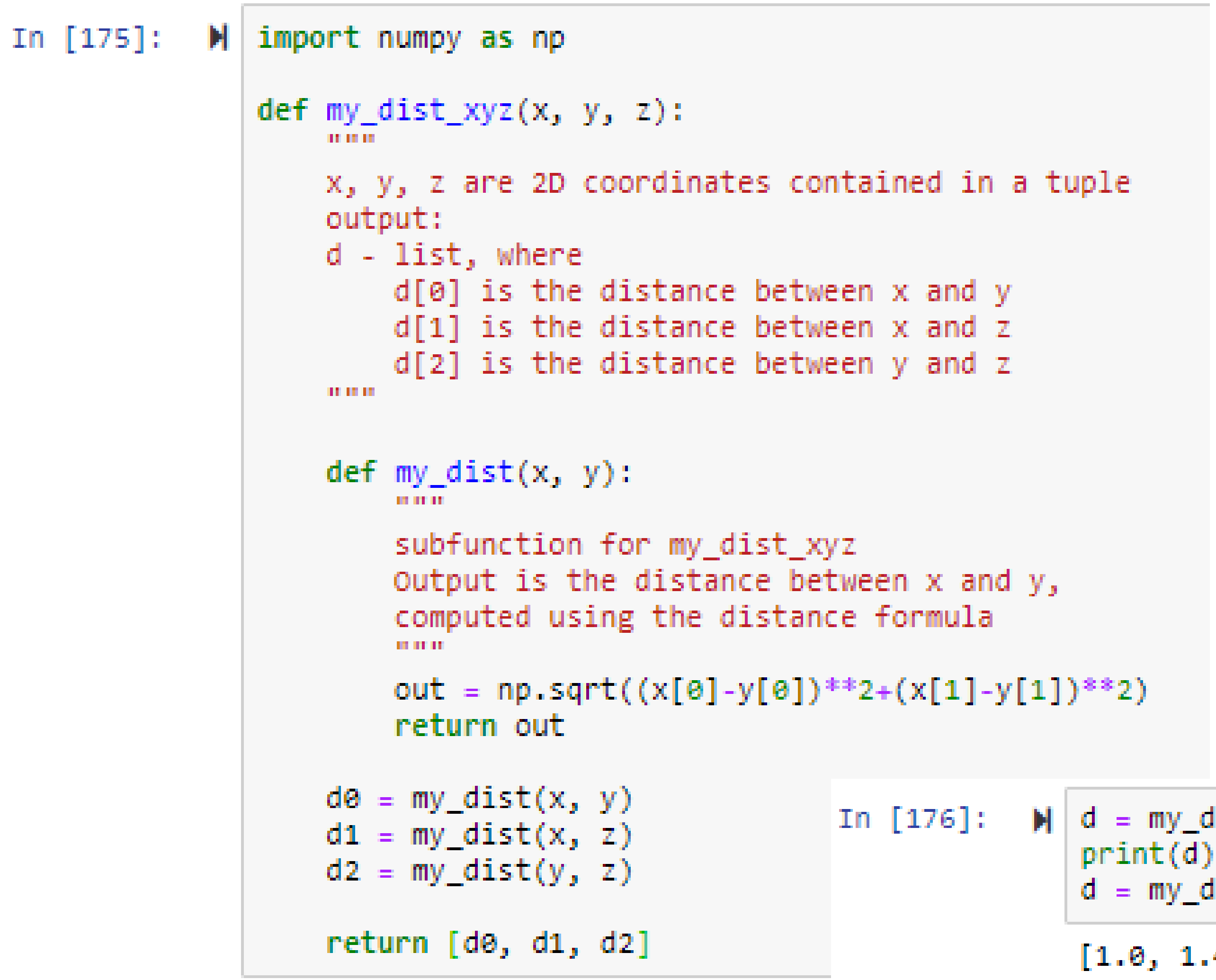
- The solution is to use the keyword **global** to let Python know this variable is a global variable that it can be used both outside and inside the function.



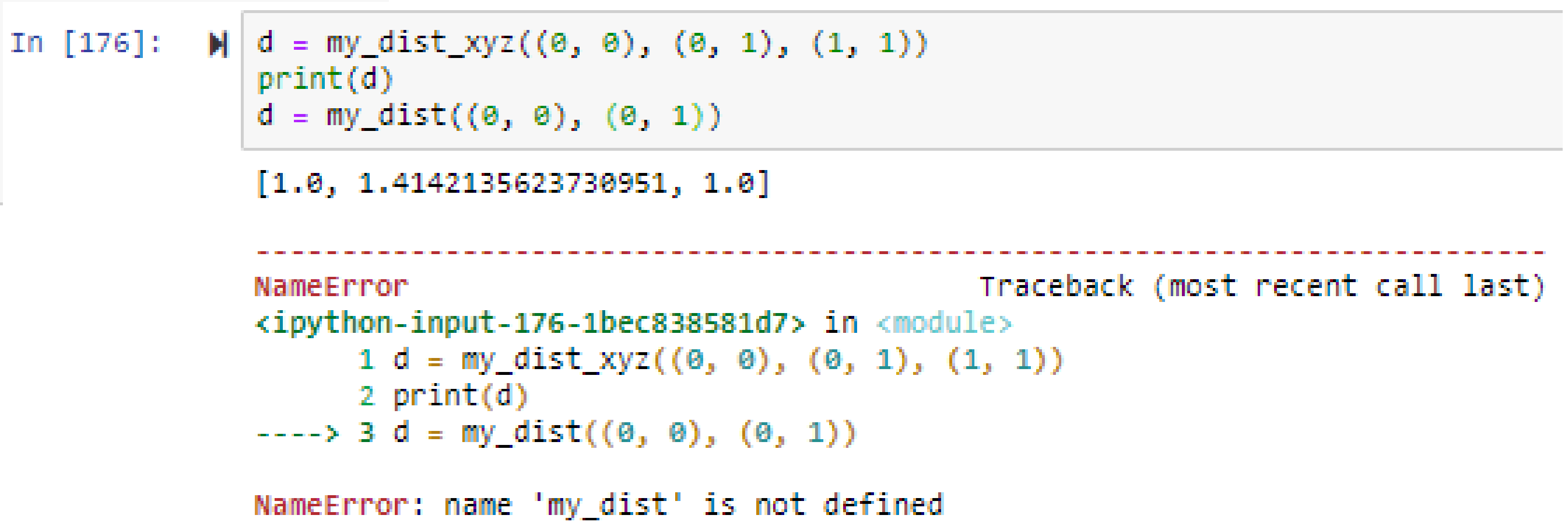
```
In [174]:   n = 42

            def func():
                global n
                print(f'Within function: n is {n}')
                n = 3
                print(f'Within function: change n to {n}')

            func()
            print(f'Outside function: Value of n is {n}')

            Within function: n is 42
            Within function: change n to 3
            Outside function: Value of n is 3
```

Nested Functions

- A **nested function** is a function that is defined within another function - parent function. Only the parent function is able to call the nested function. However, the nested function retains a separate memory block from its parent function.
- Consider the following function and nested function shown on the next slide.
- Notice that the variables x and y appear in both **my_dist_xyz** and **my_dist**. This is permissible because a nested function has a separate memory block from its parent function.
- Nested functions are useful when a task must be performed many times within the function but not outside the function. In this way, nested functions help the parent function perform its task while hiding in the parent function.



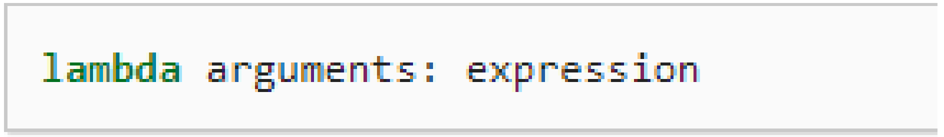
```
In [175]:   import numpy as np

            def my_dist_xyz(x, y, z):
                """
                x, y, z are 2D coordinates contained in a tuple
                output:
                d - list, where
                    d[0] is the distance between x and y
                    d[1] is the distance between x and z
                    d[2] is the distance between y and z
                """

                def my_dist(x, y):
                    """
                    subfunction for my_dist_xyz
                    Output is the distance between x and y,
                    computed using the distance formula
                    """
                    out = np.sqrt((x[0]-y[0])**2+(x[1]-y[1])**2)
                    return out

                d0 = my_dist(x, y)
                d1 = my_dist(x, z)
                d2 = my_dist(y, z)

                return [d0, d1, d2]
```



```
In [176]:   d = my_dist_xyz((0, 0), (0, 1), (1, 1))
            print(d)
            d = my_dist((0, 0), (0, 1))

            [1.0, 1.4142135623730951, 1.0]

            ---------------------------------------------------------------------------
            NameError                                 Traceback (most recent call last)
            <ipython-input-176-1bec838581d7> in <module>
                  1 d = my_dist_xyz((0, 0), (0, 1), (1, 1))
                  2 print(d)
            ----> 3 d = my_dist((0, 0), (0, 1))

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

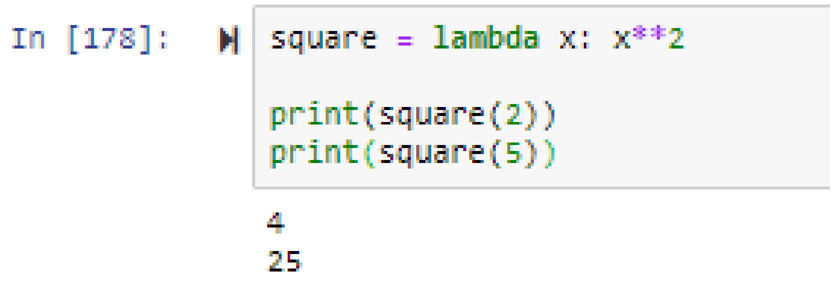
Lambda Functions

- Sometimes, we don't want to use the normal way to define a function, especially if our function is just one line. In this case, we can use **anonymous** function in Python, which is a function that is defined without a name. This type of functions also called **lambda** function, since they are defined using the **lambda** keyword.
- A typical lambda function is defined:

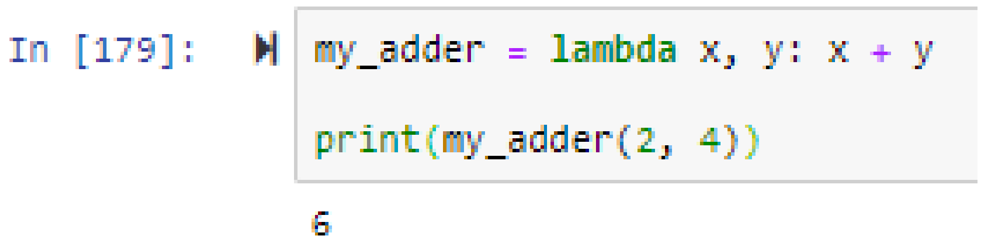


```
lambda arguments: expression
```

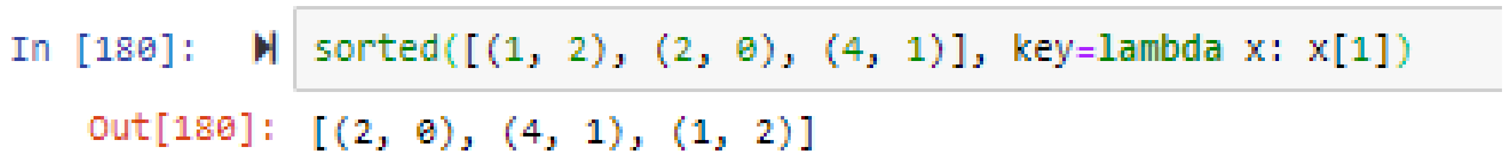
- It can have any number of arguments, but with only one expression.



```
In [178]:  square = lambda x: x**2

            print(square(2))
            print(square(5))

            4
            25
```



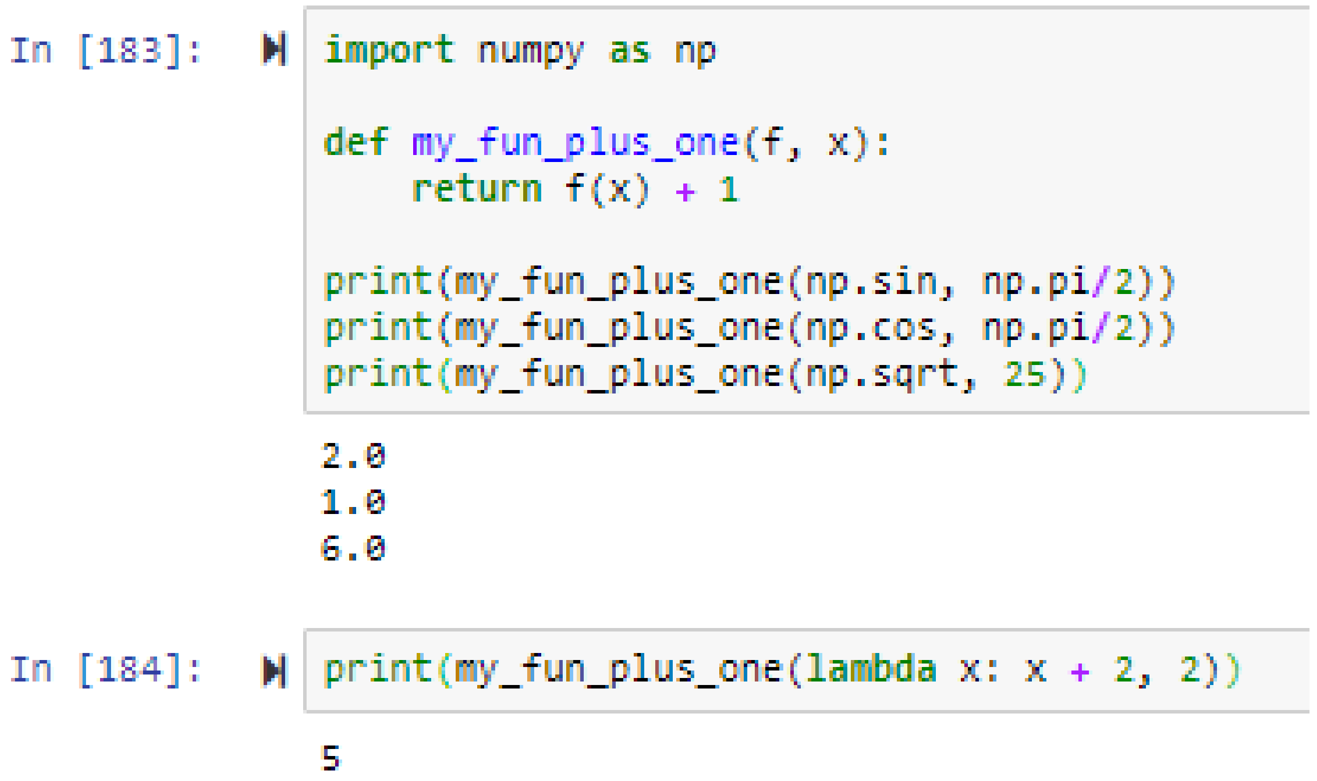
```
In [179]:  my_adder = lambda x, y: x + y

           print(my_adder(2, 4))

           6
```



```
In [180]:  sorted([(1, 2), (2, 0), (4, 1)], key=lambda x: x[1])

Out[180]:  [(2, 0), (4, 1), (1, 2)]
```

Functions as Arguments to Functions

- Sometimes it is useful to be able to pass a function as a variable to another function.
- In other words, the input to some functions may be other functions.



```
In [183]:  import numpy as np

           def my_fun_plus_one(f, x):
               return f(x) + 1

           print(my_fun_plus_one(np.sin, np.pi/2))
           print(my_fun_plus_one(np.cos, np.pi/2))
           print(my_fun_plus_one(np.sqrt, 25))

           2.0
           1.0
           6.0

In [184]:  print(my_fun_plus_one(lambda x: x + 2, 2))

           5
```

Branching Statements

- When writing functions, it is very common to want certain parts of the function body to be executed only under certain conditions.
- For example, if the input argument is odd, you may want the function to perform one operation on it, and another if the input argument is even.
- This effect can be achieved in Python using **branching** statements (i.e., the execution of the function branches under certain conditions).

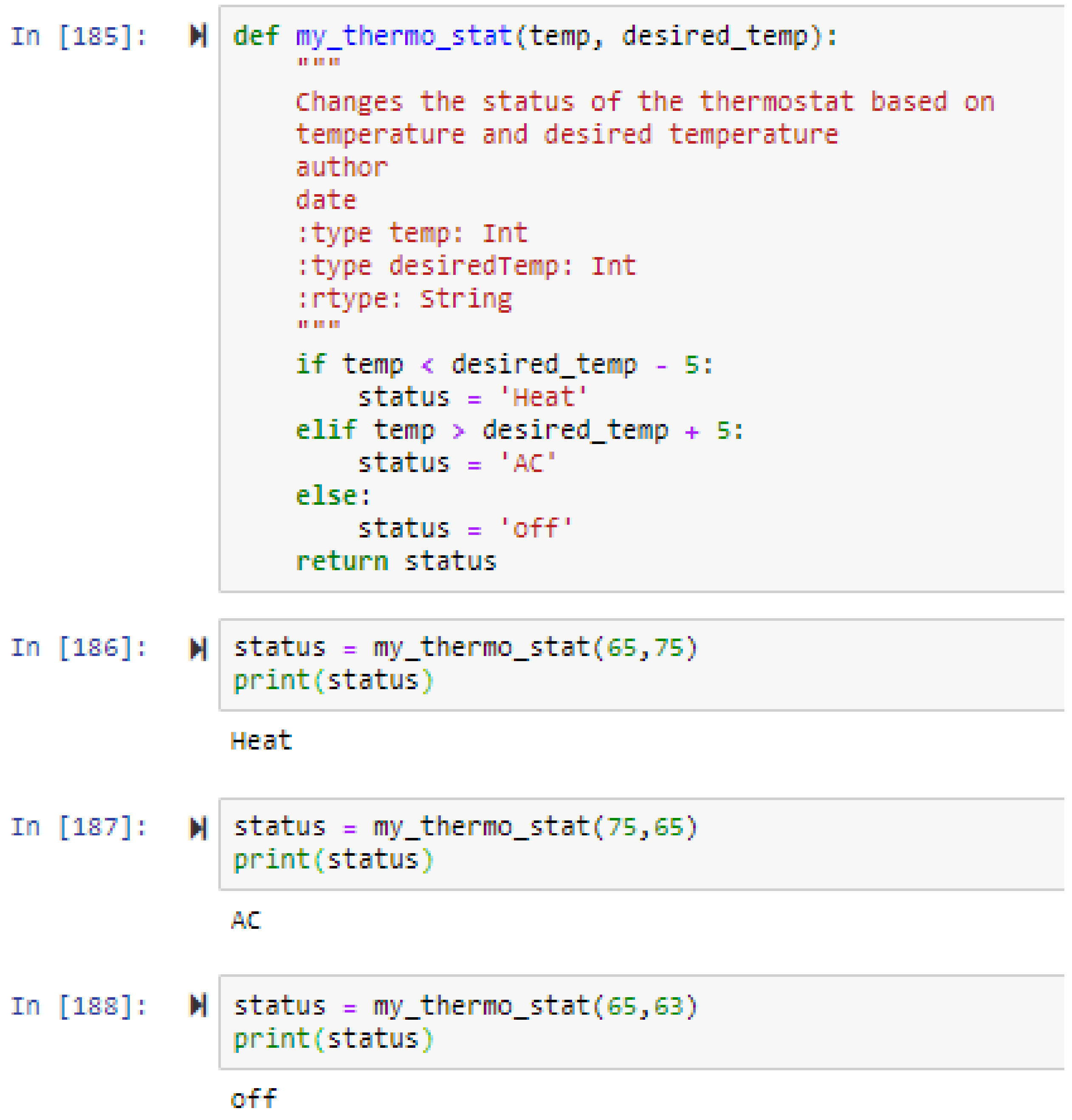
If-Else Statements

- A branching statement, **If-Else** Statement, or **If-Statement** for short, is a code construct that executes blocks of code only if certain conditions are met. These conditions are represented as logical expressions. Let P, Q, and R be some logical expressions in Python. The following shows an if-statement construction.

```
if logical expression:
    code block
```

```
if logical expression P:
    code block 1
elif logical expression Q:
    code block 2
elif logical expression R:
    code block 3
else:
    code block 4
```

- The word "**if**" is a keyword. When Python sees an if-statement, it will determine if the associated logical expression is true. If it is true, then the code in code block will be executed. If it is false, then the code in the if-statement will not be executed. The way to read this is "If logical expression is true then do code block."
- When there are several conditions to consider you can include **elif**-statements; if you want a condition that covers any other case, then you can use an **else** statement.

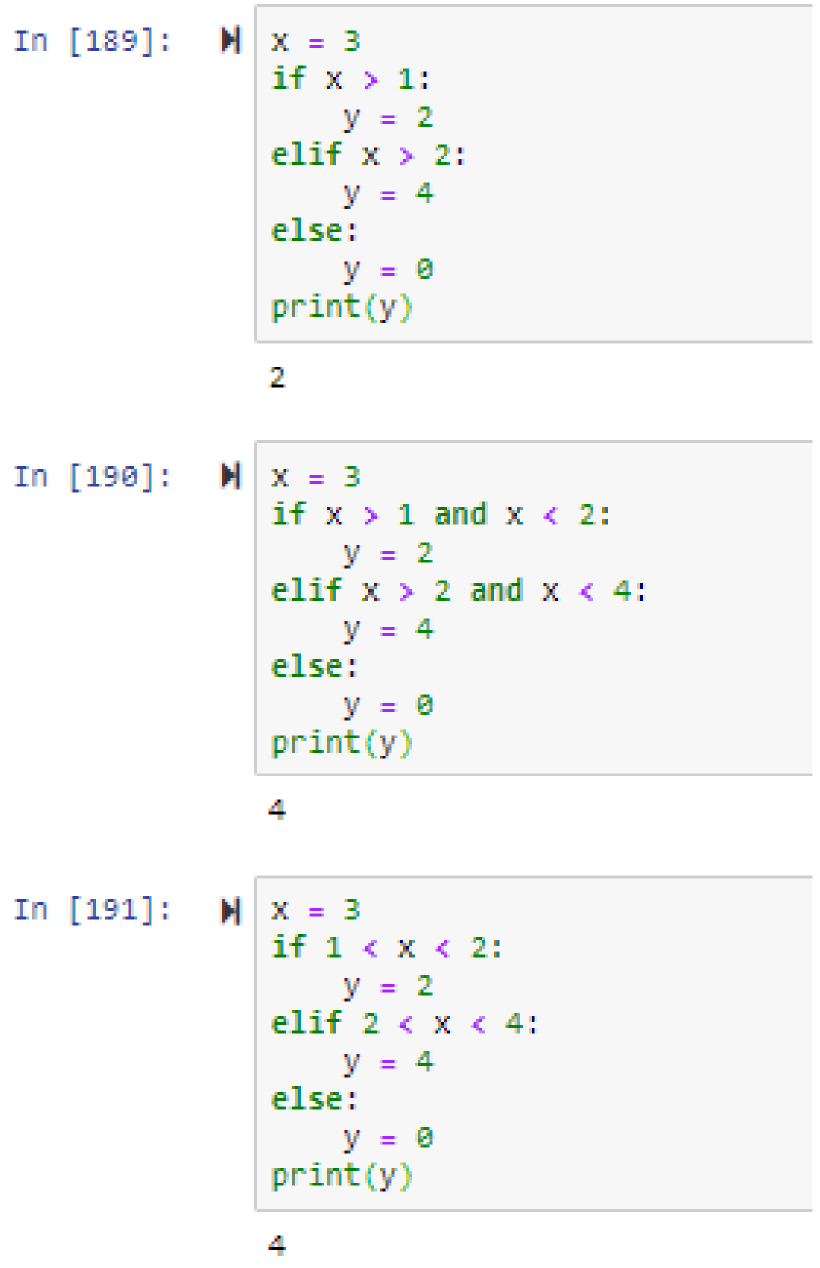


```
In [185]:  def my_thermo_stat(temp, desired_temp):
                """
                Changes the status of the thermostat based on
                temperature and desired temperature
                author
                date
                :type temp: Int
                :type desiredTemp: Int
                :rtype: String
                """
                if temp < desired_temp - 5:
                    status = 'Heat'
                elif temp > desired_temp + 5:
                    status = 'AC'
                else:
                    status = 'off'
                return status
```

```
In [186]:  status = my_thermo_stat(65,75)
           print(status)

           Heat
```

```
In [187]:  status = my_thermo_stat(75,65)
           print(status)

           AC
```

```
In [188]:  status = my_thermo_stat(65,63)
           print(status)

           off
```



```
In [189]:  x = 3
           if x > 1:
               y = 2
           elif x > 2:
               y = 4
           else:
               y = 0
           print(y)

           2
```

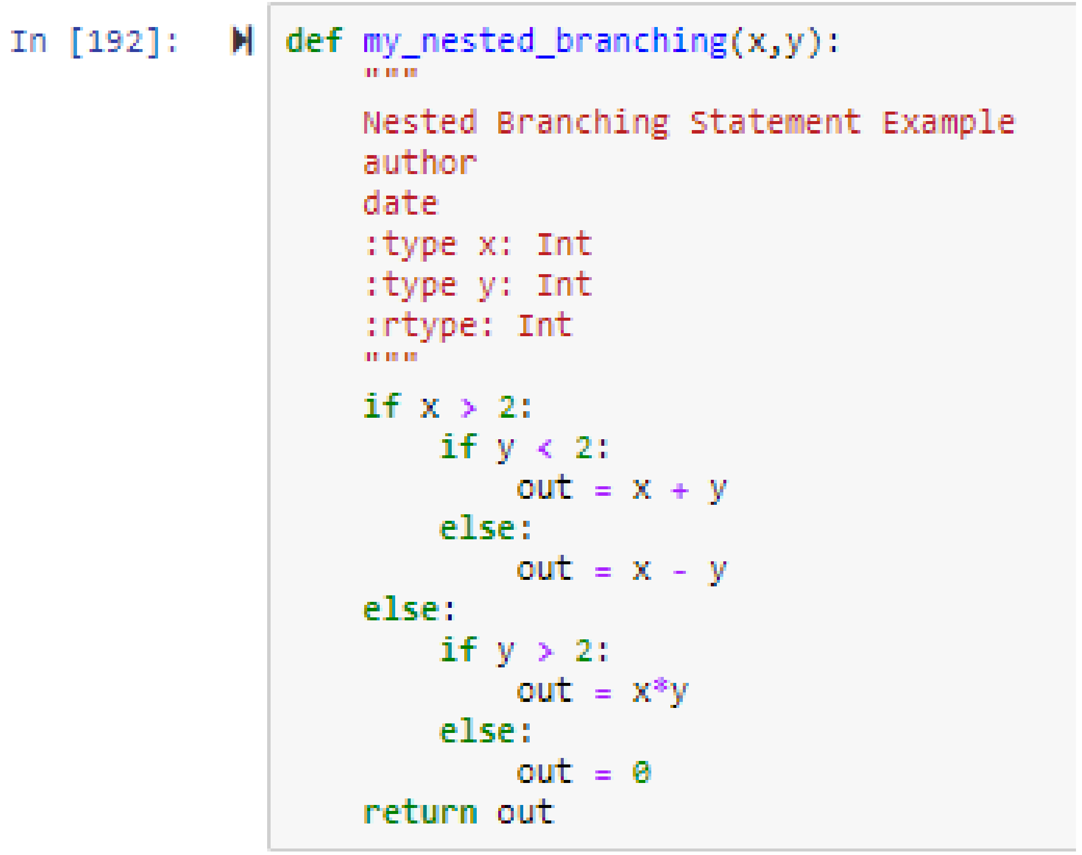
```
In [190]:  x = 3
           if x > 1 and x < 2:
               y = 2
           elif x > 2 and x < 4:
               y = 4
           else:
               y = 0
           print(y)

           4
```

```
In [191]:  x = 3
           if 1 < x < 2:
               y = 2
           elif 2 < x < 4:
               y = 4
           else:
               y = 0
           print(y)

           4
```

If-Else Statements

- A statement is called nested if it is entirely contained within another statement of the same type as itself. For example, a **nested if-statement** is an if-statement that is entirely contained within a clause of another if-statement.

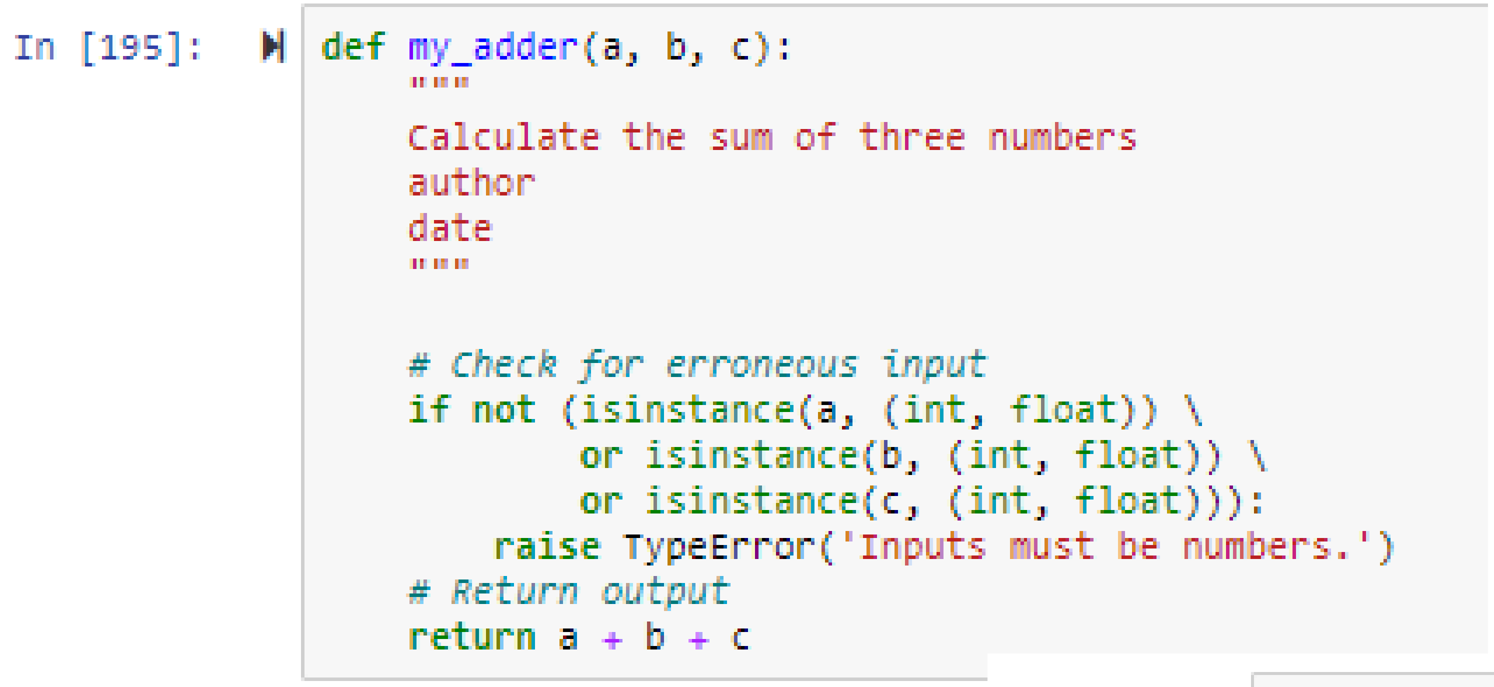
- As before, Python gives the same level of **indentation** to every line of code within a conditional statement.
- The nested if-statement have deeper indentation by increasing **four white spaces**.
- You will get an **IndentationError** if the indentation is not correct.



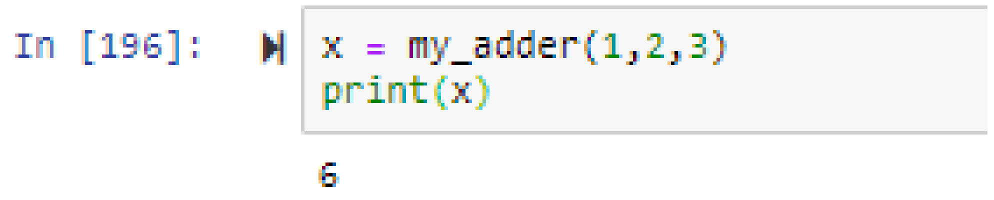
```
In [192]:   def my_nested_branching(x,y):
                """
                Nested Branching Statement Example
                author
                date
                :type x: Int
                :type y: Int
                :rtype: Int
                """
                if x > 2:
                    if y < 2:
                        out = x + y
                    else:
                        out = x - y
                else:
                    if y > 2:
                        out = x*y
                    else:
                        out = 0
                return out
```

If-Else Statements

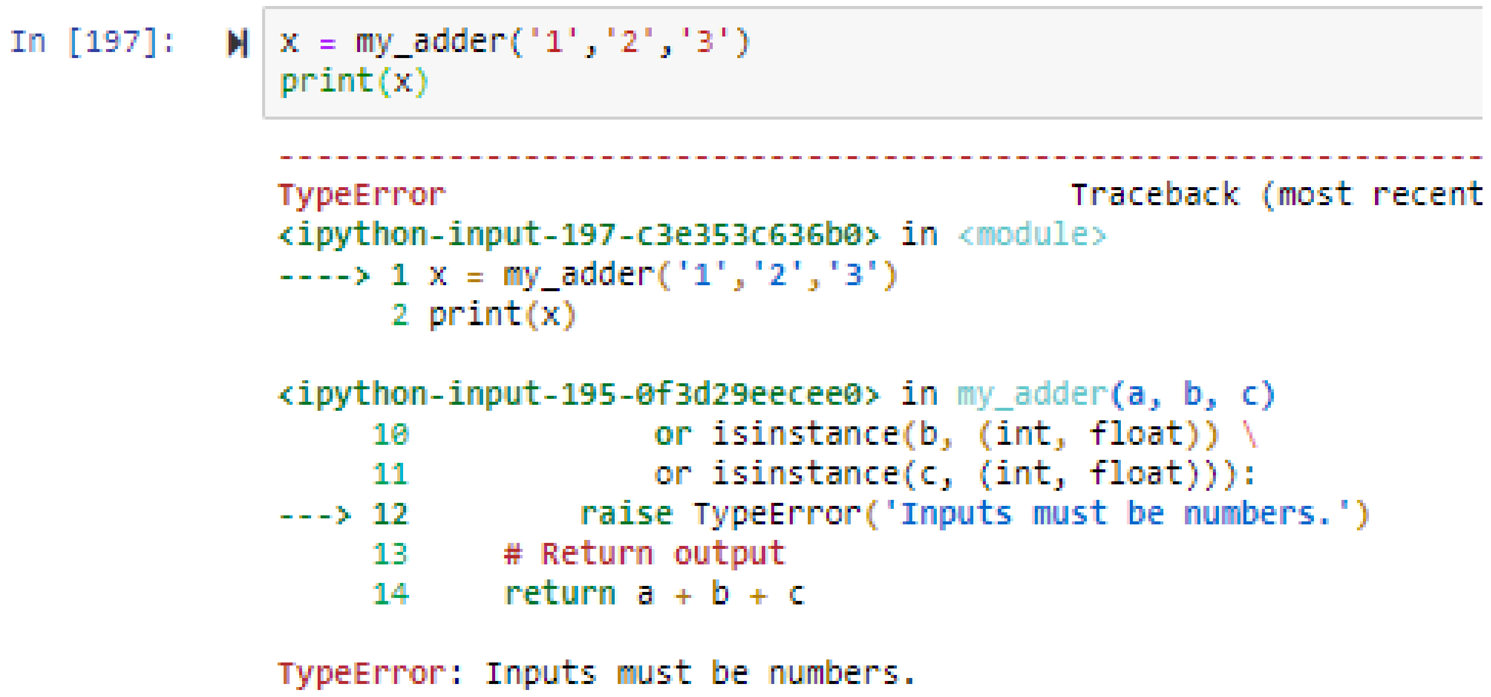
- Sometimes you may want to design your function to check the inputs of a function to ensure that your function will be used properly. For example, the function my_adder in the previous lesson expects doubles as input. If the user inputs a list or a string as one of the input variables, then the function will throw an **error** or have **unexpected** results. To prevent this, you can put a check to tell the user the function has not been used properly.
- There is a large variety of erroneous inputs that your function may encounter from users, and it is unreasonable to expect that your function will catch them all.
- Therefore, unless otherwise stated, write your functions assuming the functions will be used properly.



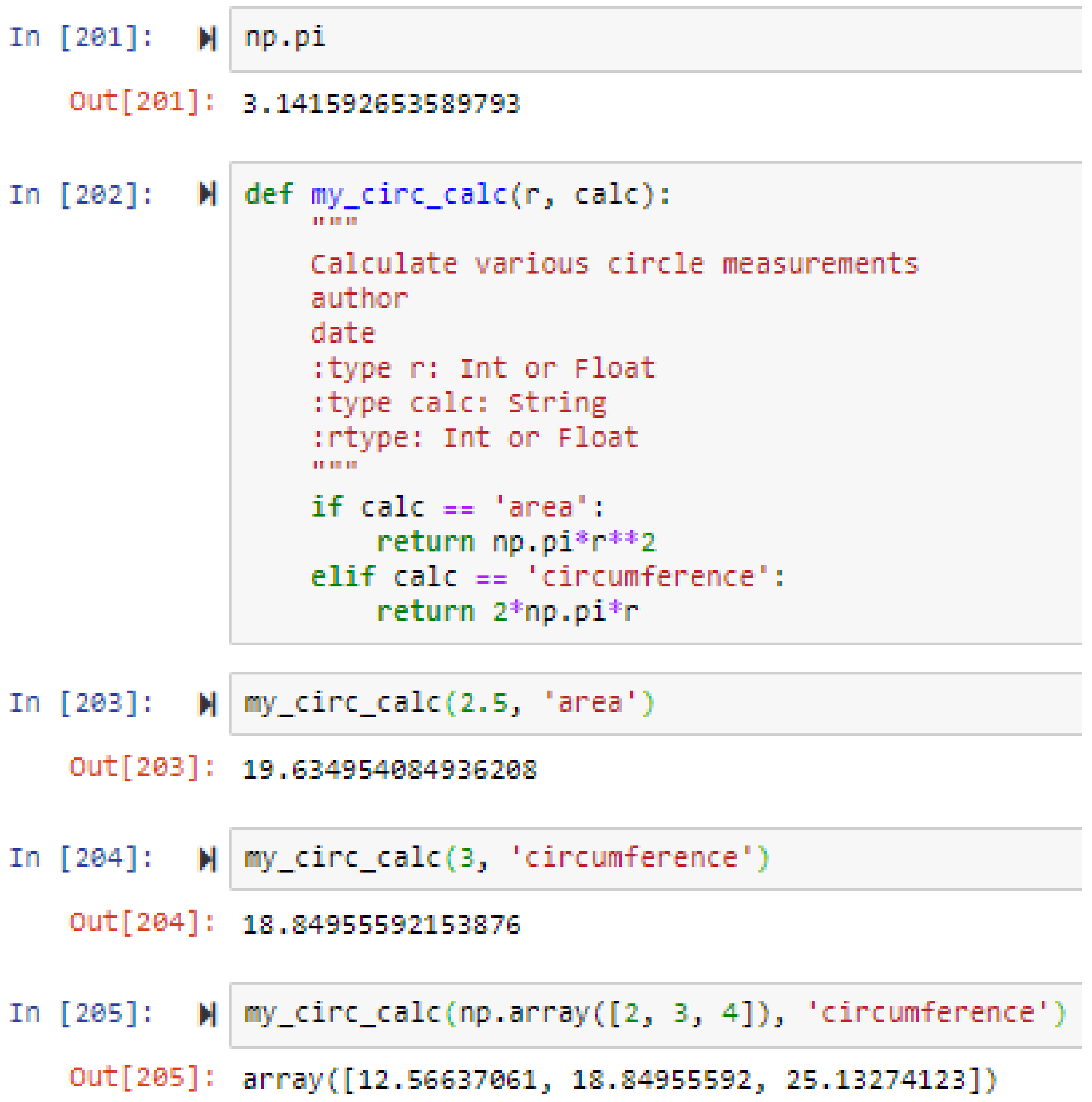
```
In [195]:  def my_adder(a, b, c):
               """
               Calculate the sum of three numbers
               author
               date
               """

               # Check for erroneous input
               if not (isinstance(a, (int, float)) \
                       or isinstance(b, (int, float)) \
                       or isinstance(c, (int, float))):
                   raise TypeError('Inputs must be numbers.')
               # Return output
               return a + b + c
```



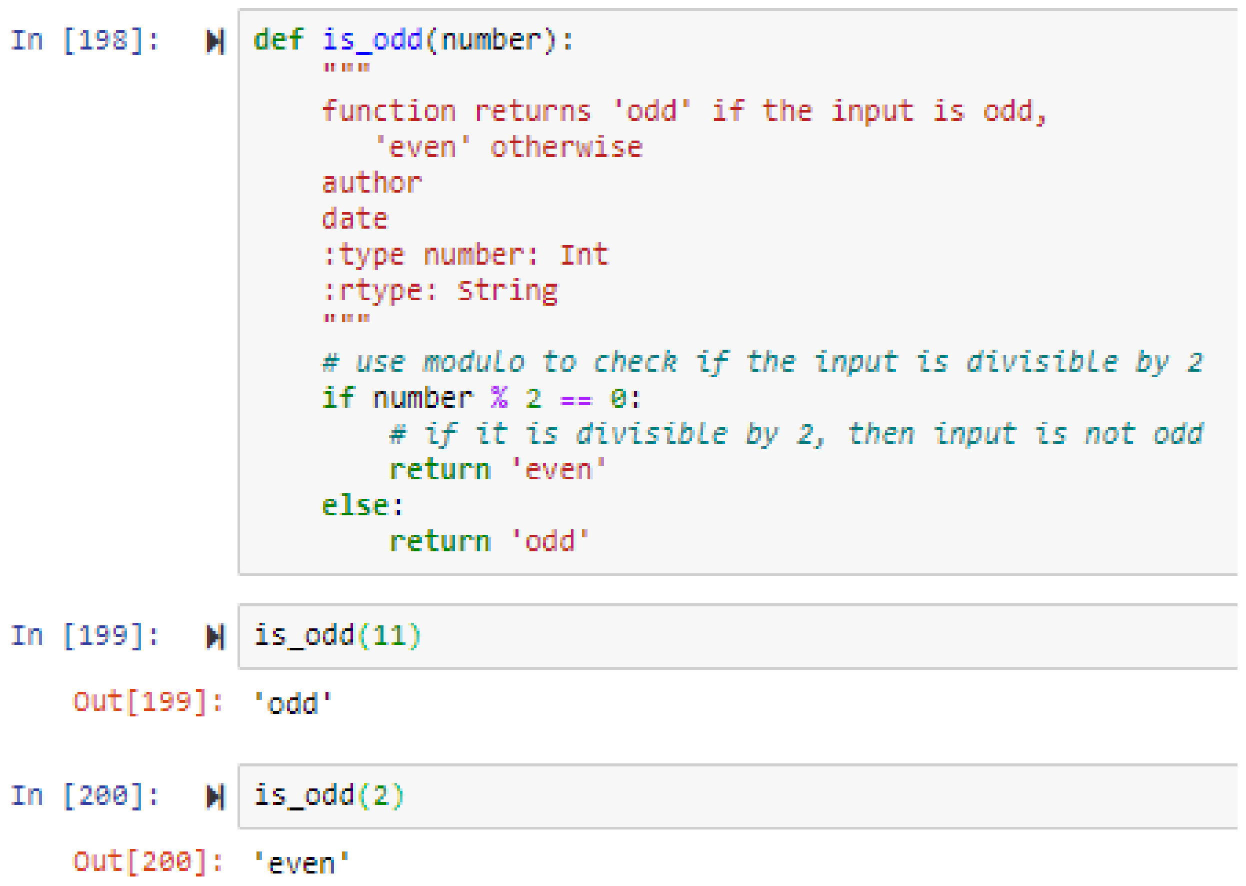
```
In [196]:  x = my_adder(1,2,3)
           print(x)

           6
```



```
In [197]:  x = my_adder('1','2','3')
           print(x)

           ---------------------------------------------------------------------------
           TypeError                                 Traceback (most recent
           <ipython-input-197-c3e353c636b0> in <module>
           ----> 1 x = my_adder('1','2','3')
                 2 print(x)

           <ipython-input-195-0f3d29eecee0> in my_adder(a, b, c)
                10             or isinstance(b, (int, float)) \
                11             or isinstance(c, (int, float))):
           ---> 12         raise TypeError('Inputs must be numbers.')
                13     # Return output
                14     return a + b + c

           TypeError: Inputs must be numbers.
```



```
In [201]:  np.pi

Out[201]:  3.141592653589793

In [202]:  def my_circ_calc(r, calc):
               """
               Calculate various circle measurements
               author
               date
               :type r: Int or Float
               :type calc: String
               :rtype: Int or Float
               """
               if calc == 'area':
                   return np.pi*r**2
               elif calc == 'circumference':
                   return 2*np.pi*r

In [203]:  my_circ_calc(2.5, 'area')

Out[203]:  19.634954084936208

In [204]:  my_circ_calc(3, 'circumference')

Out[204]:  18.84955592153876

In [205]:  my_circ_calc(np.array([2, 3, 4]), 'circumference')

Out[205]:  array([12.56637061, 18.84955592, 25.13274123])
```

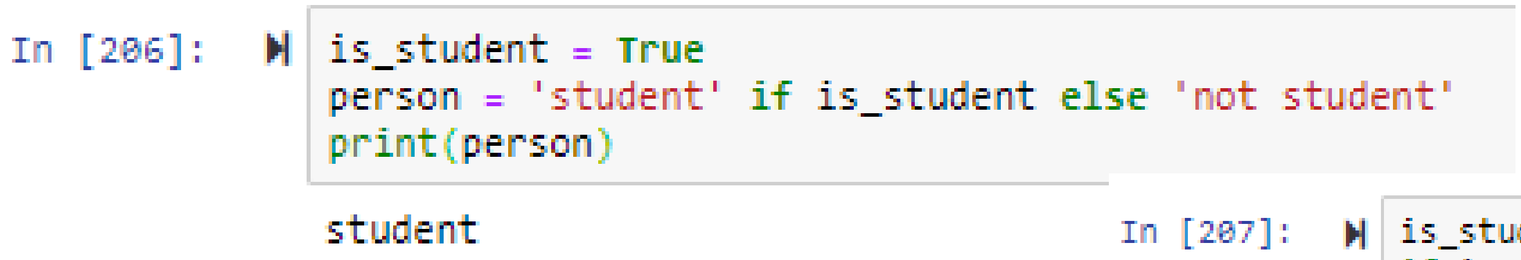


```
In [198]:  def is_odd(number):
               """
               function returns 'odd' if the input is odd,
                  'even' otherwise
               author
               date
               :type number: Int
               :rtype: String
               """
               # use modulo to check if the input is divisible by 2
               if number % 2 == 0:
                   # if it is divisible by 2, then input is not odd
                   return 'even'
               else:
                   return 'odd'

In [199]:  is_odd(11)

Out[199]:  'odd'

In [200]:  is_odd(2)

Out[200]:  'even'
```

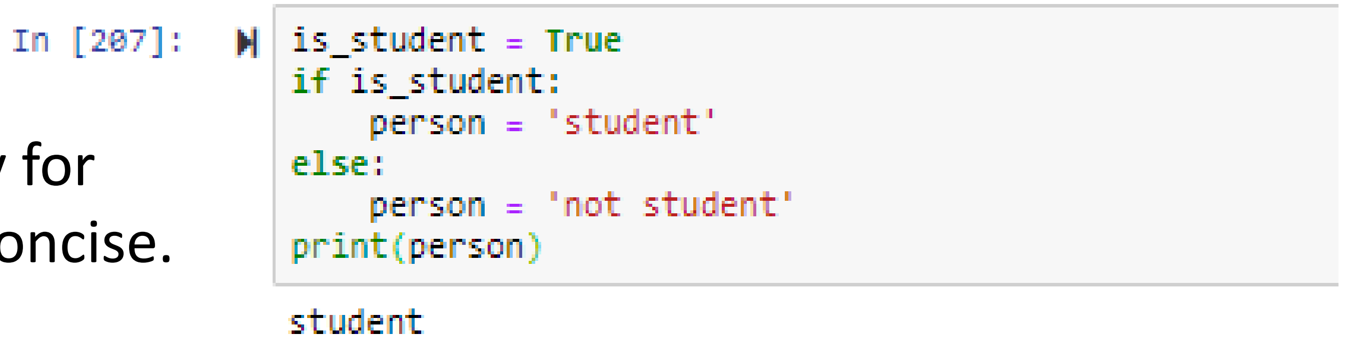
Ternary Operators

- Most programming languages have **ternary** operators, which usually known as conditional expressions. It provides a way that we can use **one-line** code to evaluate the first expression if the condition is true, otherwise it evaluates the second expression.
- Python has its way to implement the ternary operator, which can be constructed as below:

```
expression_if_true if condition else expression_if_false
```



```
In [206]:  is_student = True
           person = 'student' if is_student else 'not student'
           print(person)

           student
```



```
In [207]:  is_student = True
           if is_student:
               person = 'student'
           else:
               person = 'not student'
           print(person)

           student
```

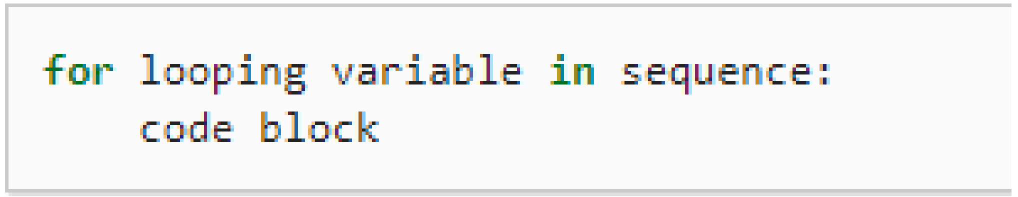
- Ternary operator provides a simple way for branching, and it can make our codes concise.

Iteration

- Many tasks in life are boring or tedious because they require doing the same basic actions over and over again—**iterating**—in slightly different contexts.
- For example, consider looking up the definition of 20 words in the dictionary, populating a large table of numbers with data, alphabetizing many stacks of paper, or dusting off every object and shelf in your room.
- Since repetitive tasks appear so frequently, it is only natural that programming languages like Python would have direct methods of performing iteration.
- With **branching** and **iteration**, it is possible to program just about any task that you can imagine.

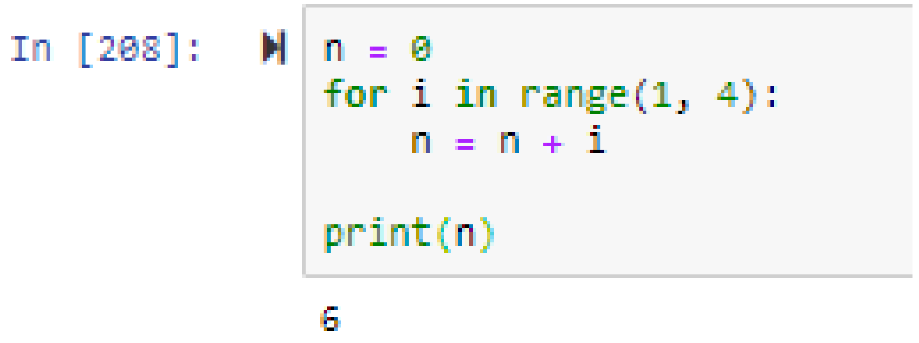
For Loops

- A **for-loop** is a set of instructions that is repeated, or iterated, for every value in a sequence. Sometimes for-loops are referred to as **definite** loops because they have a predefined begin and end as bounded by the sequence.
- The general syntax of a for-loop block is as follows.

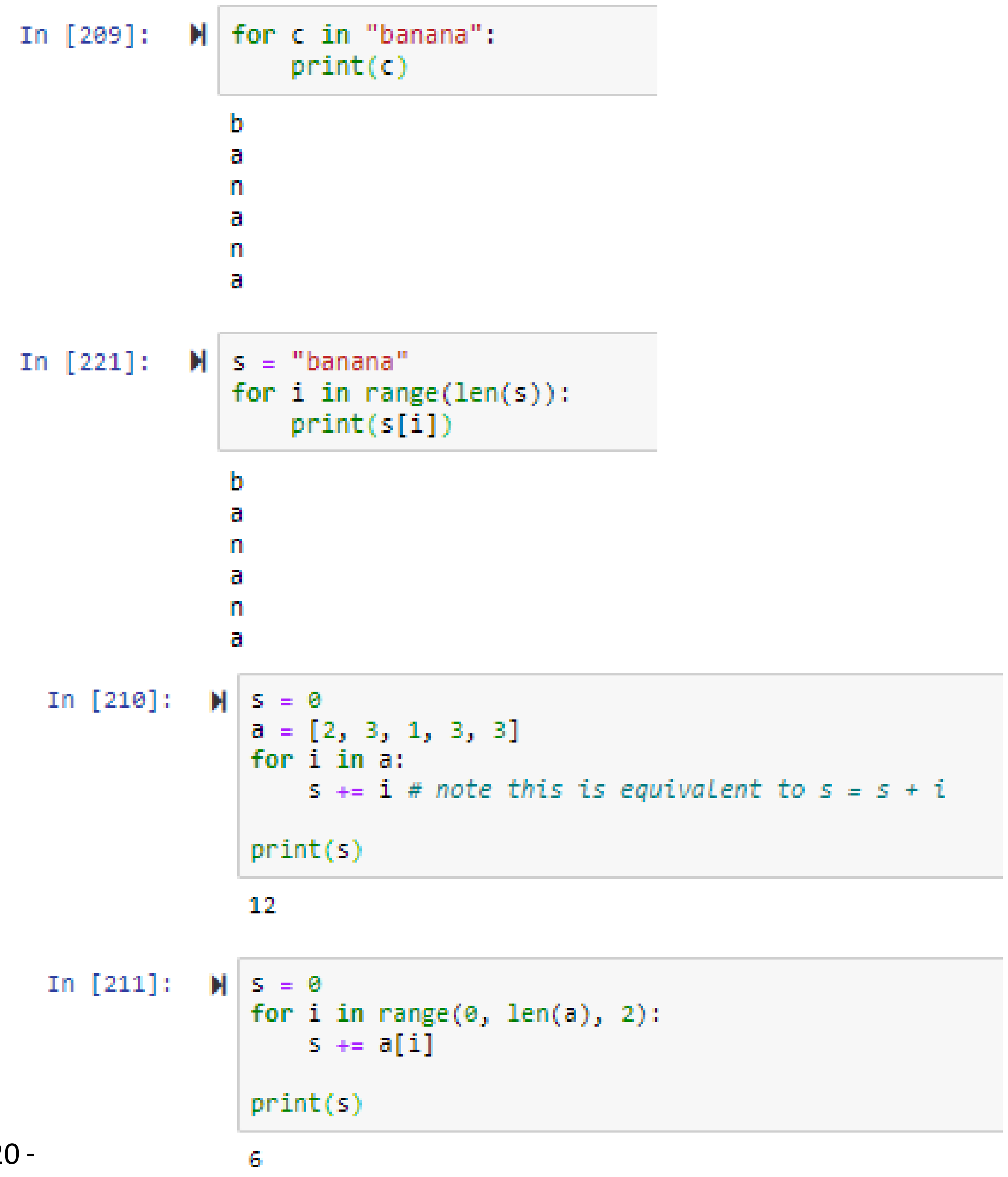


```
for looping variable in sequence:
    code block
```

- A for-loop assigns the looping variable to the **first** element of the sequence. It executes everything in the **code block**. Then it assigns the looping variable to the **next** element of the sequence and executes the code block again. It continues until there are no more elements in the sequence to assign.



```
In [208]:  n = 0
           for i in range(1, 4):
               n = n + i

           print(n)

           6
```

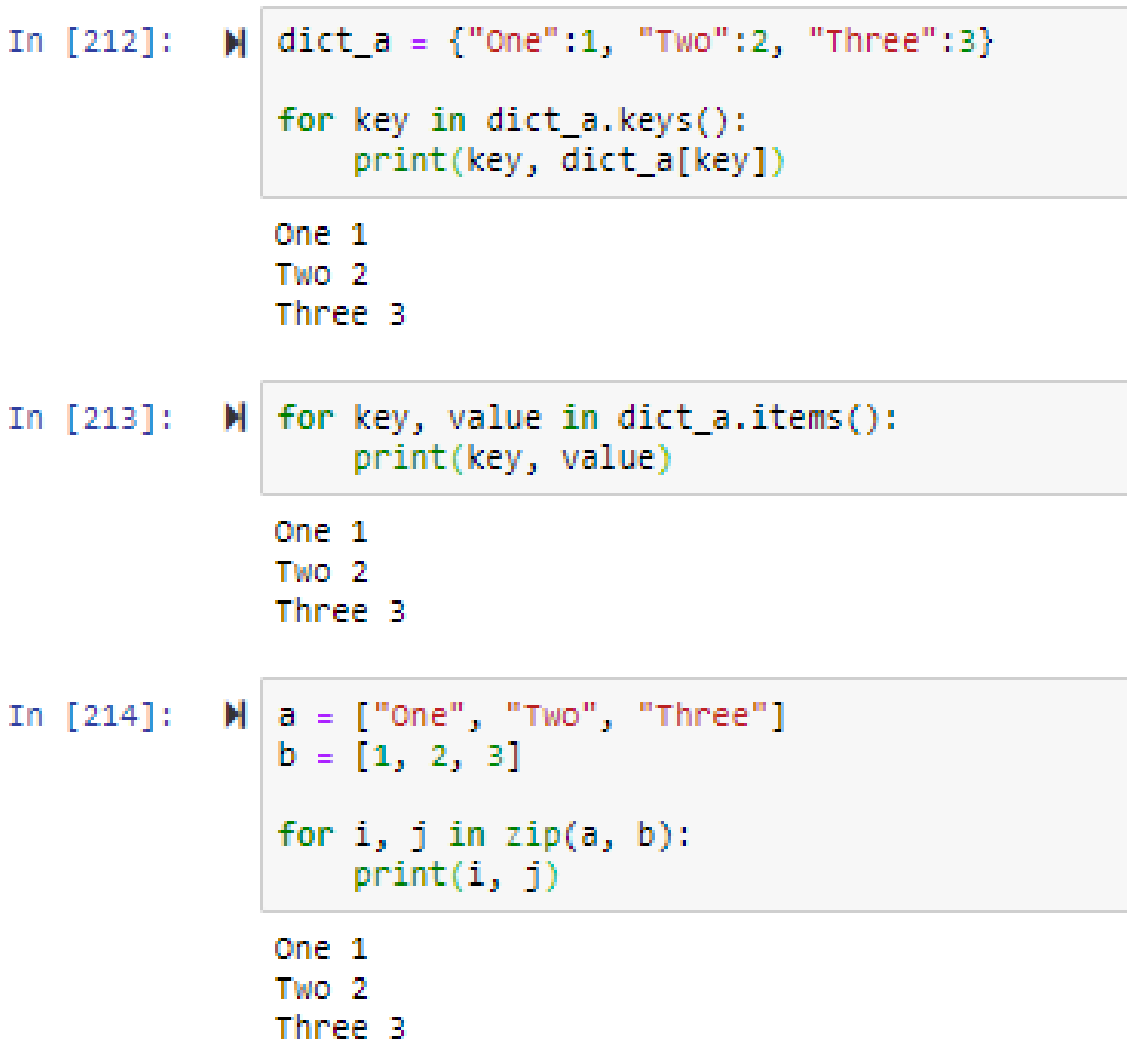


```
In [209]:  for c in "banana":
               print(c)

b
a
n
a
n
a
```

```
In [221]:  s = "banana"
           for i in range(len(s)):
               print(s[i])

b
a
n
a
n
a
```

```
In [210]:  s = 0
           a = [2, 3, 1, 3, 3]
           for i in a:
               s += i # note this is equivalent to s = s + i

           print(s)

12
```

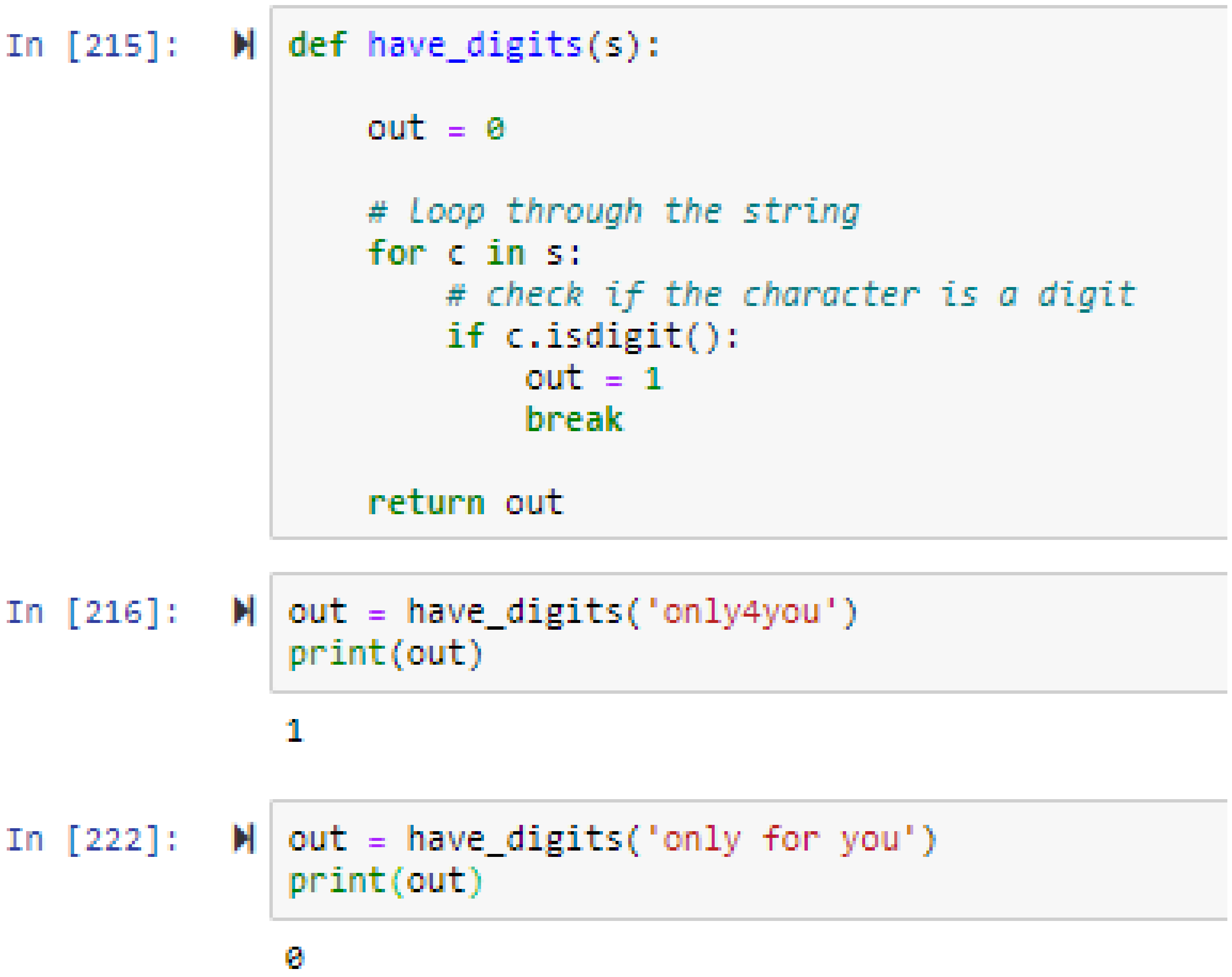
```
In [211]:  s = 0
           for i in range(0, len(a), 2):
               s += a[i]

           print(s)

6
```



```
In [212]:  dict_a = {"One":1, "Two":2, "Three":3}

           for key in dict_a.keys():
               print(key, dict_a[key])

One 1
Two 2
Three 3
```

```
In [213]:  for key, value in dict_a.items():
               print(key, value)

One 1
Two 2
Three 3
```

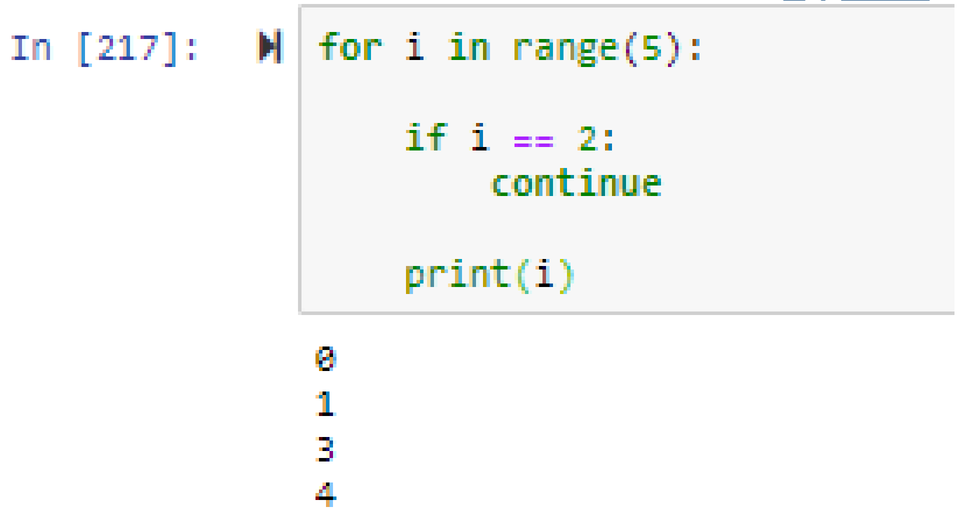
```
In [214]:  a = ["One", "Two", "Three"]
           b = [1, 2, 3]

           for i, j in zip(a, b):
               print(i, j)

One 1
Two 2
Three 3
```



```
In [215]:  def have_digits(s):

               out = 0

               # Loop through the string
               for c in s:
                   # check if the character is a digit
                   if c.isdigit():
                       out = 1
                       break

               return out
```

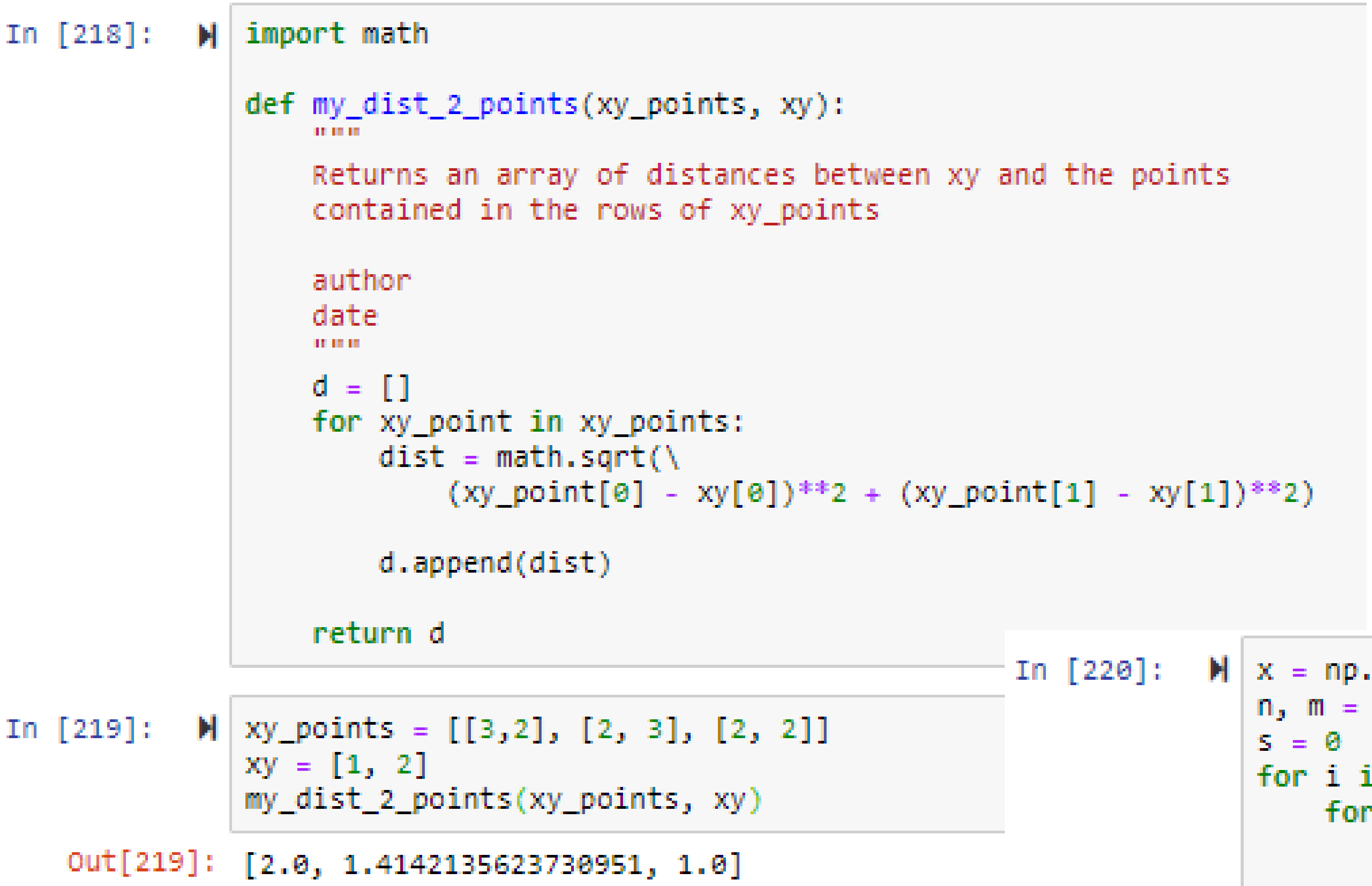
```
In [216]:  out = have_digits('only4you')
           print(out)

           1
```

```
In [222]:  out = have_digits('only for you')
           print(out)

           0
```



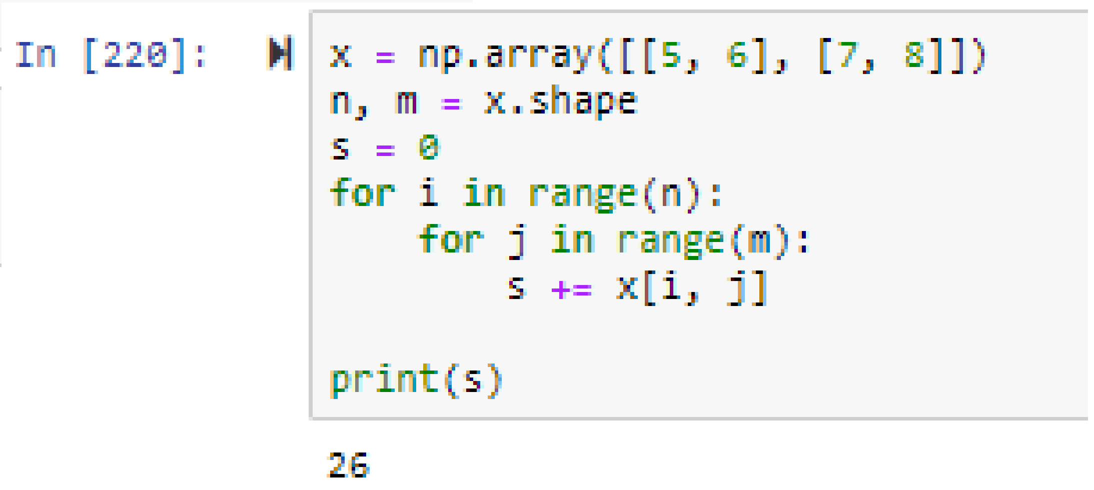
```
In [217]:  for i in range(5):

               if i == 2:
                   continue

               print(i)

           0
           1
           3
           4
```

- **Break** statements are used when anything happens in a for-loop that would make you want it to stop early. A less intrusive command is the keyword **continue**, which skips the remaining code in the current iteration of the for-loop, and continues on to the next element of the looping array.



```
In [218]:  import math

           def my_dist_2_points(xy_points, xy):
               """
               Returns an array of distances between xy and the points
               contained in the rows of xy_points

               author
               date
               """
               d = []
               for xy_point in xy_points:
                   dist = math.sqrt(\
                       (xy_point[0] - xy[0])**2 + (xy_point[1] - xy[1])**2)

                   d.append(dist)

               return d
```

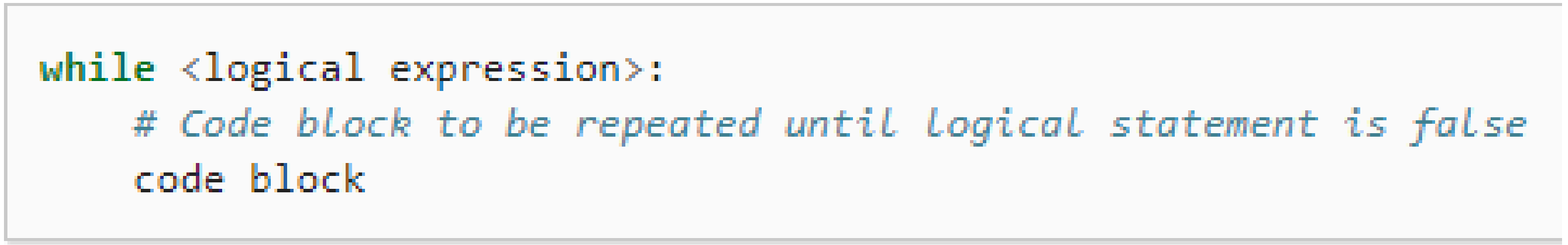
```
In [219]:  xy_points = [[3,2], [2, 3], [2, 2]]
           xy = [1, 2]
           my_dist_2_points(xy_points, xy)

Out[219]:  [2.0, 1.4142135623730951, 1.0]
```



```
In [220]:  x = np.array([[5, 6], [7, 8]])
           n, m = x.shape
           s = 0
           for i in range(n):
               for j in range(m):
                   s += x[i, j]

           print(s)

           26
```

While Loops

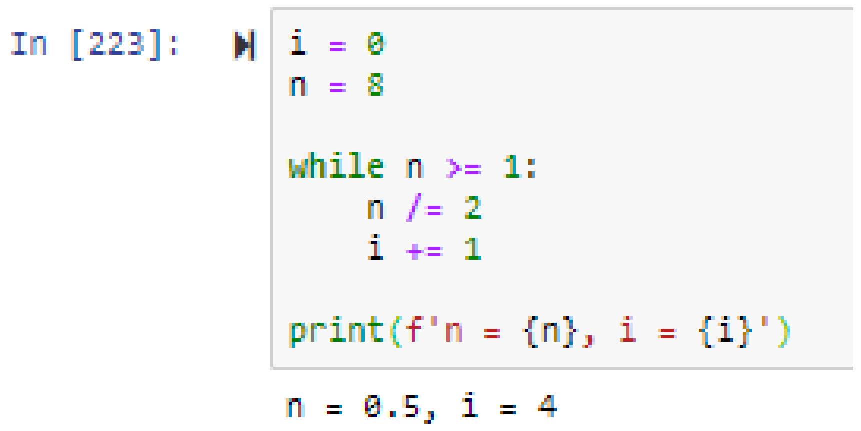
- A **while loop** or **indefinite** loop is a set of instructions that is repeated as long as the associated logical expression is true. The following is the abstract syntax of a while loop block.



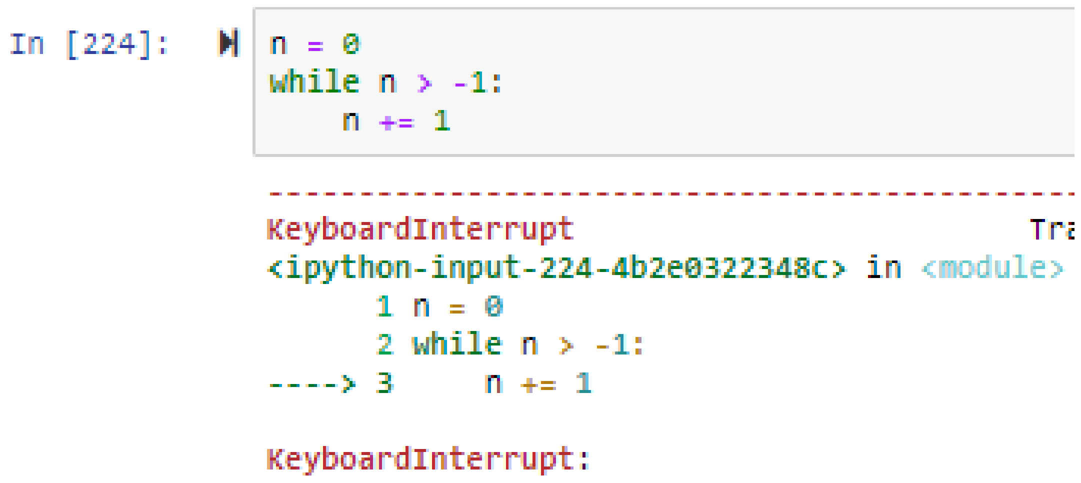
```
while <logical expression>:
    # Code block to be repeated until logical statement is false
    code block
```

- When Python reaches a while loop block, it first determines if the logical expression of the while loop is **true** or **false**. If the expression is true, the code block will be executed, and after it is executed, the program returns to the logical expression at the beginning of the while statement. If it is false, then the while loop will terminate.
- If the logical expression is true, and nothing in the while-loop code changes the expression, then the result is known as an infinite loop. Infinite loops run forever, or until your computer breaks or runs out of memory.

While Loops



```
In [223]:   i = 0
            n = 8

            while n >= 1:
                n /= 2
                i += 1

            print(f'n = {n}, i = {i}')

            n = 0.5, i = 4
```

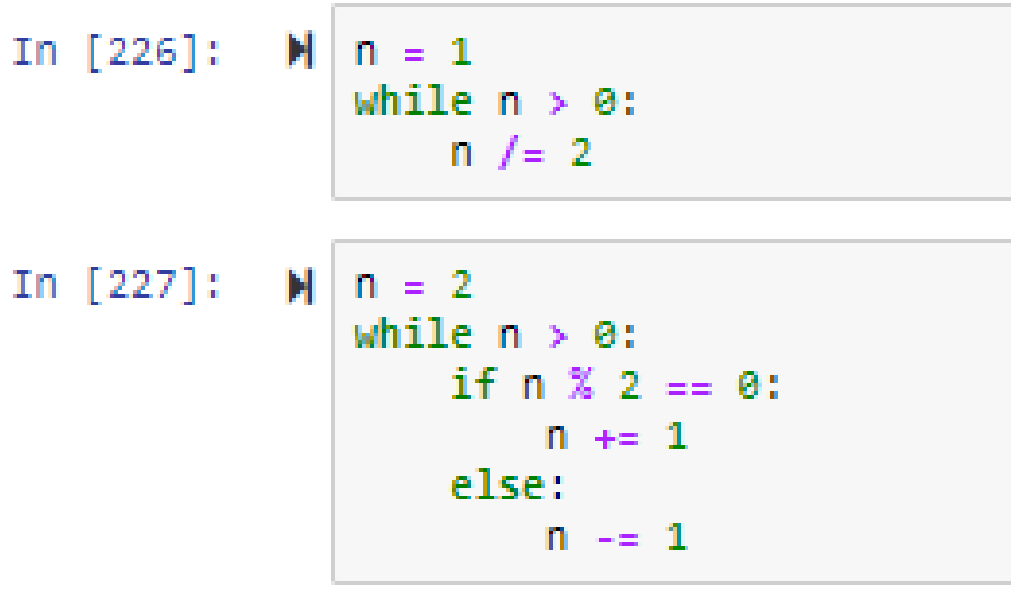


```
In [224]:   n = 0
            while n > -1:
                n += 1

            ---------------------------------------------
            KeyboardInterrupt                         Tra
            <ipython-input-224-4b2e0322348c> in <module>
                  1 n = 0
                  2 while n > -1:
            ----> 3     n += 1

            KeyboardInterrupt:
```

- You can terminate the infinite while loop manually by pressing the **interrupt** the kernel - the **black square button** in the tool bar above, or the drop down menu - **Kernel - Interrupt** in the notebook.
- Can you change a single character so that the while-loop will run at least once but will not infinite loop?

While Loops

- Infinite loops are not always easy to spot. Consider the next two examples: one infinite loops and one does not. Can you determine which is which?



```
In [226]:  n = 1
           while n > 0:
               n /= 2
```

```
In [227]:  n = 2
           while n > 0:
               if n % 2 == 0:
                   n += 1
               else:
                   n -= 1
```

- Now we know two types of loops: for-loops and while-loops.
- In some cases, either can be used equally well, but sometimes one is better suited for the task than the other.
- In general, we should use for-loops when the number of iterations to be performed is well-defined. Conversely, we should use while-loops statements when the number of iterations to be performed is indefinite or not well known.

Comprehensions

- In Python, there are other ways to do iterations, list (dictionary, set) **comprehensions** are an important and popular way that you will use frequently in your work.
- Comprehensions allow sequences to be created from other sequence with very compact syntax.

```
[Output Input_sequence Conditions]
```

```
In [228]:  x = range(5)
           y = []

           for i in x:
               y.append(i**2)
           print(y)

           [0, 1, 4, 9, 16]
```

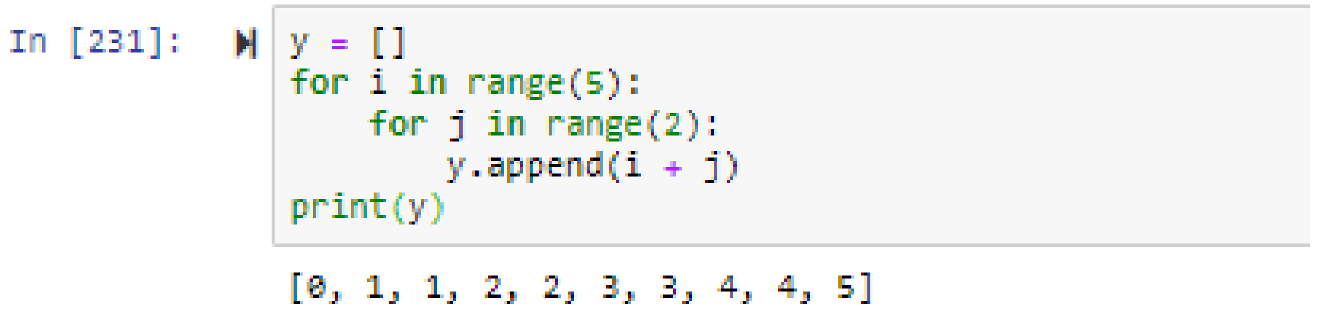
```
In [229]:  y = [i**2 for i in x]
           print(y)

           [0, 1, 4, 9, 16]
```

- We can have **conditions** in the list comprehension as well.

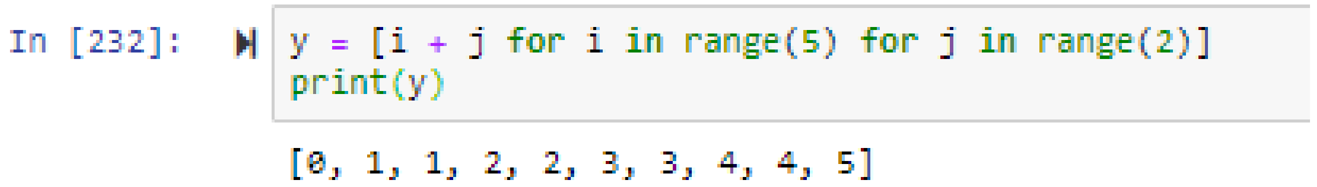
```
In [230]:  y = [i**2 for i in x if i%2 == 0]
           print(y)

           [0, 4, 16]
```

Comprehensions

- If we have two **nested** for loops, we can also use the list comprehensions as well.

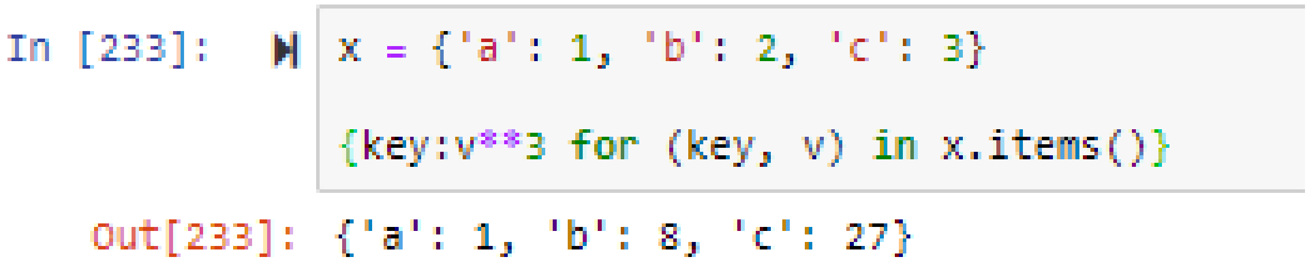


```
In [231]:  y = []
           for i in range(5):
               for j in range(2):
                   y.append(i + j)
           print(y)

           [0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
```



```
In [232]:  y = [i + j for i in range(5) for j in range(2)]
           print(y)

           [0, 1, 1, 2, 2, 3, 3, 4, 4, 5]
```

- Similarly, we can do the **dictionary comprehension** as well. See the following example.



```
In [233]:  x = {'a': 1, 'b': 2, 'c': 3}

           {key:v**3 for (key, v) in x.items()}

Out[233]: {'a': 1, 'b': 8, 'c': 27}
```

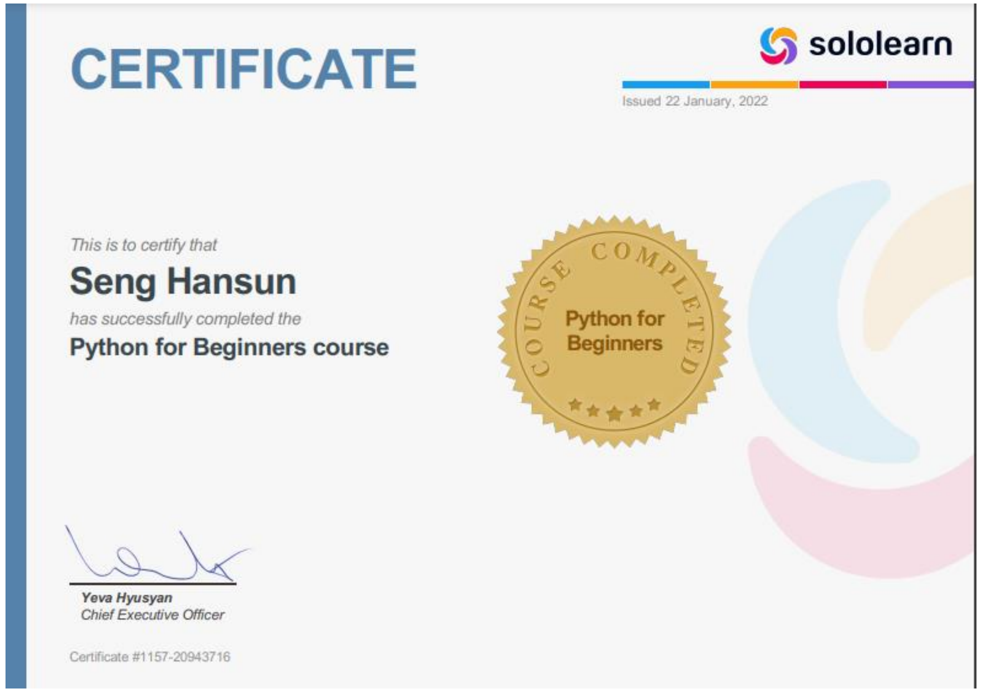
Assignments/ Problems

- Complete all lessons at Sololearn for the 'Python for Beginners' course, which can be accessed at

  https://www.sololearn.com/learning/1157

  Earn the certificate and submit it via UMN e-learning within three (3) weeks!

https://www.sololearn.com/Certificate/1157-8DIGITSID/pdf/



CERTIFICATE

sololearn

Issued 22 January, 2022

This is to certify that

Seng Hansun

has successfully completed the

Python for Beginners course

COURSE COMPLETED
Python for Beginners

Yeva Hyusyan
Chief Executive Officer

Certificate #1157-20943716

Next Week's Outline

- Recursion
- Object Oriented Programming (OOP)
- Complexity
- Representation of Numbers
- Errors, Good Programming Practices, and Debugging

References

- Kong, Qingkai; Siauw, Timmy, and Bayen, Alexandre. 2020. Python Programming and Numerical Methods: A Guide for Engineers and Scientists. Academic Press. https://www.elsevier.com/books/python-programming-and-numerical-methods/kong/978-0-12-819549-9
- Other online and offline references

Visi

Menjadi Program Studi Strata Satu Informatika **unggulan** yang menghasilkan lulusan **berwawasan internasional** yang **kompeten** di bidang Ilmu Komputer (*Computer Science*), **berjiwa wirausaha** dan **berbudi pekerti luhur**.

Misi

1. Menyelenggarakan pembelajaran dengan teknologi dan kurikulum terbaik serta didukung tenaga pengajar profesional.
2. Melaksanakan kegiatan penelitian di bidang Informatika untuk memajukan ilmu dan teknologi Informatika.
3. Melaksanakan kegiatan pengabdian kepada masyarakat berbasis ilmu dan teknologi Informatika dalam rangka mengamalkan ilmu dan teknologi Informatika.