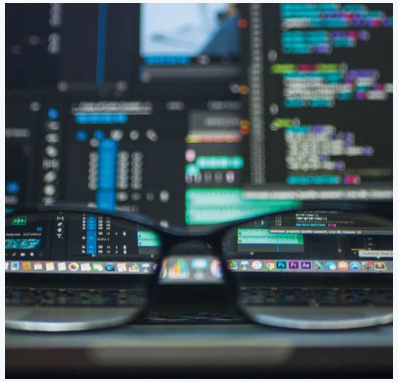
Certificate in Introductory Data Analytics

Fundamentals of Programming

Unit 2



Overview

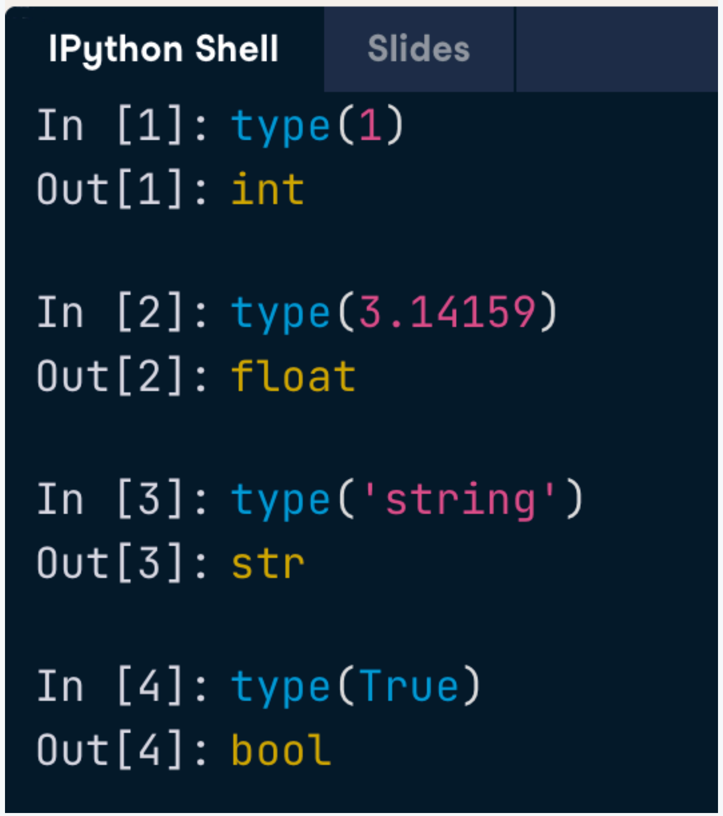
- Data Types
- Operations
- Collection Data Types
  - Tuples
  - Lists
  - Dictionaries
- Conditional Statements
  - If
  - While
  - For
- Functions
- Python Packages

In This Unit

UCD DUBLIN PROFESSIONAL ACADEMY

Four Main Data Types

- Integers (`int`)
  - Whole numbers
  - `1, 2, 3`
- Floats (`float`)
  - Floating-point numbers, numbers with decimals
  - `3.14159`
- Strings (`str`)
  - Sequence of characters
  - `‘string’, ‘this is also a string’`
- Booleans (`bool`)
  - Only one of two values: `True` or `False`



```
In [1]: type(1)
Out[1]: int

In [2]: type(3.14159)
Out[2]: float

In [3]: type('string')
Out[3]: str

In [4]: type(True)
Out[4]: bool
```

Use the function type() to find out the data type

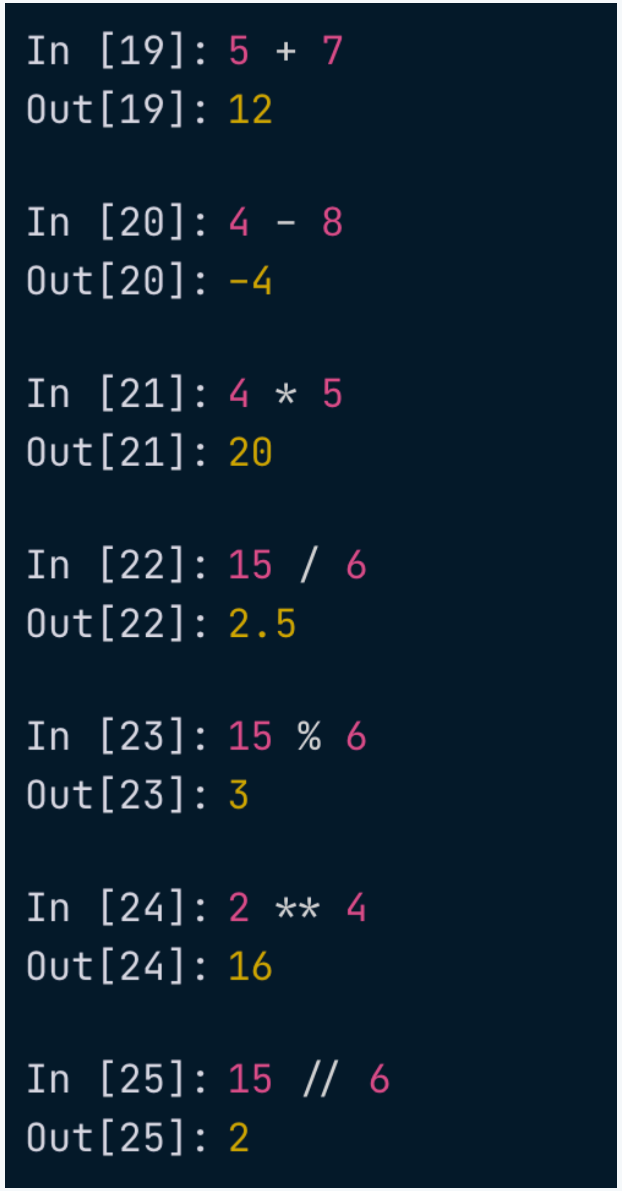
Note: You can make collections of these data types to form ‘collection data types’. We will get to these later

Operations

Arithmetic Operators

- Addition (+)
- Subtraction (-)
- Multiplication (*)
- Division (/)
- Modulus (%)
- Exponent (**)
- Floor Division (//)

Note: You can add also add strings together!



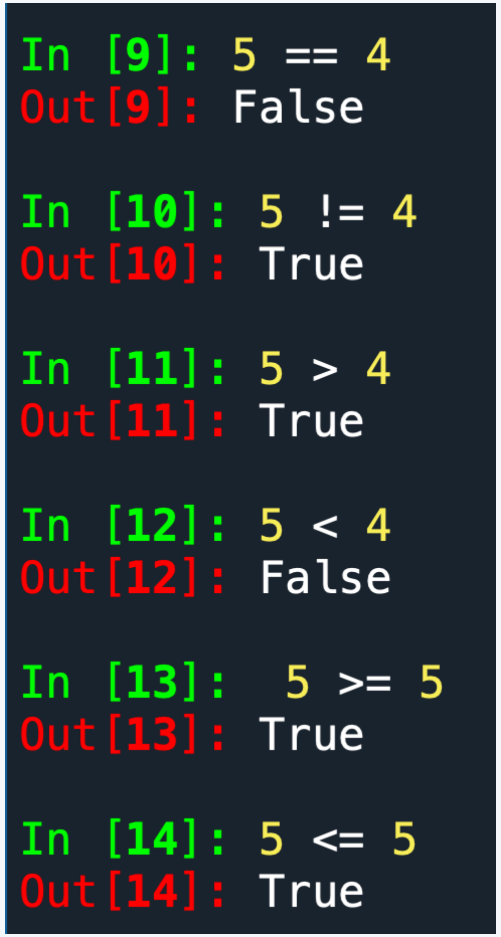
```
In [19]: 5 + 7
Out[19]: 12

In [20]: 4 - 8
Out[20]: -4

In [21]: 4 * 5
Out[21]: 20

In [22]: 15 / 6
Out[22]: 2.5

In [23]: 15 % 6
Out[23]: 3

In [24]: 2 ** 4
Out[24]: 16

In [25]: 15 // 6
Out[25]: 2
```

Operations

Comparison Operators – Returns Boolean (**True** or **False**)

- Equal to (==)
- Not equal to (!=)
- Greater than (>)
- Less than (<)
- Great than or equal to (>=)
- Less than or equal to (<=)

Note: `You can even compare strings!`



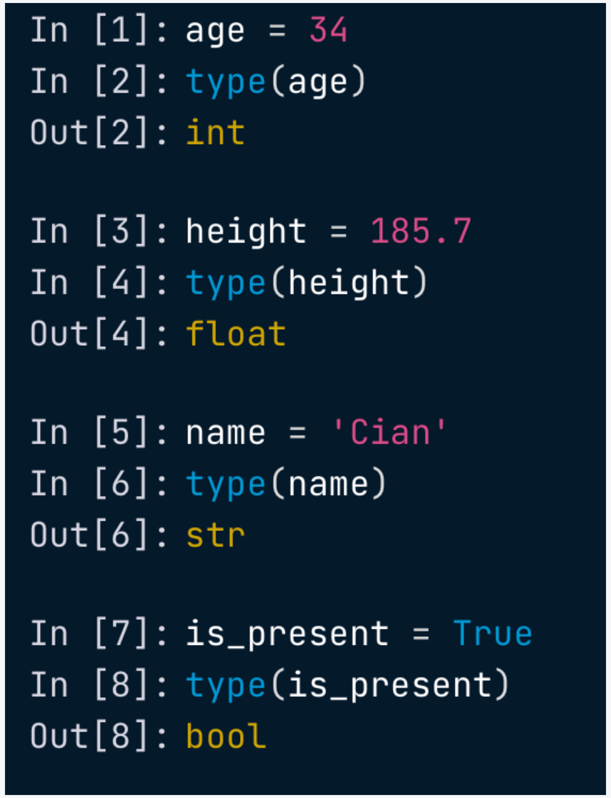
```
In [9]: 5 == 4
Out[9]: False

In [10]: 5 != 4
Out[10]: True

In [11]: 5 > 4
Out[11]: True

In [12]: 5 < 4
Out[12]: False

In [13]:  5 >= 5
Out[13]: True

In [14]: 5 <= 5
Out[14]: True
```

Variables

Assigning Variables

- Variables can be created from these four data types.
- We assign values to variables using the = operator
- Python will automatically know the which variable type to use, depending on the data type assigned to it.

  - Whole numbers will be set to int
  - Numbers with decimals will be set to float
  - Anything within ‘single’ or “double” quotes will be a str
  - True or False will be bool



```
In [1]: age = 34
In [2]: type(age)
Out[2]: int

In [3]: height = 185.7
In [4]: type(height)
Out[4]: float

In [5]: name = 'Cian'
In [6]: type(name)
Out[6]: str

In [7]: is_present = True
In [8]: type(is_present)
Out[8]: bool
```

You can use the function **`type()`** to find out the data type of a variable too.

Assignment vs. Comparison

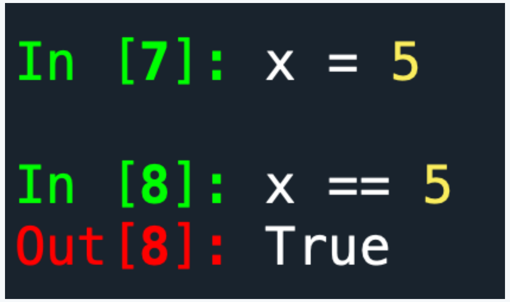
Are you asking, or are you telling?

Set x equal to 5. (x = 5)

- No output

Is x equal to 5? (x == 5)

- True or False



```
In [7]: x = 5

In [8]: x == 5
Out[8]: True
```

Collection Data Types

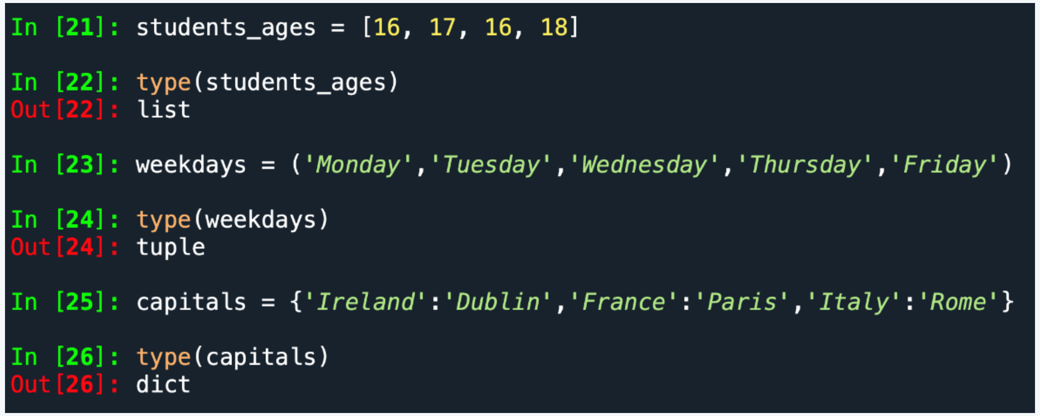
The previous data types can be collected together into collection data types.

Sequence Type

- List []
  - Ordered sequence
  - Able to change values (mutable)
- Tuple ()
  - Similar to list but cannot change values (immutable)
  - Must all be of similar data type

Mapping Type

- Dictionary {}
  - Not ordered.
  - Instead, uses unique keys to index the values

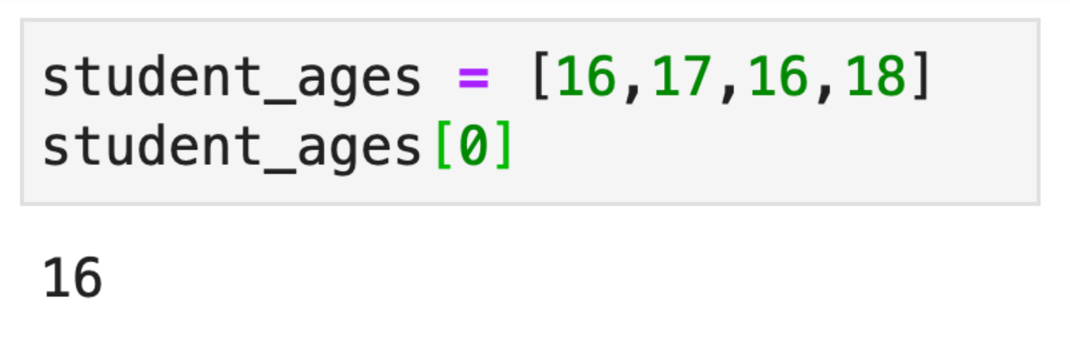


```
In [21]: students_ages = [16, 17, 16, 18]

In [22]: type(students_ages)
Out[22]: list

In [23]: weekdays = ('Monday','Tuesday','Wednesday','Thursday','Friday')

In [24]: type(weekdays)
Out[24]: tuple

In [25]: capitals = {'Ireland':'Dublin','France':'Paris','Italy':'Rome'}

In [26]: type(capitals)
Out[26]: dict
```

Note: Technically, a string is also a sequence data type, as it is a sequence of characters.

Lists

- Sequence of values, of any data type, defined by use of [square brackets]
- Values may be added, removed, changed or appended
- Since the values are ordered, they can be accessed using their index number (starting at 0) and square bracket notation.

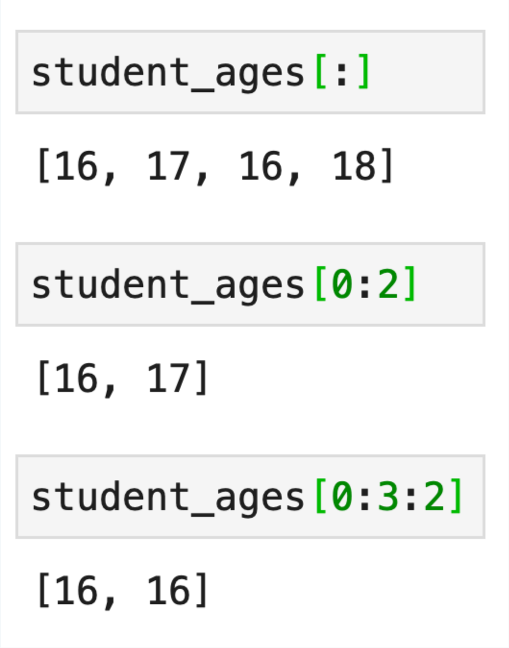


```
student_ages = [16,17,16,18]
student_ages[0]
```

```
16
```

Lists – Slicing

- To subset a list (slice), you can pass in up to three parameters
  - [start : end : step size]
- Not all three are necessary.
  - If you omit the first value, it will start at the start
  - If you omit the second value, it will run to the end
  - If you omit the step size, it will assume a value of 1 (so it will step through every value)



```
student_ages[:]
```

```
[16, 17, 16, 18]
```

```
student_ages[0:2]
```

```
[16, 17]
```

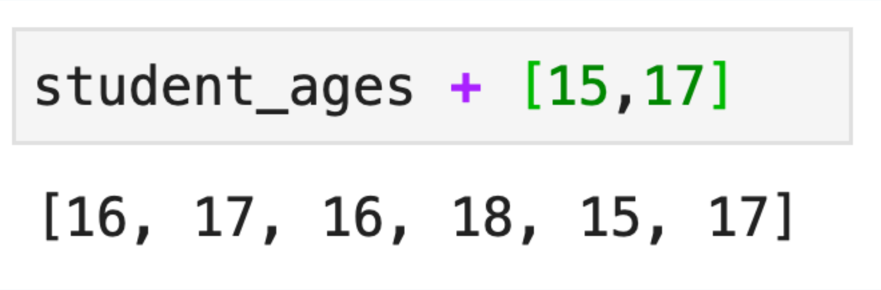
```
student_ages[0:3:2]
```

```
[16, 16]
```

`Note: The start value will be included in the slice, but the end value will not.`

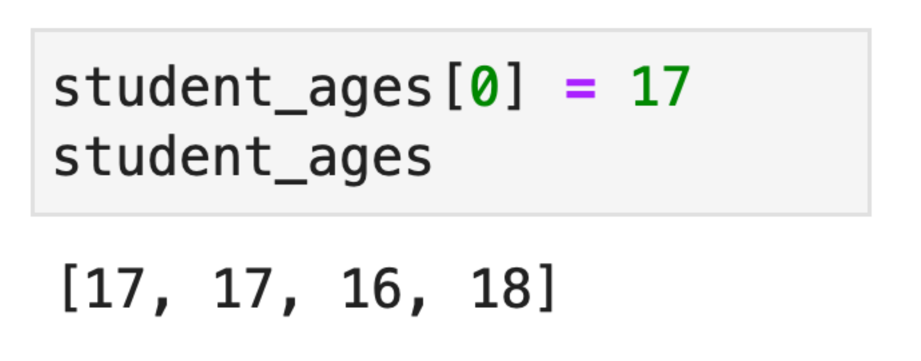
Lists – Manipulating

- Values can be added on to a list using the + operator
- Values within a list can be reassigned just as regular variables would
- Slices of lists can similarly be reassigned by providing a list of similar length



```
student_ages + [15,17]
```

```
[16, 17, 16, 18, 15, 17]
```



```
student_ages[0] = 17
student_ages
```

```
[17, 17, 16, 18]
```

Note: Adding the values to the list, returned a list with the additional values appended. It did not add them to the original list. To do this you will need the following code:

```
student_ages = student_ages + [15,17]
```

Lists – Copying

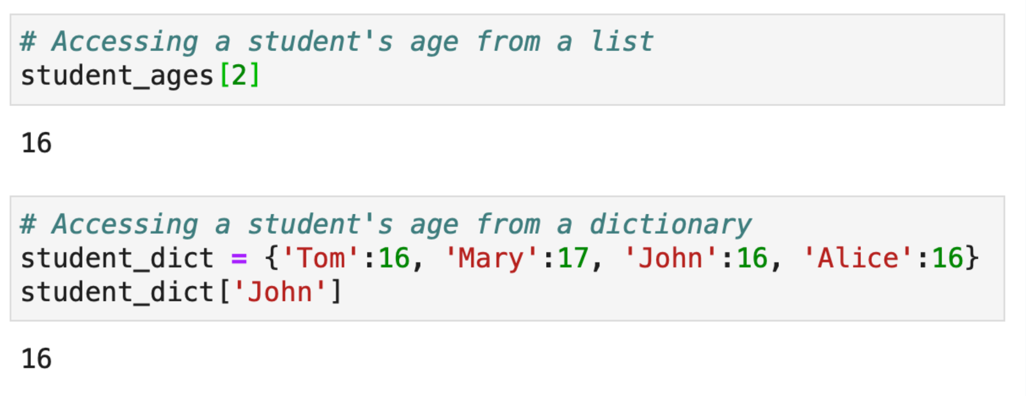
- Assigning a list to another variable does not create a new object. It just creates a new pointer to the same list
- Handiest way to create a new list is to create a slice of the old list
- You can use `list()` function to create a new list, passing the old one as a parameter
- There is a built-in method in Python 3.3:
  - `new_list = old_list.copy()`

Note: These methods will not create copies of any lists within the list. To do this, use deepcopy from the copy module:

```
import copy
new_list = copy.deepcopy(old_list)
```

Dictionaries

- When accessing values from a list, we needed to know the index number.
- In some instances, it might be more conv access values using a unique key.
- Dictionaries store data in key-value pairs
- They are defined using {curly brackets} a key:values
- Delete items with `del()` function

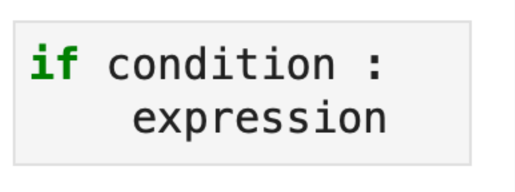


```
# Accessing a student's age from a list
student_ages[2]

16

# Accessing a student's age from a dictionary
student_dict = {'Tom':16, 'Mary':17, 'John':16, 'Alice':16}
student_dict['John']

16
```

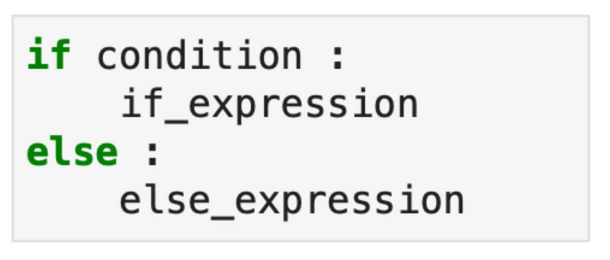
`**Note:** The key must be unique. In the presented example, we could only have one John in the dictionary.`

Conditional Statements: If-then-else

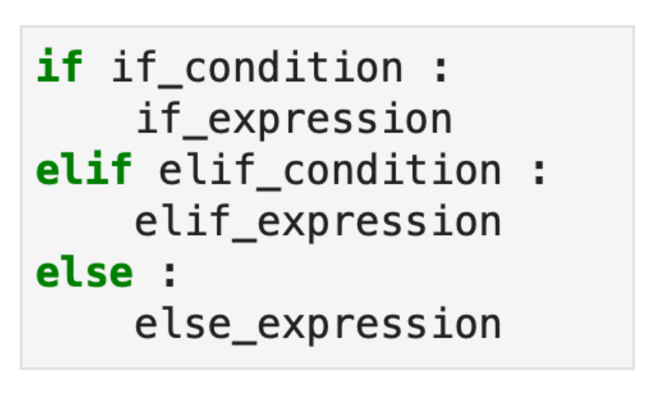
- The condition is Boolean.
- If the condition is True, the `if` expression is executed
- We can also add in a second expression, the else expression, to be executed whenever the `if` condition is `False`
- We can also add else if conditions, written as `elif`,
- In fact, there are no limit to the `elif` conditions we can add



```
if condition :
    expression
```



```
if condition :
    if_expression
else :
    else_expression
```

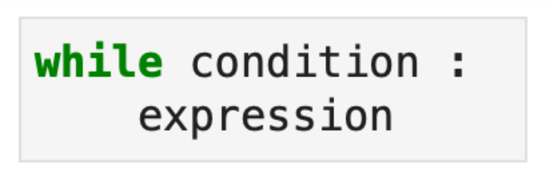


```
if if_condition :
    if_expression
elif elif_condition :
    elif_expression
else :
    else_expression
```

```
NOTE: As soon as one of the conditions is met, the associated expression is executed.
So bear in mind, the order of the expressions presented matters.
```

Conditional Statements: While

- This is our first loop.
- The condition is again **Boolean.**
- If the condition is **True**, the expression is executed.
- The difference here, is that the expression is repeated until the condition becomes **False**
- This is a form of **indefinite iteration**, since the number of iterations is not explicitly stated in advance.

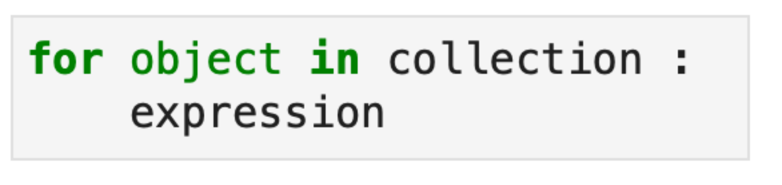


```
while condition :
    expression
```

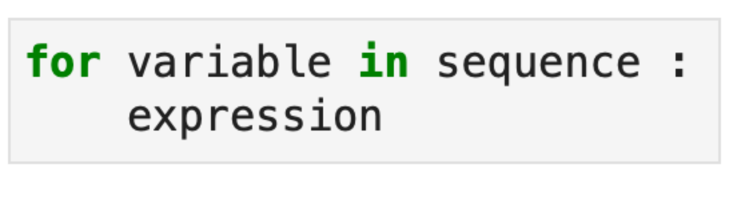
```
NOTE: Beware, you now have the ability to become trapped in an infinite loop!
Make sure your expression is doing something towards turning the condition False.
```

Conditional Statements: For

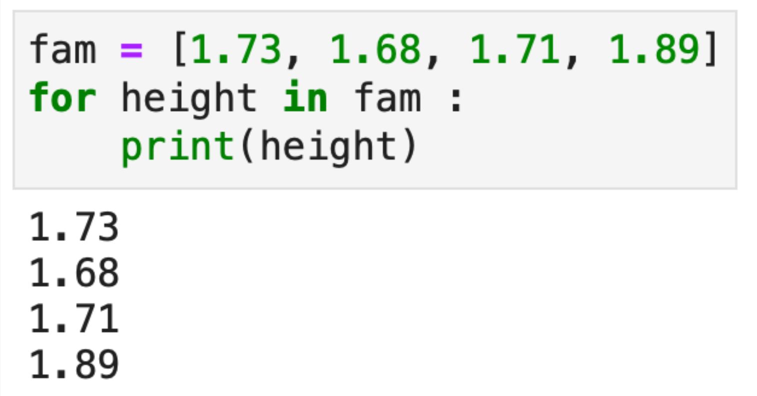
- A for loop is iterated over a collection of objects—known as an **iterable**.
- In contrast with the `while` loop, a for loop is a form of **definite iteration**. The number of iterations is explicitly stated in advance.
- The provided sequence or object collection can be a `string`, `array`, `list`, `dictionary`, `DataFrame`, `etc.`



```
for object in collection :
    expression
```



```
for variable in sequence :
    expression
```



```
fam = [1.73, 1.68, 1.71, 1.89]
for height in fam :
    print(height)
```

```
1.73
1.68
1.71
1.89
```

NOTE: Since the length of a the sequence is predefined, the loop is finite.

Functions

- Packaged piece of code to perform a particular task
- User can pass in one or more arguments that the function can use to produce its output.

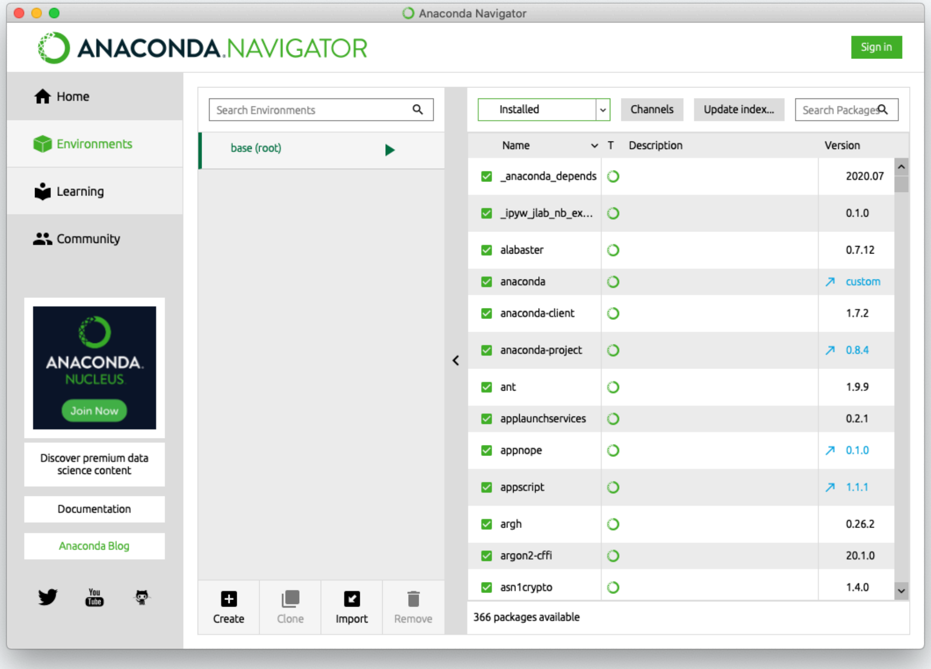
```
def square(a):
    a_sqrd = a*a
    return a_sqrd

print(square(3))
>> 9
```

Python Packages

- Now we know how to package our code as functions, your can take advantage of other pieces of code packaged the same way
- Downloading code is called **installing packages**
- Any external code that your code relies on is called a **dependency**
- This is also called **managing dependencies**
- Any piece of script imported is called a **module**
- A collection (folder) of modules is called a **package**.
  - A **function** is a collection of statements
  - A **module** is a collection of functions
  - A **package** is a collection of modules
  - A **library** is a collection of packages

Python Packages

- Open **Anaconda Navigator** and go to **Environments** on the left sidebar.
- Here, you can see all the pre-installed packages that are ready to be imported.
- You can also search for any additional packages and install them.

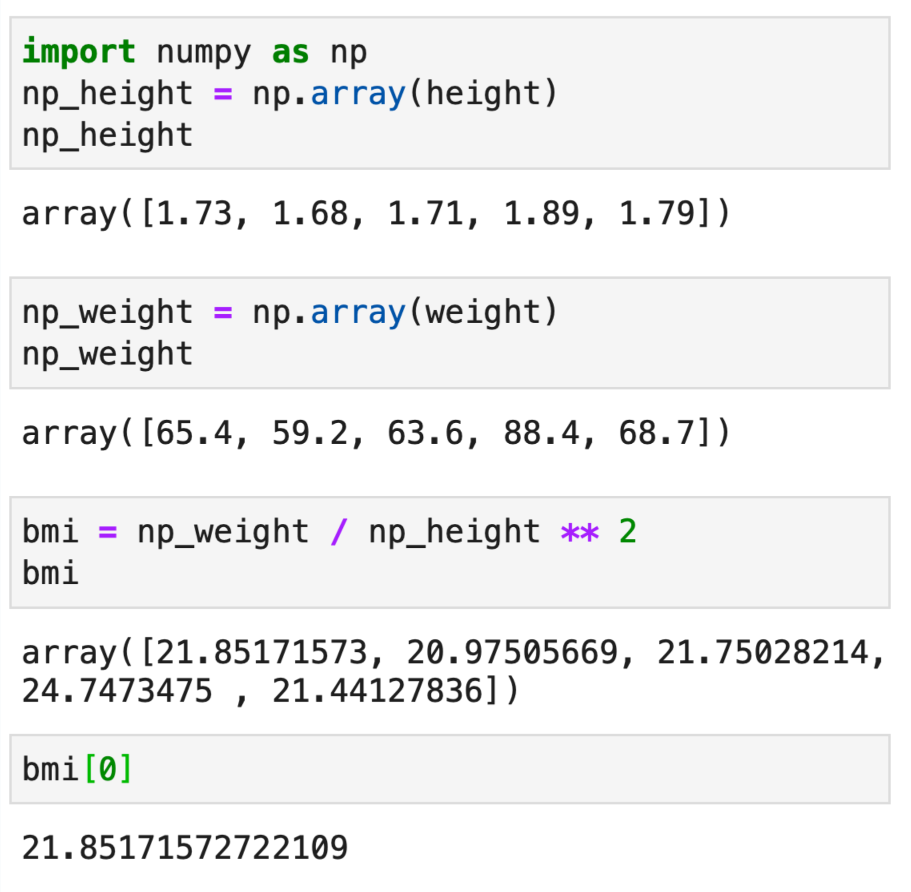


	Name	T	Description	Version
☑	_anaconda_depends	○		2020.07
☑	_ipyw_jlab_nb_ex...	○		0.1.0
☑	alabaster	○		0.7.12
☑	anaconda	○		↗ custom
☑	anaconda-client	○		1.7.2
☑	anaconda-project	○		↗ 0.8.4
☑	ant	○		1.9.9
☑	applaunchservices	○		0.2.1
☑	appnope	○		↗ 0.1.0
☑	appscript	○		↗ 1.1.1
☑	argh	○		0.26.2
☑	argon2-cffi	○		20.1.0
☑	asn1crypto	○		1.4.0

366 packages available

NumPy

- NumPy (Numeric Python) is a mathematical library for Python.
- Allows for convenient operations across tabular data (something we could not do with lists)
- For this to be possible, all data types in a NumPy array must be the same data type
- Square bracket notation still works for NumPy arrays, just like lists.



```
import numpy as np
np_height = np.array(height)
np_height
```

```
array([1.73, 1.68, 1.71, 1.89, 1.79])
```

```
np_weight = np.array(weight)
np_weight
```

```
array([65.4, 59.2, 63.6, 88.4, 68.7])
```

```
bmi = np_weight / np_height ** 2
bmi
```

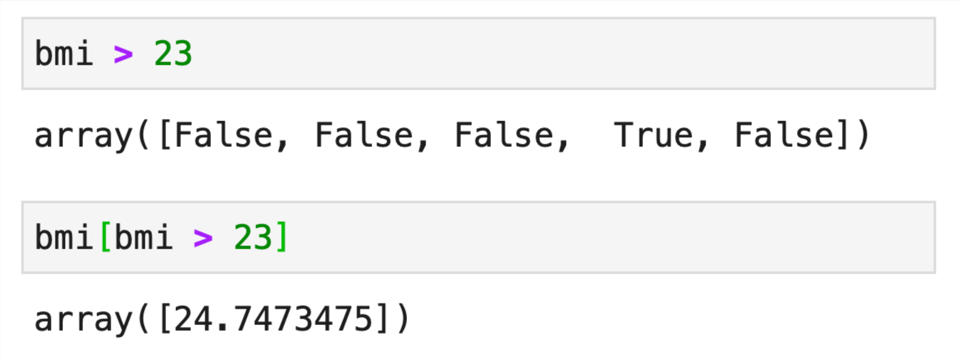
```
array([21.85171573, 20.97505669, 21.75028214,
24.7473475 , 21.44127836])
```

```
bmi[0]
```

```
21.85171572722109
```

NumPy – Subsetting

- This is a way of filtering a NumPy array for values that satisfy a particular criteria
- Like arithmetic operators, we can use comparison operators with NumPy arrays. This returns an array of Boolean data types (True or False)
- Passing this Boolean array back into the array, will return an array for only the True values



```
bmi > 23
```

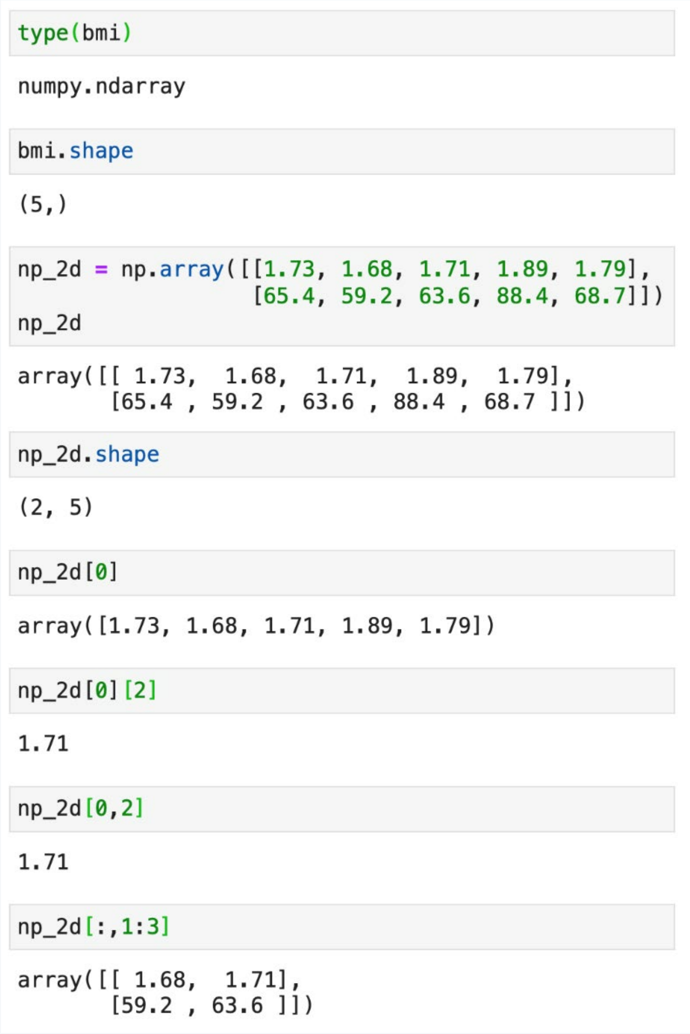
```
array([False, False, False,  True, False])
```

```
bmi[bmi > 23]
```

```
array([24.7473475])
```

NumPy – 2D Array

- So far we have been dealing with one-dimensional arrays.
- Just like we made lists of lists, we can make arrays of arrays.
- Accessing elements in the 2D array is just like with lists
  - `np_2d[0][2]`
- This can be reformatted to the below
  - `np_2d[0,2]`
- All the same slicing rules apply
  - `np_2d[:,1:3]`



```
type(bmi)
```

```
numpy.ndarray
```

```
bmi.shape
```

```
(5,)
```

```
np_2d = np.array([[1.73, 1.68, 1.71, 1.89, 1.79],
                  [65.4, 59.2, 63.6, 88.4, 68.7]])
np_2d
```

```
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],
       [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ]])
```

```
np_2d.shape
```

```
(2, 5)
```

```
np_2d[0]
```

```
array([1.73, 1.68, 1.71, 1.89, 1.79])
```

```
np_2d[0][2]
```

```
1.71
```

```
np_2d[0,2]
```

```
1.71
```

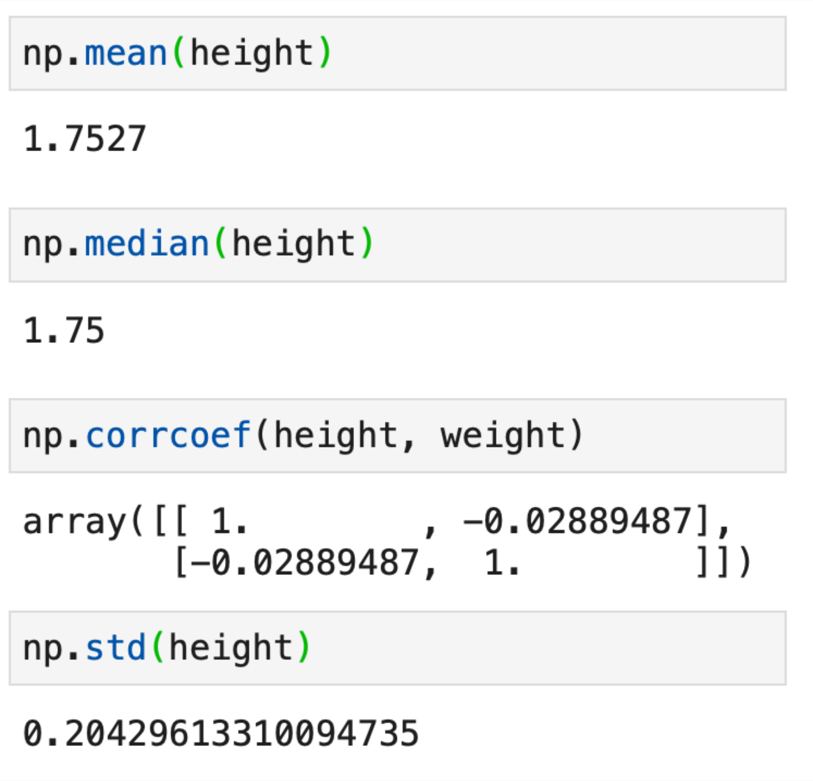
```
np_2d[:,1:3]
```

```
array([[ 1.68,  1.71],
       [59.2 , 63.6 ]])
```

NumPy – Statistics

- We can also use the NumPy library to perform some statistical analysis of our datasets

Mean, median, mode, standard deviation, correlation, etc.



```
np.mean(height)
```

```
1.7527
```

```
np.median(height)
```

```
1.75
```

```
np.corrcoef(height, weight)
```

```
array([[ 1.        , -0.02889487],
       [-0.02889487,  1.        ]])
```

```
np.std(height)
```

```
0.20429613310094735
```

Object-Oriented Programming

- Different styles or philosophies of programming are called **programming paradigms**.
- A particular programming paradigm we will be using is called ”**object-oriented programming**”.
  - In this paradigm, “**objects**” contain data and functions associated with them.
- Data within an object are called **attributes**; functions within an object are called **methods**.

"Find out *exactly* how many ways there are to skin a cat."

Methods & Attributes

- **Methods** are functions associated with an object.
- **Attributes** are data associated with an object

Objects	Methods	Attributes
`str`	`capitalize(), replace()`	-
`float`	`bit_length(), conjugate()`	`real, imag`
`list`	`index(), count()`	-
`dict`	`keys(), values()`	-
`ndarray`	`copy(), mean()`	`size, shape`
`DataFrame`	`head(), info(), describe()`	`index, columns`

Data Sources

Google Dataset Search

Kaggle

Dataquest

Datahub.io

Resources

Data Types:

https://colab.research.google.com/drive/1rYZjYlhEL6GcqZ7Xjg7-WPDAMcuL9T_7?usp=sharing