جامعة بيرزيت
BIRZEIT UNIVERSITY

Data Science and Analytics
Comp 4381

Ch2: Introduction to Python

References

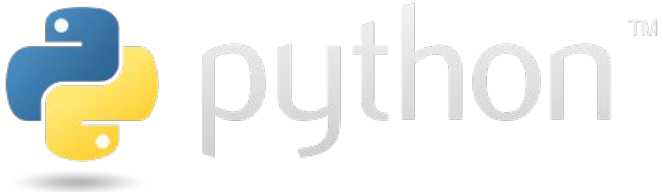
- **Books**:
  - Python for Data Analysis 3rd edition - Wes McKinney – O'RIELLY (Ch 2-10)
  - Python data science handbook 2nd edition - Jake VanderPlas – O'RIELLY (Ch 37-40)
  - Statistics unplugged 4th edition – Sally Cardwell - Wadsworth: (Ch 1, 2)
- **Material & Notebooks**:
  - Mr. Hussein Soboh.
- **Additional Resources:**
  - Computational and Inferential Thinking: The Foundations of Data Science 2nd Edition by Ani Adhikari, John DeNero, David Wagner. Link
  - https://www.w3schools.com/python

Introduction to Python 1

Python

Python is a high-level, flexible programming language known for its simplicity and readability.

- It supports multiple programming paradigms, including procedural, object-oriented, and functional programming.
- Python is widely used in web development, **data science**, automation, **artificial intelligence**, and scientific computing due to its extensive libraries and frameworks.
- Clear syntax makes it beginner-friendly and suitable for rapid development.



Why Python for data science?

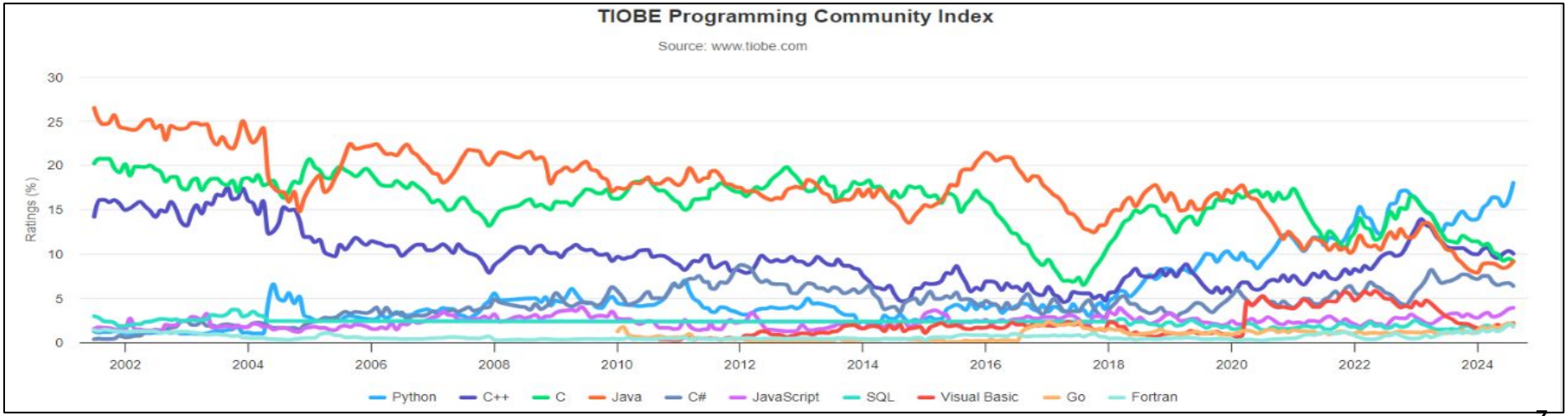
- **Simple and easy to learn**
  - Python syntax is clear and resembles everyday language
  - Python uses straightforward commands and readable structures, which helps new learners focus on problem-solving rather than getting stuck on complex syntax
  - Its design emphasizes code readability, allowing developers to write clean and concise code
- **Extensive libraries and frameworks**
  - Python is known for its extensive libraries and frameworks, which are collections of pre-built tools that make coding faster and easier
  - For data science, libraries like **Pandas** (for data manipulation), **NumPy** (for numerical operations), **Matplotlib and Seaborn** (for data visualization), and **Scikit-learn** (for machine learning). These libraries save time by providing ready-made functions to handle common tasks, so you don't have to write code from scratch

Why Python for data science?

- **Open source**
  - Anyone can use it for free
  - Allows people to not only use Python but also to improve it and share their changes
  - Large community of users who help each other, develop new tools, and provide support
- **Large community and support**
  - **Strong Community Support:** Developers and data scientists worldwide contribute to Python's growth
  - **Learning Resources:** Tutorials, forums, and open-source projects help beginners and professionals
  - **Problem-Solving:** Extensive documentation and community support make debugging and development easier
  - **Popularity in Data Science:** Python's support network enhances its use in analytics and AI

Why Python for data science?

- Python's popularity has steadily increased over the years, especially since 2018.
- Around 2020, Python surpassed several other major programming languages, including Java, to become one of the top-ranked languages.
- As of 2024, Python holds the highest rating on the index, showing it is currently the most popular programming language among the ones listed.



IDE: NoteBook

IDE: Notebook

- **What are notebooks?**
  - Notebooks combine code, the output from the code, and rich text elements (formatting, tables, figures, equations, links, etc.) in a single document.
- **Benefit of notebooks:**
  - Include commentary with your code. You can void the error-prone process of copying and pasting analysis results into a separate report
  - According to Kaggle Survey 2022 results, Jupyter Notebooks are the most popular data science IDE, used by over 80% of respondents
- **Types of notebooks:**
  - **Hosted** such as Google Colab, DataLab, Azure notebooks: No need to set up a local environment
  - **Local**: Install Jupyter notebook or JupyterLab
- ***Demo :Create a new notebook and write python code to print "Welcome to Data Analytics using Python !"***

Expressions

Expressions

- Expressions, which describe to the computer how to combine pieces of data.
- For example, a multiplication expression consists of a * symbol between two numerical expressions

```
3 * 4
```

```
12
```

```
3 * * 4
```

```
  File "<ipython-input-2-012ea60b41dd>", line 1
    3 * * 4
        ^
SyntaxError: invalid syntax
```

```
3 ** 4
```

```
81
```

Exponentiation expression (the first number raised to the power of the second: 3 times 3 times 3 times 3). The symbols * and ** are called ***operators***, and the values they combine are called ***operands***.

Expressions : Common Operators

- Python expressions obey the same familiar rules of *precedence* as in algebra: multiplication and division occur before addition and subtraction. Parentheses can be used to group together smaller expressions within a larger expression.

Expression Type	Operator	Example	Value
Addition	`+`	`2 + 3`	`5`
Subtraction	`-`	`2 - 3`	`-1`
Multiplication	`*`	`2 * 3`	`6`
Division	`/`	`7 / 3`	`2.66667`
Remainder	`%`	`7 % 3`	`1`
Exponentiation	`**`	`2 ** 0.5`	`1.41421`

Expressions : Exercise

- What the output of this expressions :

```
1 + 2 * 3 * 4 * 5 / 6 ** 3 + 7 + 8 - 9 + 10
```

```
1 + 2 * (3 * 4 * 5 / 6) ** 3 + 7 + 8 - 9 + 10
```

Declaring variables in Python

Variables

- Python has no command for declaring a variable
- A variable is created the moment you first assign a value to it

```
x = 5
y = "John"
print(x)
print(y)
```

```
x = 4       # x is of type int
x = "Sally" # x is now of type str
print(x)
```

```
height = 1.79
weight = 68.7
print(height)
print(weight)
```

```
1.79
68.7
```

Calculate $BMI = weight/height^2$

```
height = 1.79
weight = 68.7
bmi = weight/height**2
bmi
```

```
21.44127836209856
```

Casting

- If you want to specify the data type of a variable, this can be done with casting

```
x = str(3)    # x will be '3'
y = int(3)    # y will be 3
z = float(3)  # z will be 3.0
```

Variable Names

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)
- A variable name cannot be any of the Python keywords.

```
myvar = "John"
my_var = "John"
_my_var = "John"
myVar = "John"
MYVAR = "John"
myvar2 = "John"
```

Illegal variable names:

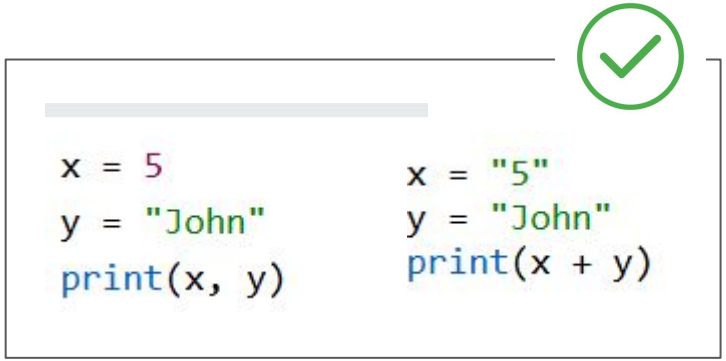
```
2myvar = "John"
my-var = "John"
my var = "John"
```

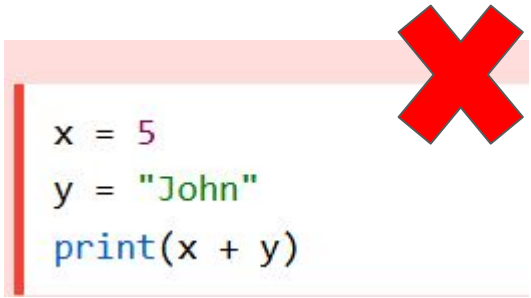
Python Variables - Assign Values

- Many Values to Multiple Variables
  - `x, y, z = "Orange", "Banana", "Cherry"`
- One Value to Multiple Variables
  - `x = y = z = "Orange"`
- Unpack a Collection
  - `fruits = ["apple", "banana", "cherry"]`
  - `x, y, z = fruits`

Output Variables

- The Python `print()` function is often used to output variables
  - `x = "Python is awesome"`
  - `print(x)`
- Output multiple variables, separated by a comma:
  - `x = "Python "` ← Space
  - `y = "is "`
  - `z = "awesome"`
  - `print(x, y, z)`
- Use the **+** operator to output multiple variables:
  - `x = 5`
  - `y = 10`
  - `print(x + y)`





Global Variables

- Global variables can be used by everyone, both inside of functions and outside.

```
x = "awesome"

def myfunc():
  print("Python is " + x)

myfunc()
```

Python is awesome

```
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)
```

Python is fantastic
Python is awesome

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```

Python is fantastic

Global Variables

- What is the output

```
x = "awesome"

def myfunc():

  global x

  x = "fantastic"

myfunc()

print("Python is " + x)
```

Python Types

Python Types

In Python, the data type is set whe

```
[6]: day_of_week = 5
     type(day_of_week)

[6]: int

[7]: x = "body mass index"
     type(x)

[7]: str

[8]: z = True
     type(z)

[8]: bool
```

Example	Data Type
x = "Hello World"	str
x = 20	int
x = 20.5	float
x = 1j	complex
x = ["apple", "banana", "cherry"]	list
x = ("apple", "banana", "cherry")	tuple
x = range(6)	range
x = {"name" : "John", "age" : 36}	dict
x = {"apple", "banana", "cherry"}	set
x = frozenset({"apple", "banana", "cherry"})	frozenset
x = True	bool
x = b"Hello"	bytes

Python Types

Different types have different behavior

```
2 + 3
```

```
5
```

```
"abc" + 'def'
```

```
'abcdef'
```

Specify the data type, using constructor functions

Example	Data Type
`x = str("Hello World")`	str
`x = int(20)`	int
`x = float(20.5)`	float
`x = complex(1j)`	complex
`x = list(("apple", "banana", "cherry"))`	list
`x = tuple(("apple", "banana", "cherry"))`	tuple
`x = range(6)`	range
`x = dict(name="John", age=36)`	dict
`x = set(("apple", "banana", "cherry"))`	set
`x = frozenset(("apple", "banana", "cherry"))`	frozenset
`x = bool(5)`	bool
`x = bytes(5)`	bytes

Python Operators

Python Arithmetic Operators

- Arithmetic operators are used with numeric values to perform common mathematical operations

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Python Assignment Operators

- Assignment operators are used to assign values to variables

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3

Python Identity Operators

- Used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location (***is, is not***)

```
x = ["apple", "banana"]
y = ["apple", "banana"]
z = x

print(x is z)

# returns True because z is the same object as x

print(x is y)

# returns False because x is not the same object as y, even if they have the same content

print(x == y)

# to demonstrate the difference betweeen "is" and "==": this comparison returns True
because x is equal to y
```

Python Membership Operators

- Membership operators are used to test if a sequence is presented in an object (***in, not in***)

```
x = ["apple", "banana"]

print("banana" in x)

# returns True because a sequence with the value "banana" is in the list
```

```
x = ["apple", "banana"]

print("pineapple" not in x)

# returns True because a sequence with the value "pineapple" is not in the list
```

Python Collections (Arrays)

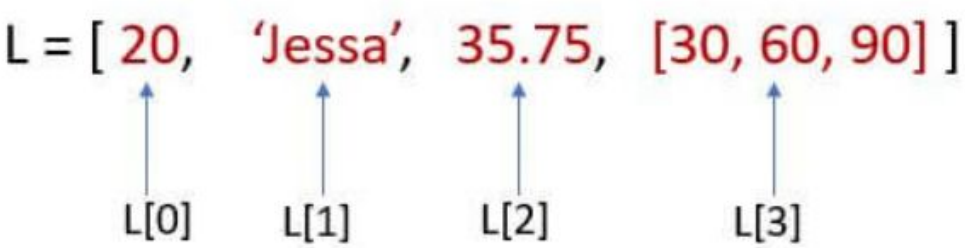
Python Collections (Arrays)

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered, unchangeable, and unindexed. No duplicate members.
    - Unchangeable : but you can remove and/or add items whenever you like
- **Dictionary** is a collection which is ordered and changeable. No duplicate members.
    - As of Python version 3.7, dictionaries are ordered. In Python 3.6 and earlier, dictionaries are unordered

Python List

Python List

- List items are
  - **Ordered**: In general, the order of the items will not change
  - **Changeable**: we can change, add, and remove items in a list after it has been created
  - **Allow duplicate values:** Since lists are indexed, lists can have items with the same value
  - **Heterogeneous:** Can contain multiple types
- List items are indexed, the first item has index [0], the second item has index [1] etc.



Python List

- **Create list**

```
list1 = ["apple", "banana", "cherry"]
list2 = [1, 5, 7, 9, 3]
list3 = [True, False, False]
```

```
thislist = list(("apple", "banana", "cherry"))
print(thislist)
```

```
#determine how many items a list has, use the len() function
thislist = ["apple", "banana", "cherry"]
print(len(thislist))
```

```
# height of entire family
hts = [1.73, 1.68, 1.71, 1.89 ]
hts
```

```
# Can contain multiple types
list = ['Father', 1.73, True, 10 ]
list
```

Python List: Access Items

- **Using indexing**, we can access any item from a list using its index number


```
hts = [1.73, 1.68, 1.71, 1.89 ]
print(hts[2])
print(hts[-1])
```

```
1.71
1.89
```

```
print(hts[5])
```

```
---------------------------------------
IndexError
Cell In[17], line 1
----> 1 hts[5]

IndexError: list index out of range
```

Python List: Access Items

- **Using slicing**, we can access a range of items from a list

```
hts = [1.73, 1.68, 1.71, 1.89 ]
hts[1:3]
```

```
[1.68, 1.71]
```

Note: The search will start at index **1** (included) and end at index **3** (not included).

- Start is inclusive

```
hts[2:]
```

```
[1.71, 1.89]
```

- End is exclusive

```
hts[:2]
```

```
[1.73, 1.68]
```

```
thislist = ["apple", "banana",
"cherry", "orange", "kiwi",
"melon", "mango"]
print(thislist[-4:-1])
```

Output : …….. ?

Python List: Modify list

```
# Change value at index
hts = [1.73, 1.68, 1.71, 1.89 ]
hts[2] = 1.75
print(hts)
```

```
[1.73, 1.68, 1.75, 1.89]
```

```
thislist = ["apple", "banana", "cherry"]

thislist[1:2] = ["blackcurrant", "watermelon"]

print(thislist)

['apple', 'blackcurrant', 'watermelon', 'cherry']
```

```
thislist = ["apple", "banana", "cherry", "orange", "kiwi", "mango"]

thislist[1:3] = ["blackcurrant", "watermelon"]

print(thislist)

['apple', 'blackcurrant', 'watermelon', 'orange', 'kiwi', 'mango']
```

```
thislist = ["apple", "banana", "cherry"]

thislist[1:3] = ["watermelon"]

print(thislist)

['apple', 'watermelon']
```

Python List: Add List Items

```
thislist = ["apple", "banana", "cherry"]
thislist.append("orange")
print(thislist)

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

```
thislist = ["apple", "banana", "cherry"]
thislist.insert(1, "orange")
print(thislist)

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

```
thislist = ["apple", "banana", "cherry"]
tropical = ["mango", "pineapple", "papaya"]
thislist.extend(tropical)
print(thislist)

['apple', 'banana', 'cherry', 'mango', 'pineapple', 'papaya']
```

```
hts = [1.73, 1.68, 1.71, 1.89 ]
```

```
# Concatenate lists
hts = hts + [1.81, 1.5]
print(hts)

[1.73, 1.68, 1.75, 1.89, 1.81, 1.5, 1.81, 1.5]
```

```
# Delete element by index
del hts[4]
hts

[1.73, 1.68, 1.75, 1.89, 1.5, 1.81, 1.5]
```

Python List: Remove List Items

```
thislist = ["apple", "banana", "cherry"]
thislist.remove("banana")
print(thislist)

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

```
thislist = ["apple", "banana", "cherry"]
thislist.pop(1)
print(thislist)

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

```
#Remove the first occurrence of "banana"
thislist = ["apple", "banana", "cherry", "banana", "kiwi"]
thislist.remove("banana")
print(thislist)

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

```
#If you do not specify the index
thislist = ["apple", "banana", "cherry"]
thislist.pop()
print(thislist)

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

```
thislist = ["apple", "banana", "cherry"]
del thislist[0]
print(thislist)

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

```
thislist = ["apple", "banana", "cherry"]
del thislist
print(thislist) #this will cause an error
#because you have succsesfully deleted "thislist".
```

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)

[]
```

Python List: Remove List Items

Method	Description
`remove(item)`	To remove the first occurrence of the item from the list.
`pop(index)`	Removes and returns the item at the given index from the list.
`clear()`	To remove all items from the list. The output will be an empty list.
`del list_name`	Delete the entire list.

Python List: Iterating a List

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

apple
banana
cherry
```

```
thislist = ["apple", "banana", "cherry"]
for i in range(len(thislist)):
  print(thislist[i])

apple
banana
cherry
```

```
thislist = ["apple", "banana", "cherry"]
i = 0
while i < len(thislist):
  print(thislist[i])
  i = i + 1
```

Python List: List Comprehension

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []

for x in fruits:
  if "a" in x:
    newlist.append(x)

print(newlist)

['apple', 'banana', 'mango']
```

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x.upper() for x in fruits]
print(newlist)

['APPLE', 'BANANA', 'CHERRY', 'KIWI', 'MANGO']
```

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]

print(newlist)
```

```
newlist = [expression for item in iterable if condition == True]
```

Python List: Sort Lists

- List objects have a **sort()** method that will sort the list alphanumerically, ascending, by default

```
thislist = [100, 50, 65, 82, 23]
thislist.sort()
print(thislist)

[23, 50, 65, 82, 100]
```

```
# Perform a case-insensitive sort of the list
thislist = ["A", "c", "B", "a"]
thislist.sort(key = str.lower)
print(thislist)

['A', 'a', 'B', 'c']
```

```
#Sort Descending
thislist = ["F", "R", "K", "A", "D"]
thislist.sort(reverse = True)
print(thislist)

['R', 'K', 'F', 'D', 'A']
```

```
# Perform a case-sensitive sort of the list
thislist = ["A", "c", "B", "a"]
thislist.sort()
print(thislist)

['A', 'B', 'a', 'c']
```

Python List: Copy Lists

- Cannot copy a list simply by typing `list2 = list1`, because: `list2` will only be a *reference* to `list1`, and changes made in `list1` will automatically also be made in `list2`.

```
thislist = [22, 44, 12]
mylist = thislist.copy()
print(mylist)

[22, 44, 12]
```

```
# Make a copy of a list with the list() method:
thislist = ["C#", "Java", "C++"]
mylist = list(thislist)
print(mylist)

['C#', 'Java', 'C++']
```

```
# Use the slice Operator
thislist = ["C#", "Java", "C++"]
mylist = thislist[:]
print(mylist)

['C#', 'Java', 'C++']
```

Python List: Join Lists

+

```
list1 = ["a", "b", "c"]
list2 = [1, 2, 3]

list3 = list1 + list2
print(list3)

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

append

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

for x in list2:
  list1.append(x)

print(list1)

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

extend

```
list1 = ["a", "b" , "c"]
list2 = [1, 2, 3]

list1.extend(list2)
print(list1)

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

Python List: List Methods

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

Functions

Python Functions

- A function is a block of code which only runs when it is called. It can receive data, known as parameters, into a function. A function can return data as a result.

```
..def my_function():
 …print("Hello from a function")

..my_function()
```

```
def my function(fname,lname):
  print(fname + " "+ lanme)

my_function("Ahmed", "Sabbah")
```

- If the number of arguments is unknown, add a * before the parameter name:
- Function will receive a ***tuple*** of arguments, and can access the items accordingly

```
def my function(*args):
  print("The youngest child is " + args[2])

my_function("Emil", "Ali", "Saleh")
```

Python Functions

- Send arguments with the ***key = value*** syntax

```
def my_function(child3, child2, child1):
  print("The youngest child is " + child3)

my_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")
```

- Arbitrary Keyword Arguments, **kwargs
  - If you do not know how many keyword arguments that will be passed into your function, add ** before the parameter name in the function definition.
  - The function will receive a ***dictionary*** of arguments, and can access the items accordingly

```
def my function(**kid):
  print("His last name is " + kid["lname"])

my_function(fname = "Ahmed", lname = "Sabbah")
```

Python Functions

- Default Parameter Value

```
def my_function(country =
"Norway"):
  print("I am from " + country)

my_function("Sweden")
my_function("India")
my_function()
my_function("Brazil")
```

- Return Values

```
def my_function(x):
  return 5 * x

print(my_function(3))
print(my_function(5))
print(my_function(9))
```

- Passing a List as an Argument

```
def my_function(food):
  for x in food:
    print(x)

fruits = ["apple", "banana",
"cherry"]

my_function(fruits)
```

- Positional-Only and Keyword-Only (**Exercise**)

```
def my_function(a, b, /, *, c, d):
  print(a + b + c + d)

my_function(5, 6, c = 7, d = 8)
```

Python Functions: Built-in Functions

- The **`max()`** and **`min()`** function returns the largest and smallest item in an iterable (like a list or tuple) or more arguments

```
numbers = [3, 7, 1, 9]
max(numbers)
```

```
9
```

```
numbers = [3, 7, 1, 9]
min(numbers)
```

```
1
```

- The **`round()`** function rounds a floating-point number to the nearest integer or to a specified number of decimal pla

```
value = 5.678
print(round(value)) # rounds to nearest integer
```

```
6
```

```
value = 5.678
print(round(value, 2)) # rounds to 2 decimal places
```

```
5.68
```

Python Functions: Built-in Functions

- `sum()` returns the sum of all items in an iterable (like a list or tuple). You can also specify a starting value for the sum

```
numbers = [1, 2, 3, 4]
print(sum(numbers))
```

```
10
```

- `len()` returns the length (number of items) of an object. This function works with many data types like strings, lists, tuples, and dictionaries

```
numbers = [1, 2, 3, 4]
print(len(numbers))
```

```
4
```

```
word = "Python"
print(len(word))
```

```
6
```

Python Functions: Built-in Functions

- **`sorted()`** returns a new sorted list from the items in an iterable. You can specify whether to sort in ascending or descending order with the reverse parameter.

```
numbers = [3, 1, 4, 1, 5]
print(sorted(numbers))
```

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

```
numbers = [3, 1, 4, 1, 5]
sorted(numbers, reverse=True)
```

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

- **`any()`** returns True if at least one element in the iterable is True; otherwise, it returns False

```
values = [False, False, True, False]
print(any(values))
```

```
True
```

- **`all()`** returns True if all elements in the iterable are True;

```
values = [True, True, True]
print(all(values))
```

```
True
```

Python Functions: Functions vs. Methods

- Python **functions** are defined independently and can be called on their own.
- **Methods** are functions that are associated with an object (like a list, string, dictionary, etc.). They belong to a class or data type and can only be called on instances of that type
- The `split()` function breaks a string into a list of substrings based on a specified separator (default is any whitespace).

```
text = "Python for data science and AI"
words = text.split()  # Splits the string into a list of words
print(words)
```

```
['Python', 'for', 'data', 'science', 'and', 'AI']
```

```
text = "Jerusalem,Gaza,Nablus"
fruits = text.split(',') # You can also specify a custom delimiter
print(fruits)
```

```
['Jerusalem', 'Gaza', 'Nablus']
```

Python Functions: String Methods

- The **`join()`** function does the opposite of split(). It combines a list of strings into a single string with a specified separator

```
words = ['Python', 'for', 'data', 'science', 'and', 'AI']
sentence = " ".join(words)
print(sentence)
```

```
Python for data science and AI
```

- The **`replace()`** function replace a specific substring within a string with another substring.

```
text = "Python for data science and AI"
new_text = text.replace("AI", "artificial intelligence")
print(new_text)
```

```
Python for data science and artificial intelligence
```

Python Functions: String Methods

- The **`append()`** function adds a single item to the end of a list

```
numbers = [1, 2, 3]
numbers.append(4)
print(numbers)
```

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

- The `index()` function returns the index of the first occurrence of a specified substring in the string. If the substring is not found, it raises a ValueError

```
text = "Python for data science and AI"
index_w = text.index("AI")
print(index_w)
```

```
28
```

```
text = "Python for data science and artificial intelligence"
index_w = text.index("AI")
print(index_w)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[26], line 2
      1 text = "Python for data science and artificial intelligence"
----> 2 index_w = text.index("AI")
      3 print(index_w)

ValueError: substring not found
```

Python Functions: List methods

- The **`index()`** function finds the first occurrence of a specified value in a list. If the value is found, it returns the index (position) of that value. If the value is not found, it raises a ValueError.

```
numbers = [10, 20, 30, 20, 40]
index_20 = numbers.index(20)
print(index_20)
```

```
1
```

```
numbers = [10, 20, 30, 20, 40]
index_5 = numbers.index(5)
print(index_5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[29], line 2
      1 numbers = [10, 20, 30, 20, 40]
----> 2 index_5 = numbers.index(5)
      3 print(index_5)

ValueError: 5 is not in list
```

Python Functions: List methods

- The **`count()`** function is used to count the number of times a specified value appears in a list

```
numbers = [10, 20, 30, 20, 40, 20]
count_20 = numbers.count(20)
print(count_20)
```

```
3
```

- The **`help()`** function is a built-in function that provides detailed information about Python objects such as functions, classes, modules, methods, and more. It is an incredibly useful tool when you need to understand how something works in Python, including its syntax, parameters, and description.

```
help(len)
```

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

len(obj, /)
    Return the number of items in a container.
```

Modules and Packages

Python Functions: Modules and Packages

- Python module (**.py**) file:
  - Organize the code
  - Make it reusable
- A module is simply a Python file (`.py`) that contains Python code
- it can have functions, classes, or variables
- The purpose of a module is to organize code and make it reusable across different programs
- Modules help you avoid writing the same code repeatedly
- By using a module, you can keep your code clean, organized, and easy to manage

```
import math as m

print(m.pi)
```

```
3.141592653589793
```

Python Functions: Modules and Packages

- Example : Let's say we have a Python file named **`math_operations.py`**

```python
# math_operations.py
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a*b
```

This file is now a module that we can import into other Python scripts to reuse

```python
import math_operations

result = math_operations.multiply(5, 3)
print(result)
```

```
15
```

Python Functions: Standard library modules

- **Some common built-in modules**
  - **`math`:** Contains mathematical functions
  - **`os`**: Provides functions to interact with the operating system
  - **`random`**: Used for generating random numbers

```
import math

math.sqrt(16)
```

```
import requests

response = requests.get("https://api.github.com")
print(response.status_code)
```

```
200
```

Python Functions: Python Package

- A package is a way of organizing multiple modules into a folder structure. It is simply a directory (folder) that contains a collection of related modules. Inside a package, you can have multiple `.py` files (modules)
- To create a package, you need:
  - A directory (folder) with the module files
  - An empty file named `__init__.py` in that directory
  - The `__init__.py` file can be empty or contain initialization code, but its presence tells Python that the directory should be treated as a package

Demo : Shapes example

Python Functions: Downloading Python packages

- In addition to Python's built-in modules and packages, there's a vast collection of third-party packages available online
- These packages can be easily installed and used in your own projects using Python's package manager, called **`pip`**
- **`pip`** is the standard tool used to download and install third-party packages. These packages are hosted on the Python Package Index (PyPI), which is a massive repository of software for Python

```
pip install pandas
```

```
import numpy as np

array = np.zeros(5)
print(array)
```

```
[0. 0. 0. 0. 0.]
```

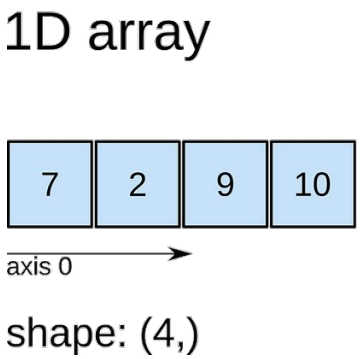
There are thousands of packages that allow you to add specialized functionality to your code

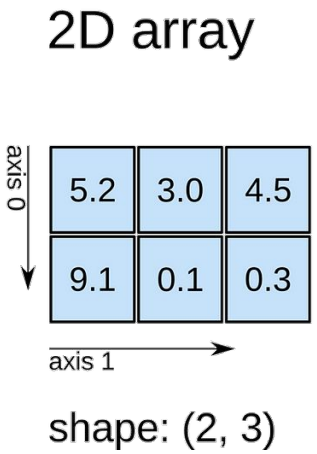
- **`numpy`**: Adds powerful numerical operations for handling arrays and matrices.
- **`pandas`**: Simplifies working with data and performing analysis.
- **`requests`**: Makes HTTP requests easier to handle (working with APIs).
- **`matplotlib`**: Creates plots and visualizations.

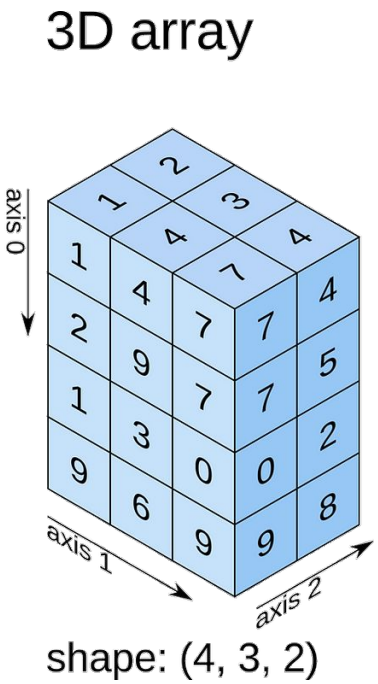
NumPy

Why NumPy?

- Python Lists are:
  - Powerful collection of values
  - Can hold different types
  - Can add, remove, and change
- BUT, Data Science requires:
  - Mathematical operations over collections
  - Speed







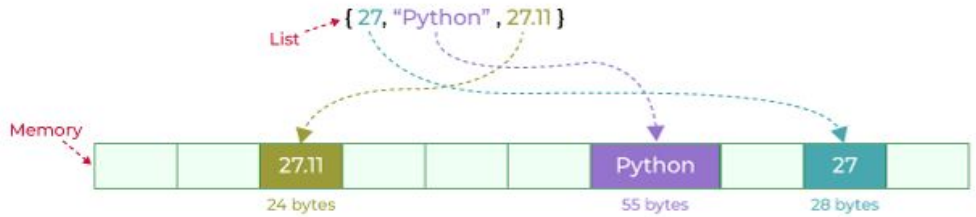
NumPyArray

- **NumPy** (short for Numerical Python) is a fundamental library for performing **mathematical and scientific computations** in Python
- It provides support for **multidimensional arrays** (`ndarray`)and a wide variety of mathematical operations on those arrays
- NumPy numerical operations **much faster** and **more efficient** than working with Python's built-in data structures, like lists.
- The NumPy API is used extensively in **Pandas, SciPy, Matplotlib, scikit-learn, scikit-image** and most other data science and scientific Python packages.

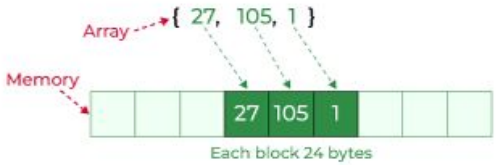
NumPyArray

Why NumPy Arrays are faster than Python Lists ?

- **Store homogeneous data**
  - Numpy arrays are densely packed arrays of homogeneous type
  - Python lists, by contrast, are arrays of pointers to objects, even when all of them are of the same type
  - Get the benefits of locality of reference



Python List



NumPy Array

- Memory size in bytes are in Mac OS X.

NumPyArray

- **Element-wise operations (Vectorization)**
  - To perform operations (like adding two lists element by element) using Python lists, you need to use loops or list comprehensions.
  - NumPy can use element-wise operations directly on entire arrays without loops
  - Making the code more concise and the execution faster

```
# Add two lists using Python List

list1 = [1,4,3,2,5,0]
list2 = [7,2,4,4,2,8]
list3 = []
for i in range(len(list1)):
    list3.append(list1[i] + list2[i])
list3
```

```
[8, 6, 7, 6, 7, 8]
```

```
# Add two lists using Numpy Array

import numpy as np

array1 = np.array(list1)
array2 = np.array(list2)

array3 = array1 + array2
array3
```

NumPy Array

- NumPy is imported and ready to use

```
import numpy
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)

[1 2 3 4 5]
```

- Using short alias

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)

[1 2 3 4 5]
```

- Using as

```
# Checking NumPy Version
import numpy as np
print(np.__version__)

1.16.3
```

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))

[1 2 3 4 5]
<class 'numpy.ndarray'>
```

NumPyArray

- The array object in NumPy is called **ndarray**, it provides a lot of supporting functions

```
import numpy
arr = numpy.array([1, 2, 3, 4, 5])
print(arr)
```

y easy.

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

rray) is the core data structure of NumPy

  - It represents a grid of values, all of the same type, indexed by non-negative integers
  - The dimensions of this array are known as its **shape**, and the number of dimensions is referred to as its **rank** (`ndim`)

- **Create ndarray:**
  - You can create an ndarray in various ways using NumPy. The most common way is to use the `np.array()` function, which takes a list or list of lists as input and converts it into a NumPy array

NumPy: Dimensions in Arrays

- 0-D Arrays
  - `arr = np.array(42)`
- 1-D Arrays
  - `arr = np.array([1, 2, 3, 4, 5])`
- 2-D Arrays
  - `arr = np.array([[1, 2, 3], [4, 5, 6]])`
- 3-D arrays
  - An array that has 2-D arrays (matrices) as its elements is called 3-D array
  - `arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])`

```
import numpy as np
a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim) # 0
print(b.ndim) # 1
print(c.ndim) # 2
print(d.ndim) # 3
```

ndim

```
import numpy as np
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(array_2d.shape)

(2, 3)
```

shape

NumPy: Element-wise operations

- **Question:** How to calculate the BMI for the family members below? ***bmi = weight / height ^2***
  - **Using Python list :** Element-wise operations are not supported in Python Lists, so we need to use loops to calculate the BMI for each family member.

```
height = [1.73, 1.68, 1.71, 1.89, 1.79]
weight = [65.4, 59.2, 63.6, 88.4, 68.7]
```

```
bmi = weight / height ** 2
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[8], line 1
----> 1 bmi = weight / height ** 2

TypeError: unsupported operand type(s) for ** or pow(): 'list' and 'int'
```

NumPy: Element-wise operations

- **Question:** How to calculate the BMI for the family members below? ***bmi = weight / height ^2***
  - The code below demonstrates how we can use **`ndarray`** to perform this calculation more efficiently:

```
height = [1.73, 1.68, 1.71, 1.89, 1.79]
weight = [65.4, 59.2, 63.6, 88.4, 68.7]
np_height = np.array(height)
np_weight = np.array(weight)
bmi = np_weight / np_height ** 2
bmi
```

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

NumPy: Data Type

- In NumPy, a regular array contains elements of the same data type (e.g., all integers or all floats)

```
np.array([1.0, "is", True])
```

```
array(['1.0', 'is', 'True'], dtype='<U32')
```

- Different types: different behavior!

```
python_list = [1, 2, 3]
python_list + python_list
```

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

```
numpy_array = np.array([1, 2, 3])
numpy_array + numpy_array
```

```
array([2, 4, 6])
```

NumPy: Subsetting

- **Examples**

```
bmi
```

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

```
# select element at specific index
bmi[1]
```

```
20.97505668934241
```

```
# Evaluate elements against condition
bmi > 23
```

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

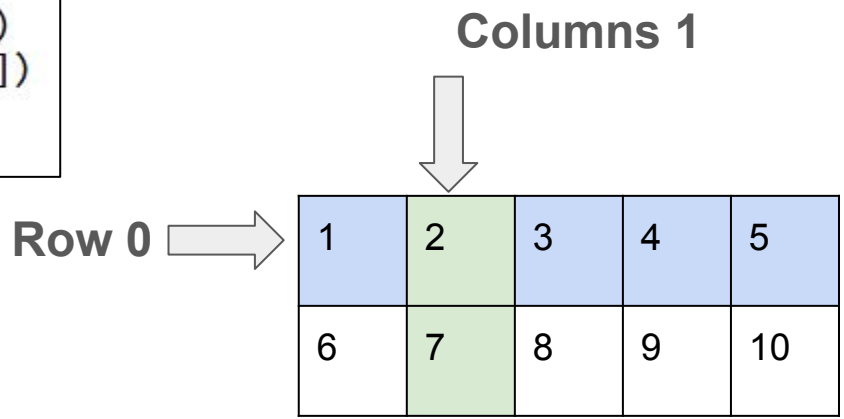
```
# Select elements which evaluate the condition to True
bmi[bmi > 23]
```

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

NumPy: Multidimensional Array

- To access elements from 2-D arrays, use **comma** separated integers representing the dimension and the index of the element
- Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st row: ', arr[0, 1])

2nd element on 1st dim:  2
```



Numpy: Multidimensional Array

- Example

```
# Create two dimensional arrays
np_array = np.array([[1.73, 1.68, 1.71, 1.89, 1.79],
                     [65.4, 59.2, 63.6, 88.4, 68.7]])
print(np_array.shape)
np_array
```

```
(2, 5)
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],
       [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ]])
```

```
# Check the type of np_array object
type(np_array)
```

```
numpy.ndarray
```

```
# Check the data type of the elements in the array
np_array.dtype
```

```
dtype('float64')
```

NumPy: Special arrays

- Example

```
# Create array of 10 zeros
np.zeros(10)
```

```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])
```

```
# Create array of 14 ones
np.ones(14)
```

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

```
# Create empty array of specific dimension
np.empty((2,3,3))
```

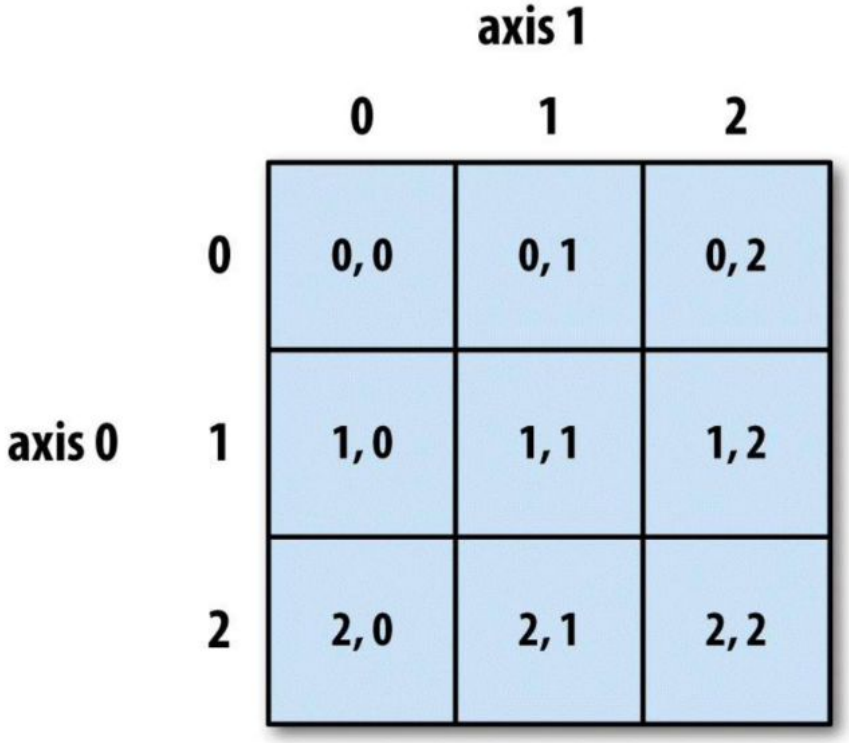
```
array([[[6.23042070e-307, 4.67296746e-307, 1.69121096e-306],
        [7.56603202e-307, 7.56587584e-307, 1.37961302e-306],
        [1.05699242e-307, 8.01097889e-307, 1.78020169e-306]],

       [[7.56601165e-307, 1.02359984e-306, 1.15710088e-306],
        [7.56597091e-307, 1.69118787e-306, 6.89806509e-307],
        [1.24611266e-306, 2.22522596e-306, 2.56765117e-312]]])
```

```
# Create array with range of values from 0 to 14
np.arange(15)
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

NumPy: Access Arrays

- **2-D array**



NumPy: Access 2-D Arrays

```
np_2d = np.array([[1.73, 1.68, 1.71, 1.89, 1.79],
                  [65.4, 59.2, 63.6, 88.4, 68.7],
                  [55.4, 49.2, 53.6, 78.4, 58.7]])
```

```
np_2d[0]
```

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

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

```
1.71
```

```
np_2d[1, 0]
```

```
65.4
```

```
np_2d[0:2, 1:3]
```

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

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```

Output:.....

NumPy: Access 3-D Arrays

- Accessing single elements in a 3D Array
- **3D array** has three dimensions: `(depth, row, column)`

```
# Creating a 3D array (2 matrices, each 2x3)
arr_3d = np.array([
    [[1, 2, 3],
     [4, 5, 6]],

    [[7, 8, 9],
     [10, 11, 12]]
])
```

- Depth 1 (second matrix)
- Row 0 (first row)
- Column 2 (third column)
- Value 9 is returned

```
# Accessing a single element
element = arr_3d[1, 0, 2]  # Depth=1, Row=0, Column=2
print("\nElement at (1, 0, 2):", element)  # Output: 9
```

NumPy: NumPy Array Slicing

- Slicing in python means taking elements from one given index to another given index
- We pass a slice instead of an index like this: ***[start:end]***
- We can also define the step like this: ***[start:end:step]***
- If we don't pass **start,** it's considered 0
- If we don't pass **end,** it's considered length of array in that dimension
- If we don't pass **step,** it's considered 1

One major difference between NumPy arrays and Python's built-in lists is that **array slices in NumPy are views, not copies** of the original array

Numpy: NumPy Array Slicing

- It is different from Python's built-in lists in that **array slices** are **views** on the original array:

```
np_2d
```

```
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],
       [65.4 , 59.2 , 63.6 , 88.4 , 68.7 ],
       [55.4 , 49.2 , 53.6 , 78.4 , 58.7 ]])
```

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

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

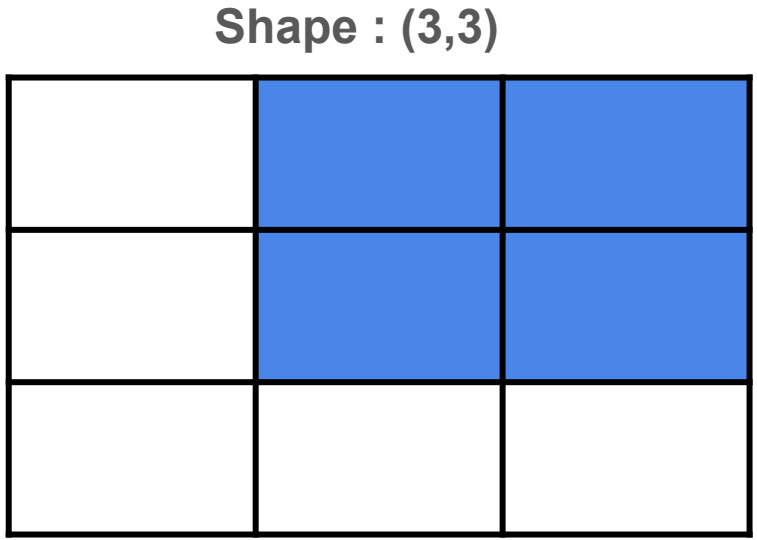
```
slice[0] = -1
np_2d
```

```
array([[ 1.73,  1.68,  1.71,  1.89,  1.79],
       [65.4 , -1.  , 63.6 , 88.4 , 68.7 ],
       [55.4 , 49.2 , 53.6 , 78.4 , 58.7 ]])
```

NumPy: Exercises

- **Exercise: 1**. How you select the shaded region ?
- **General slicing format**
- array[**row_start**:**row_end, col_start**:**col_end**]

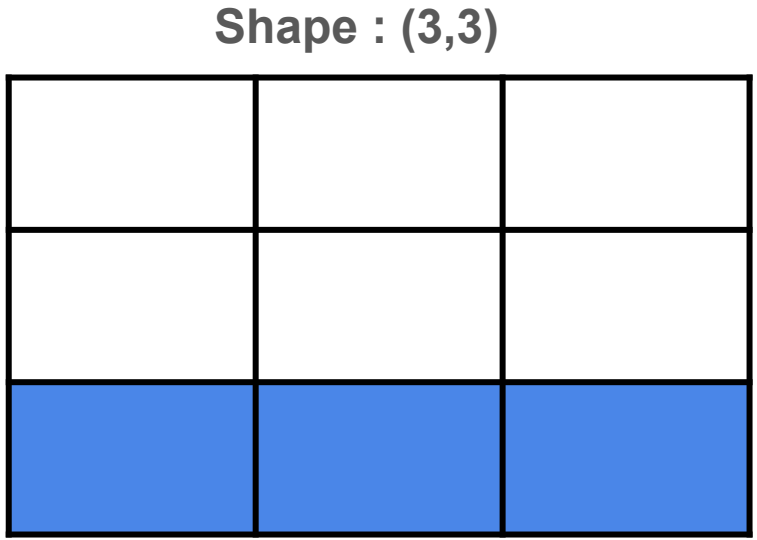
- `arr[:2, 1:]`, **shape = (2,2)**



NumPy: Exercises

- **Exercise: 2**. How you select the shaded region ?
- **General slicing format**
- array[row_start:row_end, col_start:col_end]

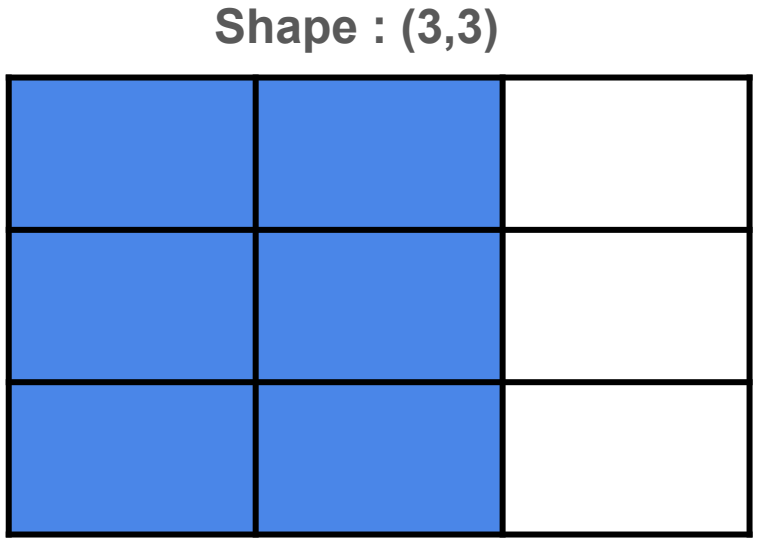
- `arr[2]`, **shape = (3,)**
- `arr[2, :]`, **shape = (3,)**
- `arr[2:,:]`, **shape = (1, 3)**



NumPy: Exercises

- **Exercise: 3**. How you select the shaded region ?
- **General slicing format**
- array[**row_start**:**row_end, col_start**:**col_end**]

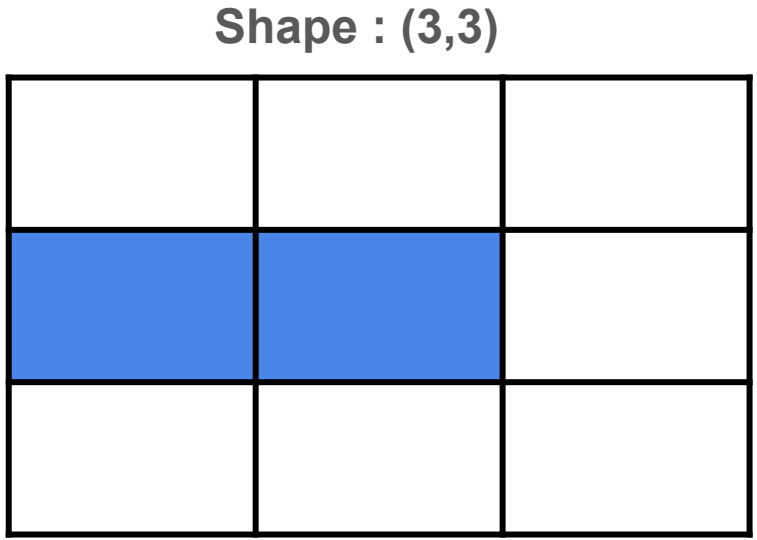
- `arr[: ,:2]`, **shape = (3,2)**



NumPy: Exercises

- **Exercise: 4**. How you select the shaded region ?
- **General slicing format**
- array[**row_start**:**row_end, col_start**:**col_end**]

- `arr[1 ,:2]`, **shape = (2,)**
- `arr[1:2 ,:2]`, **shape = (1, 2)**



NumPy: Reshape Array

- You can change the shape of an array using the **`reshape()`** method without changing the data itself

```
arr = np.arange(15)
arr
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14])
```

```
arr.reshape(5,3)
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

```
arr = np.arange(14)
arr.reshape(5,3)
```

Output:..... ?

NumPy: Transpose Array

- **Transposing** an array means flipping it over its **diagonal**, swapping its **rows and columns**

```
arr = np.arange(15).reshape((3, 5))
arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
arr.T
```

```
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

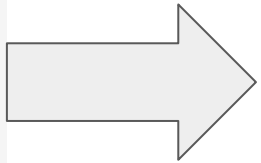
Does not modify original array, it returns a new one.

NumPy: Aggregation Functions

- NumPy provides many aggregation functions that can be applied to arrays, such as
- `sum()`, `mean()`, `max()`, and `min()`

```
# Create arrays
a = np.array([[1, 2, 3],
              [4, 5, 6]])

b = np.array([10, 20, 30])

# Perform element-wise addition
result = a + b

print(result)
```

```
[[11 22 33]
 [14 25 36]]
```



```
# Sum of all elements
print(np.sum(result))

# Mean of elements
print(np.mean(result))

# Maximum value in the array
print(np.max(result))
```

```
141
23.5
36
```

NumPy: Broadcasting

- Broadcasting allows NumPy to perform operations between arrays of different shapes, by automatically expanding the smaller array to match the shape of the larger one

```
array_1d = np.array([1, 2, 3])
array_2d = np.array([[10], [20], [30]])

# Broadcasting the 1D array to each row of the 2D array
result = array_2d + array_1d
result
```

```
array([[11, 12, 13],
       [21, 22, 23],
       [31, 32, 33]])
```

End of Python 1

More bout NumPy : W3schools