CMU Portugal Academy

CMU Portugal
Advanced Training Program

Foundations of Data Science

DAVID SEMEDO
RAFAEL FERREIRA
NOVA SCHOOL OF SCIENCE AND TECHNOLOGY

Carnegie Mellon Portugal · TÉCNICO LISBOA · Ciências ULisboa · NOVA School of Science & Technology

Email: df.semedo@fct.unl.pt

David Semedo– Short bio

df.semedo@fct.unl.pt

Assistant Professor @ NOVA FCT, Integrated Researcher @ NOVA LINCS

AI for vision and language. Neural approaches to conversational and contextualized media understanding.

Rafael Ferreira– Short bio

rah.ferreira@campus.fct.unl.pt

4th year PhD Student @ NOVA FCT

Conversational AI. Team leader of the award-winning TWIZ in the Alexa TaskBot Challenge.

Today's Topics

1. MODULE STRUCTURE AND LOGISTICS
2. INTRO TO DATA SCIENCE
3. INTRO TO PYTHON
4. DEALING WITH NUMERICAL DATA

What will you accomplish today?

- Assess Driver's License Eligibility
- Processing Sensor Data
- Sales performance evaluation
- Costumer churn prediction

CMU Portugal Academy

01

Module Structure and Logistics

Carnegie Mellon Portugal · TÉCNICO LISBOA · Ciências ULisboa · NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Module Schedule and Sessions

- 4 Data Science + Python weeks!

- Hands-On Course

	Friday	Saturday
9:00 - 10:00		**Theory + Lab** Lecture + Hands-on + Team-based Work Exercises
10:00 - 11:00		
11:00 - 12:00		
12:00 - 13:00		
13:00 - 16:30		
16:30 - 17:00	**Theory + Lab** Lecture + Hands-on	
17:00 - 18:00		
18:00 - 19:00		
19:00 - 20:00		

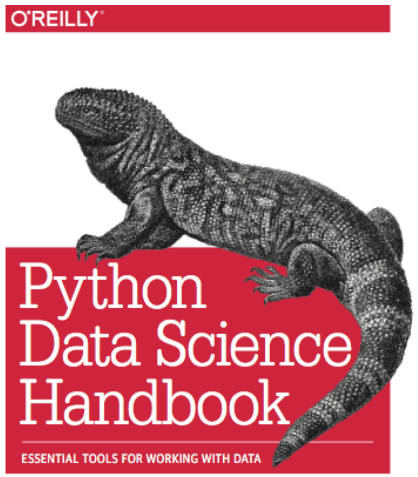
Module Topics

- Introduction to Data Science and Python Programming
- Statistics and Probability
- Data Preparation and Processing with Pandas
- Machine Learning Fundamentals
- Model Evaluation and Selection
- Data Visualization

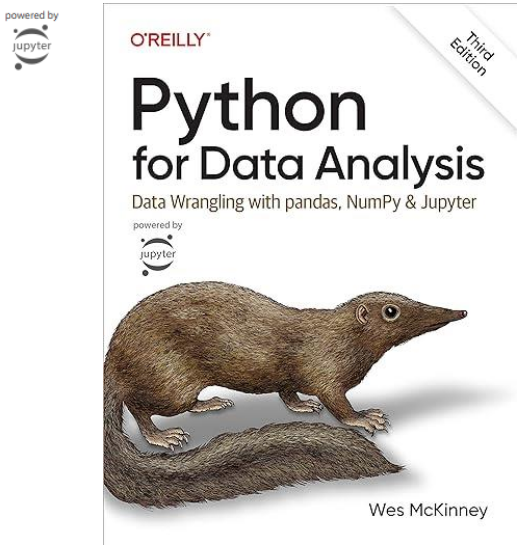
Goals

- Understand the foundations and applications of Data Science.
- Learn to identify and extract insights from data using Python tools like NumPy, pandas and visualization libraries.
- Apply foundational techniques to build simple predictive models.

Bibliography and References



- Jake VanderPlas. 2016. **Python Data Science Handbook: Essential Tools for Working with Data** (1st. ed.). O'Reilly Media, Inc.
- Wes McKinney. 2022. **Python for Data Analysis: Data Wrangling with pandas, NumPy, and Jupyter** (3rd. Ed.). O'Reilly Media, Inc.
- Other tools and resources shared throughout the course.



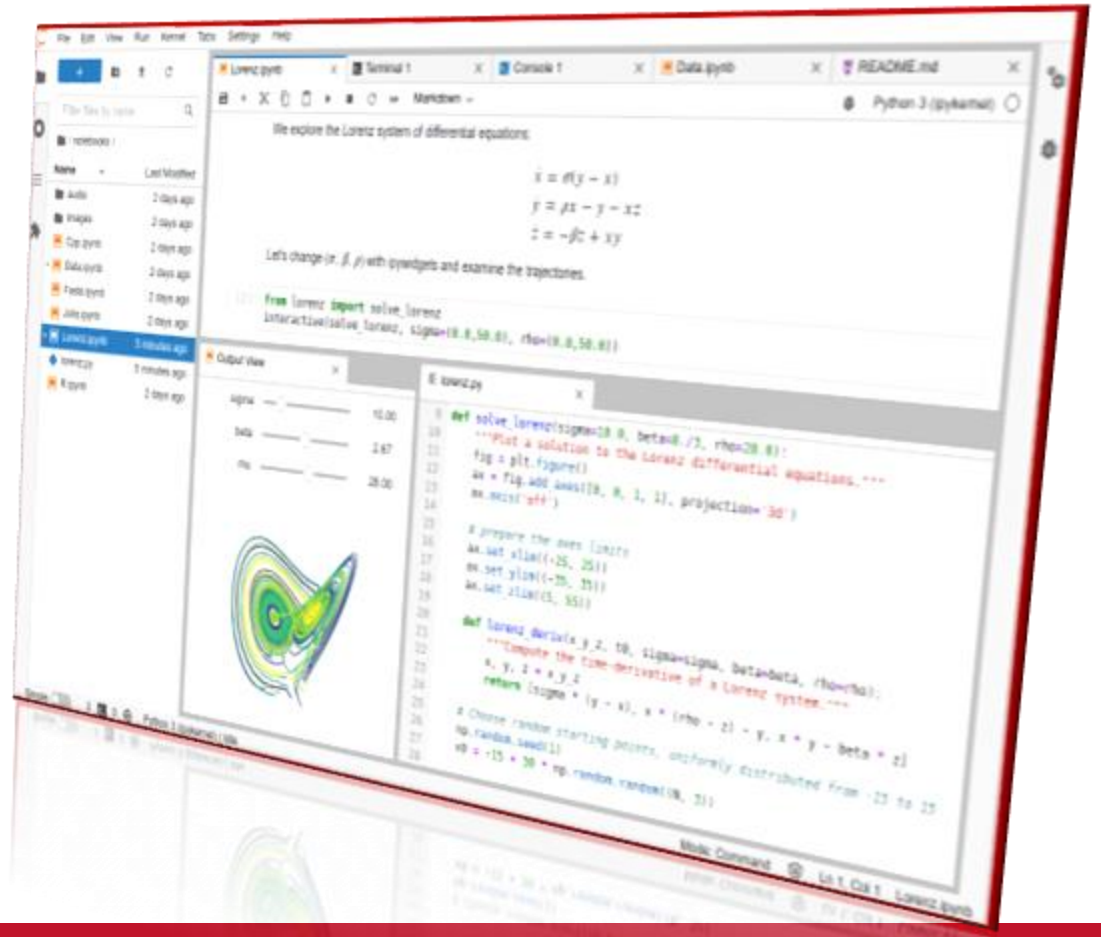
CMU Portugal Academy

Course Shared Folder

Laboratory setup

We will use your bootcamp setup!

ANACONDA® + jupyter



https://wiki.novasearch.org/wiki/lab_setup

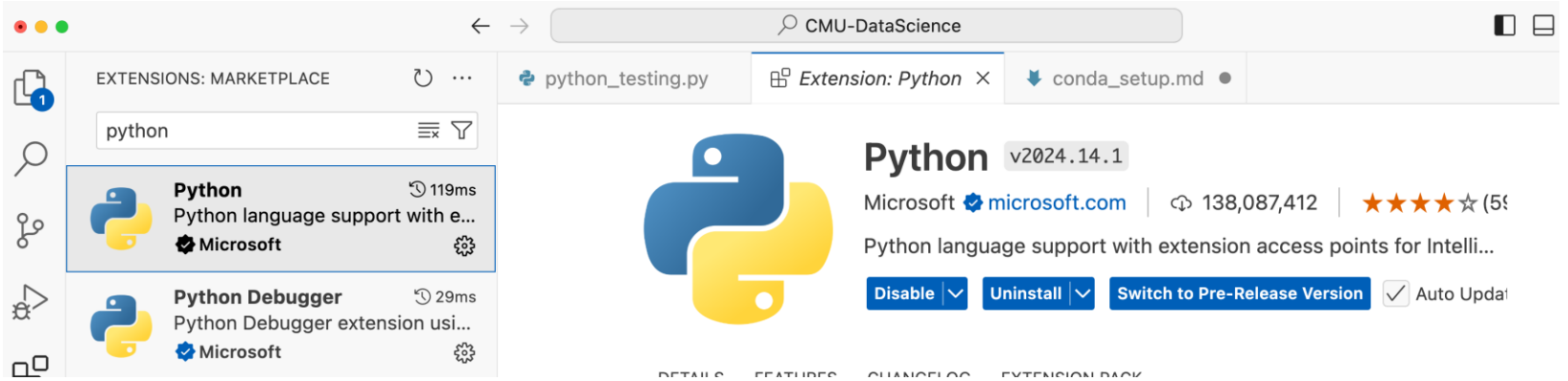
Laboratory setup – VS Code

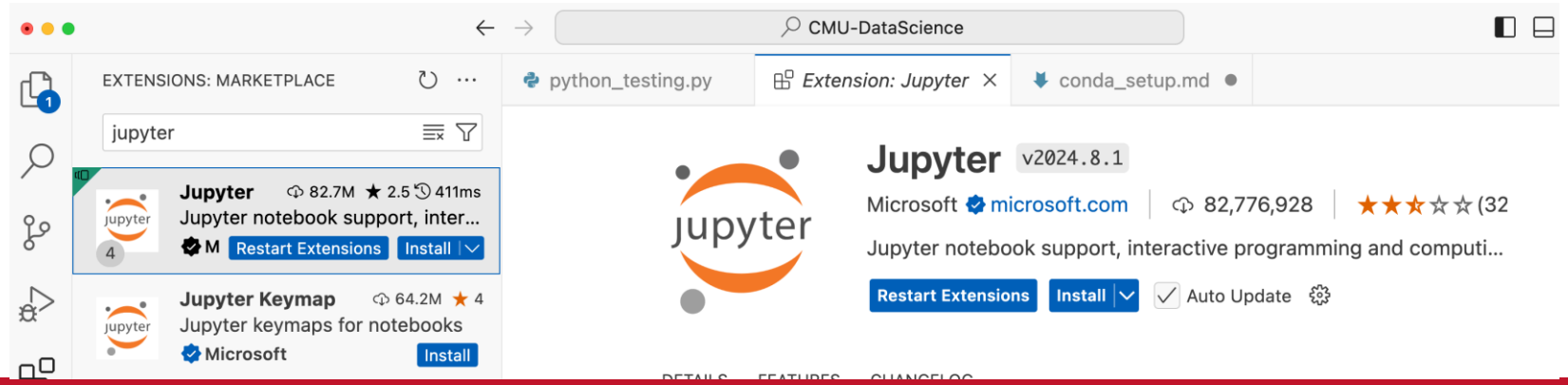
VS Code

[Download VS Code]

Install two extensions:

- Python
- Jupyter (Docs)





Course Use-Case

Apply Data Science tools to work with a specific use-case:

- Groups of 3 students
- To be done partially in lab sessions (on Saturdays)
- Final presentation in the last course session (18th October)

Course Use-Case

Apply Data Science tools to work with a specific use-case.

Pick an interesting dataset:

- From Kaggle (Link) or UCI (Link)
- Bring your own

Requirements:

- Between 500 to 100k samples
- Tabular file (csv)
- Only categorical and/or numerical features

Bring your dataset next week to get our feedback!

CMU Portugal Academy

02

Intro to Python Programming

Carnegie Mellon Portugal · TÉCNICO LISBOA · Ciências ULisboa · NOVA School of Science & Technology

Constants: Numbers

float
(floating point numbers)

int (Integer numbers)

```
In [1]: 1234
Out[1]: 1234

In [2]: -56
Out[2]: -56

In [3]: +3
Out[3]: 3

In [4]: 0
Out[4]: 0
```

```
In [1]: 3.64
Out[1]: 3.64

In [2]: -0.0747
Out[2]: -0.0747

In [3]: 2.0
Out[3]: 2.0

In [4]: 0.0
Out[4]: 0.0
```

Constants: String

str
(String - sequence of characters between single or double quotation marks)

```
In [1]: "Hello, World!"
Out[1]: 'Hello, World!'

In [2]: 'Hello, World!'
Out[2]: 'Hello, World!'

In [3]: " Bom dia  :-) "
Out[3]: ' Bom dia :-) '

In [4]: "2 + (3 * 6)"
Out[4]: '2 + (3 * 6)'
```

Use double quotation marks!

Constants: Function `type`

Constants have a **value** and a **type**.

```
In [1]: type(234)
Out[1]: int

In [2]: type(-1.24)
Out[2]: float

In [3]: type(3.0)
Out[3]: float

In [4]: type("Bom dia  :-)")
Out[4]: str
```

Arithmetic expressions: operators

- Basic arithmetic operators:
  +, -, *, /, **, //, %

```
In [1]: 7 + 12 # addition
Out[1]: 19

In [2]: 6 / 2 # division (result is float)
Out[2]: 3.0

In [3]: 5 // 2 # integer division (result is int)
Out[3]: 2

In [4]: 3 ** 2 # exponentiation
Out[4]: 9

In [5]: 7 % 2 # division remainder
Out[5]: 1

In [6]: (5 / 2) * (6 - 3)
Out[6]: 7.5

In [7]: 2.3 + 4.6
Out[7]: 6.8999999999999995
```

Reference: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html

Arithmetic expressions: result type

The result type of the evaluating an arithmetic expression depends on the operator and the types of the two operands

The result type is

- **int** if the operands are of type **int** and the operator is +, -, *, ** , // ou %
- **float** in any other case.

Type Conversions in Python

- round(e) rounds the value of e to the nearest int
- int(e) converts the value of e (string or number) to int
- float(e) converts the value of e (string or number) to float

```
In [1]: round(2.9)
Out[1]: 3

In [2]: int(2.9)
Out[2]: 2

In [3]: float(5)
Out[3]: 5.0

In [4]: float("45")
Out[4]: 45.0
```

Operator precedence

How is the following expression evaluated?

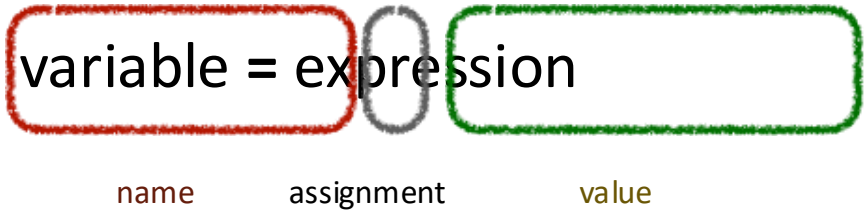
```
In [1]: 5 + 2 ** 3 * 2
Out[1]: 21
```

The following order (plus associativity) resolve ambiguity

Operator	Associativity	Precedence
()		Highest
**	right	
- (symmetry)	right	
*, /, //, %	left	
+, -	left	Lowest

Variables and assignment

- A variable is a symbolic name given to a **computer memory location** which is used to **store a value**.
- The assignment statement:



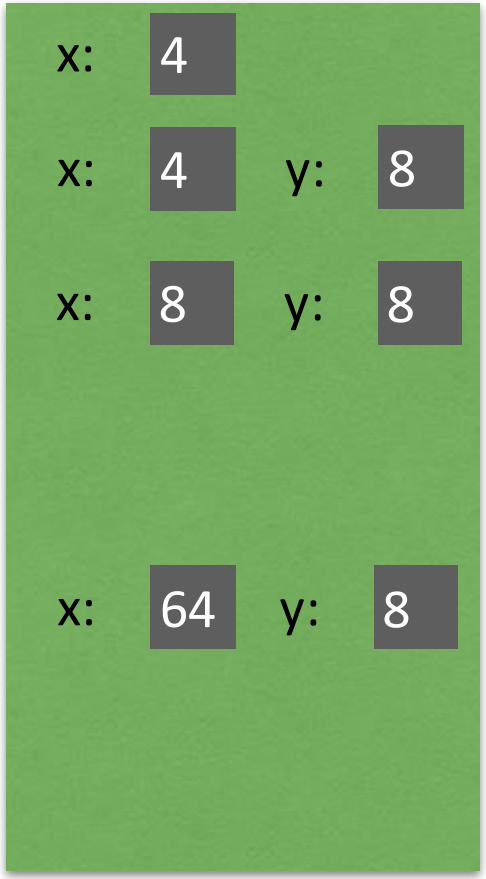
- First the expression is evaluated and then the association of the name to the value is made.
- Python keeps a record of the variables defined and their current values, this is called the *environment.*
- If the variable does not exist, it is added to the environment. Otherwise, its value is replaced by the value of the expression.

Variables and binding

IPython console

```
In [1]: x = 4

In [2]: y = x * 2

In [3]: x = y

In [4]: x
Out[4]: 8

In [5]: x = x ** 2

In [6]: x
Out[6]: 64
```

Environment



Variable names

- The name of a variable is a sequence of alphanumeric characters and underscore, that starts with a letter or an underscore.
  - grade, _max, minValue, name ✓
  - 1teste ✗
- Variable names are case-sensitive.
  - Names **Grade** and **grade** describe different variables.

Variable names: Reserved keywords

- There is a set of Python keywords that cannot be used as variable names.

FALSE	class	finally	is	return
None	continue	for	lambda	try
TRUE	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

Data input and output

- Function **input** reads a string from the terminal.
- To read numeric values we need to convert to the desired type.

```
In [1]: value = input("Amount to withdraw: ")
Amount to withdraw: 120

In [2]: type(value)
Out[2]: str

In [3]: value = int(input("Amount to withdraw: "))
Amount to withdraw: 120

In [4]: type(value)
Out[4]: int
```

Data input and output

- The function **print** writes data to the terminal.

- This function does not return any value, it presents on the screen the result of the provided expression!

```
In [1]: print()

In [2]: print("Hello, World!")
Hello, World!

In [3]: print("2 + 4")
2 + 4

In [4]: print(2 + 4)
6

In [5]: print("2 + 4 = ", 2 + 4)
2 + 4 = 6
```

Escape characters in Strings

- What if we wanted to write a String that has double quotes? For instance,

Alice said "I'm going for a coffee."

```
In [1]: print("Alice said "I'm going for a coffee."")
  File "<ipython-input-1-a4cbf5d481bb>", line 1
    print("Alice said "I'm going for a coffee."")
                       ^
SyntaxError: invalid syntax

In [2]: print("Alice said \"I'm going for a coffee.\"")
Alice said "I'm going for a coffee."
```

Escape characters in Strings

What if we wanted to write a String that has double quotes? For instance,

Escape Sequence	Meaning
\’	Single quote
\\’	Double quote
\\	Backslash
\n	Newline
\r	Carriage Return
\t	Horizontal Tab
\b	Backspace
\f	Formfeed
\v	Vertical Tab
\0	Null Character
\Uxxxxxxxx	Unicode character with a 32-bit hex value
\000	Character with octal value ooo
\xhh	Character with hex value hh

Hands-On Mini-session

Let’s write a function that computes the net expected winnings from a bet, according to a given win probability, potential reward, and bet amount.

It should consider a 23% tax deduction on the winnings.

Hands-On Mini-session

```
probability = 0.7  # 70% chance of winning
reward_amount = 100  # 100€ reward
tax_rate = 0.23  # 23% tax

# Reading the bet amount from the user
bet_amount = float(input("Enter the bet amount: "))

# Calculate expected winnings before tax
expected_win = probability * reward_amount

# Apply tax to the expected winnings
expected_win_after_tax = expected_win * (1 - tax_rate)

net_winnings = expected_win_after_tax - bet_amount

print("Net expected winnings after tax and bet:" , round(net_winnings, 2))
```

Booleans

- Beyond integers and floats, Python also has booleans (**bool**).
  - Only two constants: True and False

```
In [1]: True
Out[1]: True

In [2]: False
Out[2]: False

In [3]: type(True)
Out[3]: bool
```

bool
(logic value)

The None value

Has everyone come across None/NULL/null/Nothing/nil/0?

The None keyword is used to define a null value, or no value at all.

```
In [12]: None

In [13]: print(None)
None

In [14]: type(None)
Out[14]: NoneType

In [15]: 4 + None
---------------------------------------------
TypeError   Traceback (most recent call last)
Cell In[15], line 1
----> 1 4 + None

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

In [16]:
```

Logic operators

- Operators:

and (conjunction)	or (disjunction)	not (negation)

- Returns a boolean value from boolean values

```
In [1]: not True # negation
Out[1]: False

In [2]: True or False # disjunction
Out[2]: True

In [3]: True and False # conjunction
Out[3]: False

In [4]: not 0 # zero (int or float) is interpreted as False
Out[4]: True

In [5]: not 34 # any number not zero is interpreted as True
Out[5]: False
```

Relational operators

== (equality)	!= (inequality)
< (less than)	<= (less or equal than)
> (greater than)	>= (greater or equal than)

- These operators return a boolean value from numeric values (**int** or **float**)

```
In [1]: x = 10  # assign value 10 to variable x

In [2]: x == 10 # checks if the value of variable x is equal to 10
Out[2]: True

In [3]: x >= 12 # checks if the value of variable x is greater or equal than 10
Out[3]: False

In [4]: x != 12 # checks if the value of variable x is not equal to 12
Out[4]: True
```

Chaining relational operators

```
In [1]: x = 10  # assign value 10 to variable x

In [2]: 0 <= x <= 20 # checks if the value of x in in the interval [0,20] Out[2]: True
```

0 <= x <= 20 is equivalent to 0 <= x **and** x <= 20

Conditional statement: if-else

Syntax

```
if <exp_bool> :
    <instructions_true>
else :
    <instructions_false>
```

Example

```
if x % 2 == 0 :
    print("x is an even number")
else :
    print("x is an odd number")
```

- Evaluation steps
  1. Evaluate boolean expression <exp_bool>
     - If the value is **True** instructions <instructions_true> are executed.
     - If the value is **False** instructions <instructions_false> are executed.

Chained conditional statements

Syntax

```
if <exp_bool_1> :
   <instrucoes_1>
elif <exp_bool_2> :
   <instrucoes_2>
else :
   <instrucoes_3>
```

Example

```
if x < y :
   print("x is less than y")
elif x > y :
   print("x is greater than y")
else :
   print("x is equal to y")
```

- The `else` can be omitted.

Python Functions

- A function in Python is a sequence of instructions that is assigned a name.
- We can provide input parameters.
- Can `return` a result (or not)
- We can define our own functions.

```
def name(<parameters>):
    <instructions>
                [return <result>]
```

Rewriting our solution to the previous problem in a Function.

```
probability = 0.7  # 70% chance of winning
reward_amount = 100  # 100€ reward
tax_rate = 0.23  # 23% tax

# Reading the bet amount from the user
bet_amount = float(input("Enter the bet amount: "))

# Calculate expected winnings before tax
expected_win = probability * reward_amount

# Apply tax to the expected winnings
expected_win_after_tax = expected_win * (1 - tax_rate)

net_winnings = expected_win_after_tax - bet_amount

print("Net expected winnings after tax and bet:" , round(net_winnings, 2))
```

CMU Portugal Academy

Rewriting our solution to the previous problem in a Function.

```python
def expected_reward(bet_amount):

    probability = 0.7  # 70% chance of winning
    reward_amount = 100  # 100€ reward
    tax_rate = 0.23  # 23% tax

    # Calculate expected winnings before tax
    expected_win = probability * reward_amount

    # Apply tax to the expected winnings
    expected_win_after_tax = expected_win * (1 - tax_rate)

    net_winnings = expected_win_after_tax - bet_amount

    return round(net_winnings, 2)
```

✓ 0.0s Python

```python
bet = float(input("Enter the bet amount: "))
result = expected_reward(bet)
print("Net expected winnings after tax and bet:" , result, "€")
```

✓ 3.0s Python

```
Net expected winnings after tax and bet: 23.9 €
```


Carnegie Mellon Portugal · TÉCNICO LISBOA · Ciências ULisboa · NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Functions and Local Variables

- A function has its own environment/scope.
- Local variables are destroyed after execution

```
[18]  def power_twice(n, power):
          n_power = n**power
          return 2*n_power
      ✓ 0.0s                                                    Python

[20]  power_twice(10, 2)
      ✓ 0.0s                                                    Python
...   200

[21]  print(n_power)
      ⊗ ✧ 0.0s                                                  Python
...   ---------------------------------------------------------------
      NameError                                                 Traceback
      Cell In[21], line 1
      ----> 1 print(n_power)

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

Structuring programs with functions

- A program consists of multiple functions.
- Each concept should be programmed by a distinct function.
- Each function must have a name that clearly indicates the task it performs.
- Each function receives as arguments the data it needs to perform its task.

Program Structure Good Practices

There are two types of functions: domain and interface.

Domain functions:

Implement the program logic.

Should not include user interaction (no input or print instructions).

I**nterface** functions:

Implement the interaction with the user.

Read data and present results.

Interface functions call the domain functions, but the opposite should not happen.

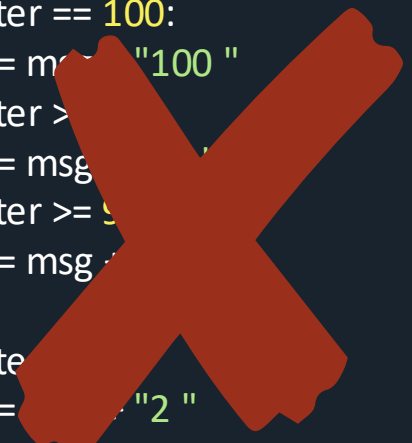
Do not mix function types - separate the logic from the user interface.

Function countdown: first attempt

```
def countdown(counter):
    """countdown : int [0,3] -> str
    Description: build a string with a regressive counting from
      counter (a value between 0 and 3) to 1
    Examples: countdown(3) -> "3 2 1 GO!"; countdown(0) -> "GO!"
    """

    msg = ""
    if counter == 3:
        msg = msg + "3 "
    if counter >= 2:
        msg = msg + "2 "
    if counter >= 1:
        msg = msg + "1 "
    msg = msg + "GO!"
    return msg
```

```
def countdown(counter):
    """countdown : int [0,100] -> str
    Description: build a string with a regressive counting from
      counter (a value between 0 and 100) to 1
    Examples: countdown(3) -> "3 2 1 GO!"; countdown(0) -> "GO!"
    """

    msg = ""
    if counter == 100:
        msg = msg + "100 "
    if counter >= 99:
        msg = msg + "99 "
    if counter >= 98:
        msg = msg + "98 "
            ...
    if counter >= 2:
        msg = msg + "2 "
    if counter >= 1:
        msg = msg + "1 "
    msg = msg + "GO!"
    return msg
```

```
def countdown(counter):
    """countdown : int [0,100] -> str
    Description: build a string with a regressive counting from
      counter (a value between 0 and 100) to 1
    Examples: countdown(3) -> "3 2 1 GO!"; countdown(0) -> "GO!"
    """

    msg = ""
    if counter == 100:
        msg = m      "100 "
    if counter >
        msg = msg
    if counter >= 9
        msg = msg
            ...
    if counte
        msg =      "2 "
    if counter >= 1:
        msg = msg + "1 "
    msg = msg + "GO!"
    return msg
```



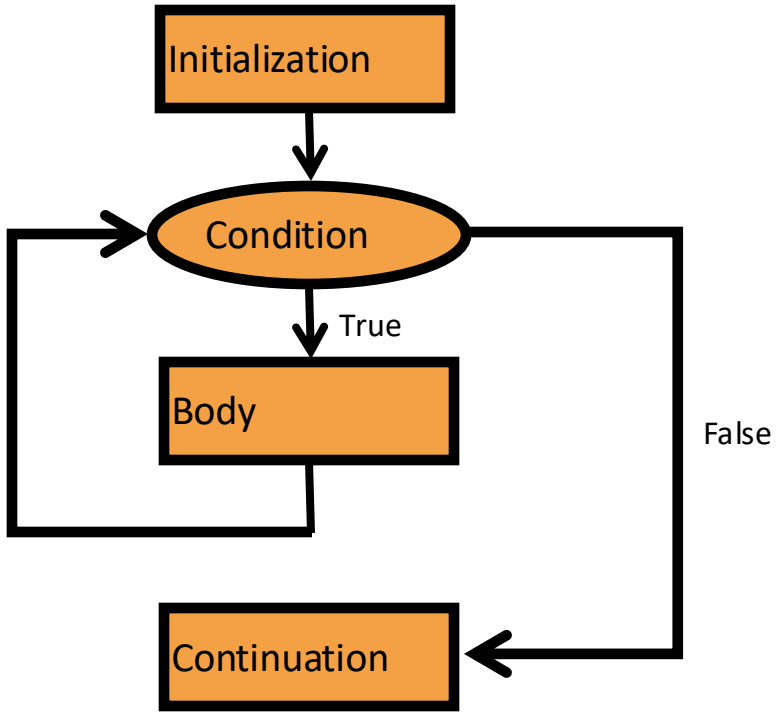
"Solution" **inadequate!**

- Impractical long sequence of ifs.
- Arbitrary upper limit - **lacks generality**!

While loop

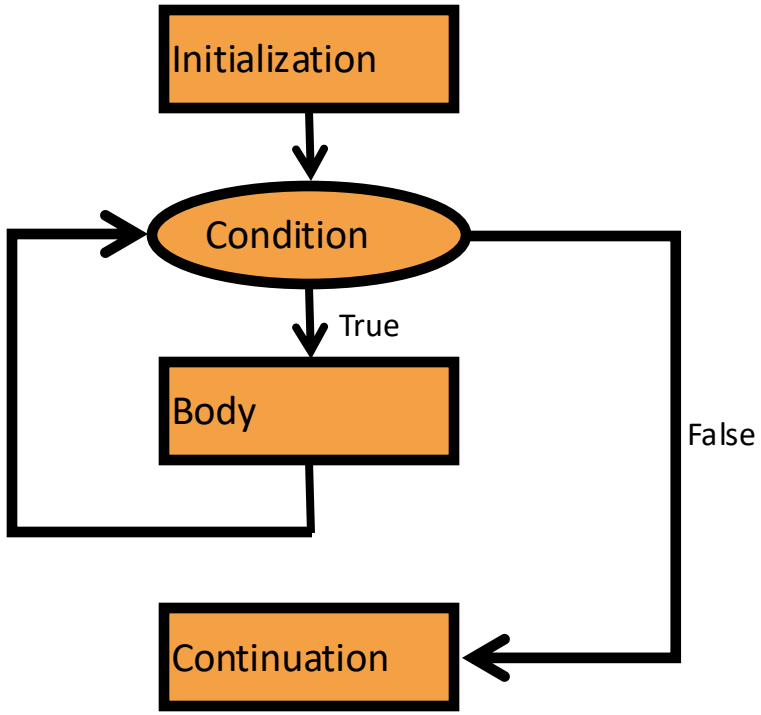
Syntax

```
<instructions_init>    # Initialization
while <condition> :    # Condition
  <instructions_body> # Body
<instructions_cont>    # Continuation
```



While loop

```
def countdown(counter):

    msg = ""

    while counter > 0:

        msg = msg + str(counter) + " "
        counter = counter - 1

    msg = msg + "GO!"
    return msg
```



If the condition is false the loop is not executed

If the execution enters the loop, the body should eventually render the condition false to allow exiting the loop

Function is_prime

How can we verify if a positive integer number is prime?

Function is_prime

How can we verify if a positive integer number is prime?

A simple (inefficient) algorithm is to check if it is divisible by any other number strictly between 1 and the given number we want to test.

Function is_prime

Let's implement the algorithm described as a function (is_prime) and test if it behaves as expected.

Write a program that accepts positive integer numbers given by the user and test them. To stop the program, the user should provide -1.

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    <initialization>
    while <condition>:
        <body>

    return <result>
```

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    <auxiliary_var_with_upper_limit>
    while <condition>:
        <body>

    return <result>
```

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    v = n - 1
    while <condition>:
        <body>

    return <result>
```

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    v = n - 1
    while <condition>:
        <body>

    return <did_v_reach_lower_limit>
```

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    v = n - 1
    while <condition>:
        <body>

    return v == 1
```

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    v = n - 1
    while <condition>:
        <decrement_v>

    return v == 1
```

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    v = n - 1
    while <condition>:
        v = v - 1

    return v == 1
```

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    v = n - 1
    while <n_not_divisible_by_v>:
        v = v - 1

    return v == 1
```

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    v = n - 1
    while n % v != 0:
        v = v - 1

    return v == 1
```

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    v = n - 1
    while n % v != 0:
        v = v - 1

    return v == 1
```

What if n is not positive?

Function is_prime

```
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    v = n - 1
    while (v > 1) and (n % v != 0):
        v = v - 1

    return v == 1
```

CMU Portugal Academy

Testing our is_prime implementation

```python
def is_prime(n):
    """is_prime : int [positive] -> bool
    Description: checks if n is prime.
    Examples: is_prime(3)->True; is_prime(4)->False
    """
    v = n - 1
    while (v > 1) and (n % v != 0):
        v = v - 1

    return v == 1
```

[24] ✓ 0.0s Python

```python
print(is_prime(7), is_prime(123), is_prime(1009))
```

[28] ✓ 0.0s Python

```
True False True
```

For Loop

Instruction **for** repeats a block of statements for each element of the sequence.

Syntax

```
for <variable> in <sequence>:
  <instruction>
```

Example

```
for i in s:
  print(i)
```

- If variable **i** does not exist, it is created.
- For each element **e** of sequence **s** (following the order of the elements in the sequence):
  1. **i** = **e**;
  2. Then execute the block of instructions within the loop (in the example, instruction print)

For Loop: Example

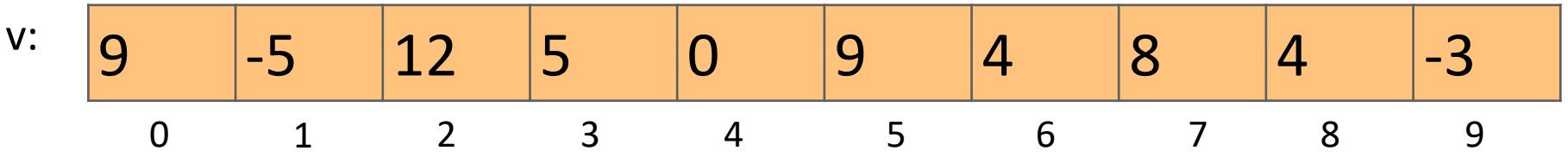
- Sequence defined by function **range**.

```
In [1]: for i in range(2, 6):
   ...:     print(i)
   ...:
2
3
4
5
```

Lists in Python

- Create a list and assign it to variable v:

  v = [9, -5, 12, 5, 0, 9, 4, 8, 4, -3]



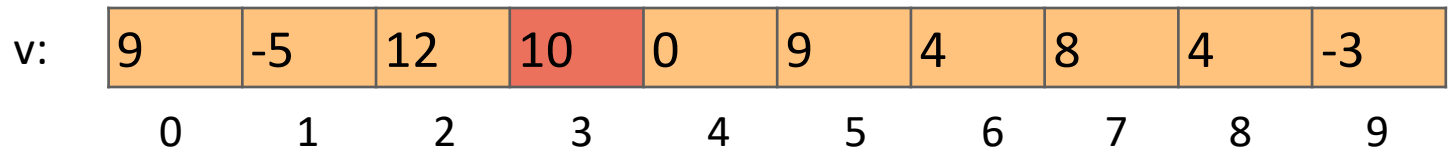
- Function len returns the size of v: len(v) is 10
- Position 3 of v: v[3] is 5

Updating a position (list)

Syntax for updating the contents of a position:

- variable_name[**expression**] = **expression**

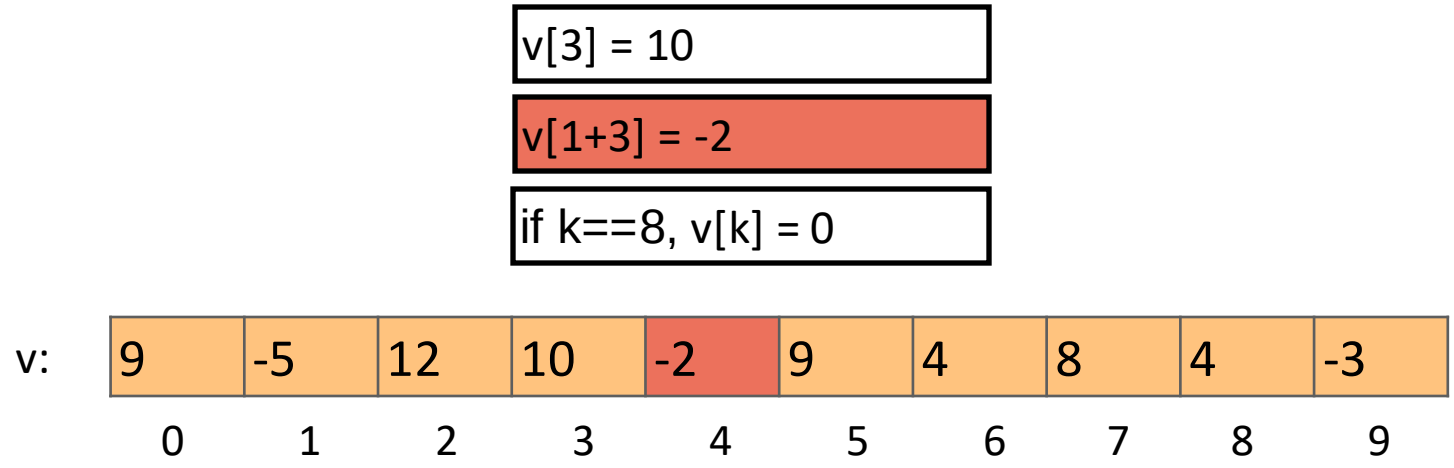
```
v[3] = 10
v[1+3] = -2
if k==8, v[k] = 0
```



Updating a position (list)

Syntax for updating the contents of a position:

- variable_name[**expression**] = **expression**



Typical indexing errors

```
v = [9, -5, 12, 5, 0, 9, 4, 8, 4, -3]

x = v[ math.sqrt(4) ]        # The index has to be an integer

y = v[12]                    # The index has to be  ≤ len(v)-1

v[12] = 4                    # The index has to be  ≤ len(v)-1
```

Slicing Lists

Lists can be sliced:

- *To slice elements within a range start_index and end_index (inclusive), we can use [start_index : end_index+1]*

A	B	C	D	F
0	1	2	3	4

```
>>> grades = ["A", "B", "C", "D", "F"]
>>> grades[1:3]
['B', 'C']
>>>
```

Slicing Lists

Lists can be sliced:

- *To slice elements from the beginning to a certain ending index (inclusive), we can use [: end_index+1]*

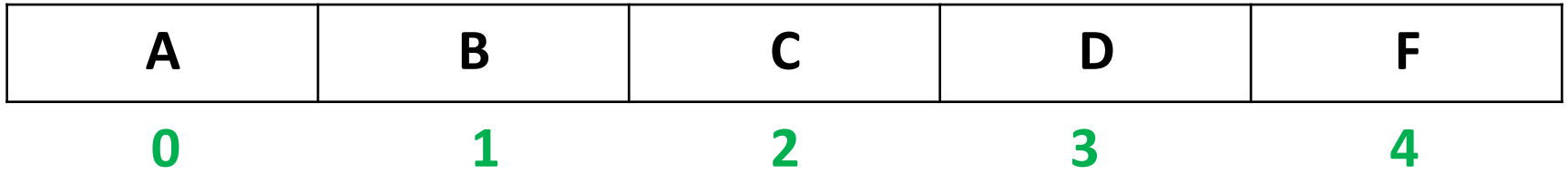
A	B	C	D	F
0	1	2	3	4

```
>>> grades = ["A", "B", "C", "D", "F"]
>>> grades[:3]
['A', 'B', 'C']
>>>
```

Slicing Lists

Lists can be sliced:

- *To slice elements from a certain starting index till the end, we can use [start_index:]*

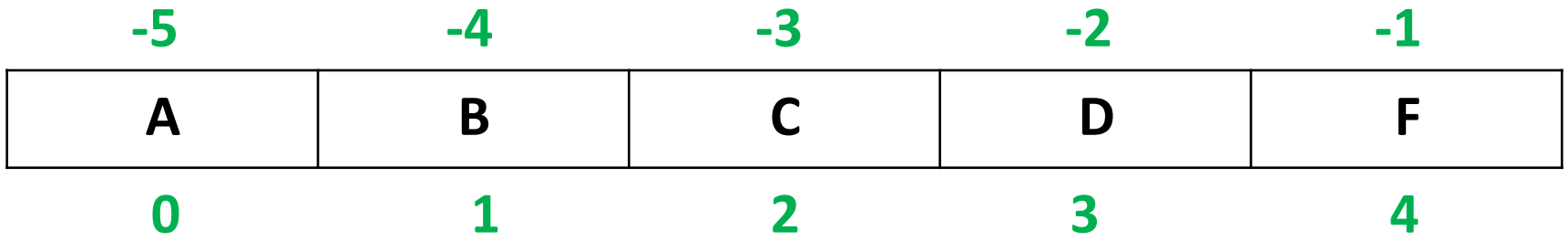



```
>>> grades = ["A", "B", "C", "D", "F"]
>>> grades[1:]
['B', 'C', 'D', 'F']
>>>
```

Slicing Lists

Lists can be sliced:

- *We can even pass a negative index, after which Python will add the length of the list to the index*

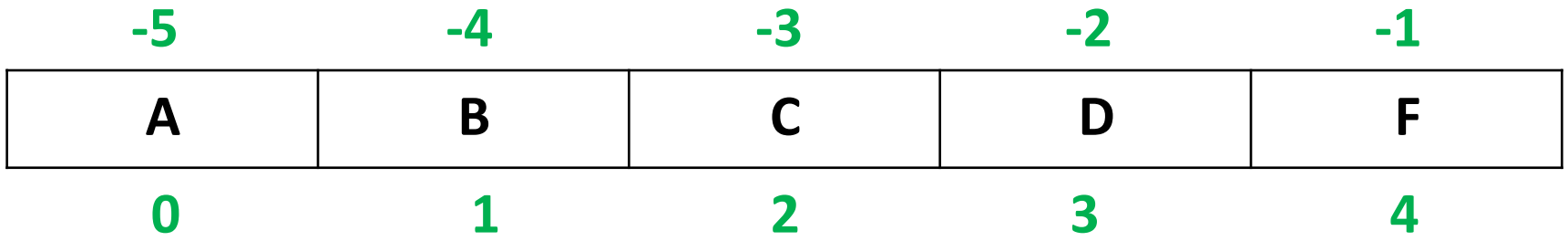



```
>>> grades = ["A", "B", "C", "D", "F"]
>>> grades[-1]
'F'
>>>
```

Slicing Lists

Lists can be sliced:

- *To slice elements from a certain ending index (inclusive) till the beginning, we can use [:-end_index+1]*

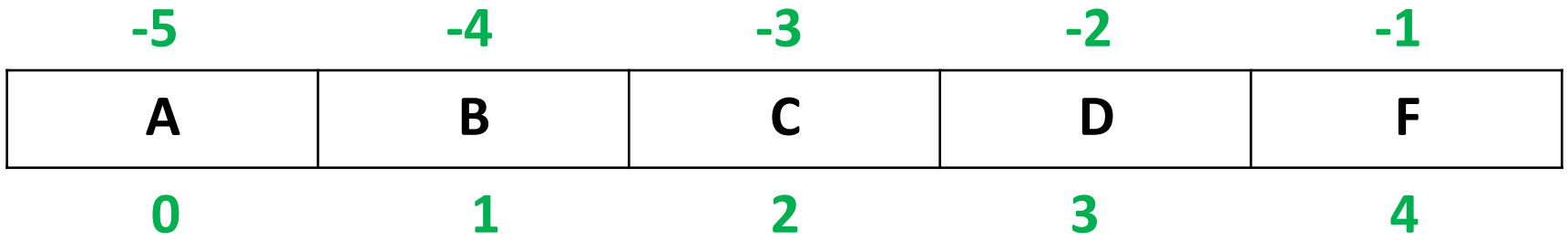



```
>>> grades = ["A", "B", "C", "D", "F"]
>>> grades[:-1]
['A', 'B', 'C', 'D']
>>>
```

Slicing Lists

Lists can be sliced:

- *To slice in steps, we can use [start_index, end_index+1, step] (idea applies to slicing from forward and backward)*




```
>>> grades = ["A", "B", "C", "D", "F"]
>>> grades[0:5:2]
['A', 'C', 'F']
>>>
```

Lists Are Not Sets

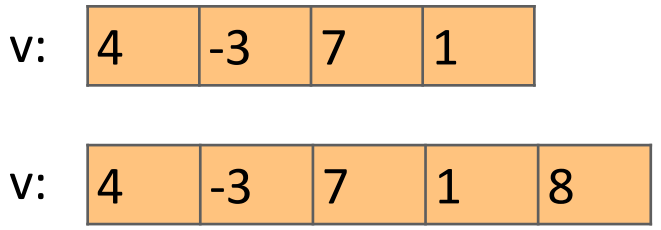
- Lists can have *duplicate* values

```
>>> myList = [5, 2, 2, 3]
>>> myList
[5, 2, 2, 3]
>>>
```

List: Functions append and extend

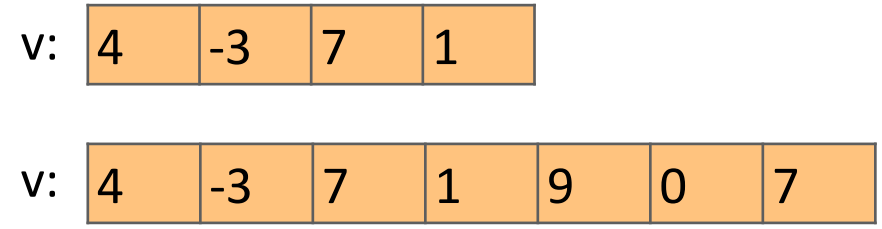
append adds an element to the end of the list:

```
>>> v = [4, -3, 7, 1]
>>> v.append(8)
```



extends adds a sequence of elements to the end of the list:

```
>>> v = [4, -3, 7, 1]
>>> v.extend([9, 0, 7])
```



List Functions

- Here is the set of functions that you can use with lists

Function	Description
L.append(x)	Add element x to the end of list L
L.extend(L2)	Add all elements of list L2 to the end of list L
L.insert(i, x)	Insert item x at the defined index i of list L
L.remove(x)	Removes item x from list L (valueError exception will be thrown if x does not exist)
L.pop(i)	Removes and returns the element at index i of list L. If no parameter is passed, the last item in L will be removed and returned

Python Reference – Lists

List Functions

- Here is the set of functions that you can use with lists

Function	Description
L.clear()	Removes all items from list L
L.index(x)	Returns the index of the first matched item x in list L
L.count(x)	Returns the count of times item x appears in list L
L.sort()	Sort items in list L in an ascending order
L.reverse()	Reverses the order of items in list L
L.copy()	Returns a copy of list L (*making any change to the returned list will not impact the original list L*)

Lists: Useful functions

- **v.index(n)** returns the index of the first occurrence of **n** in **v**
- **v.remove(n)** removes the first occurrence of value **n** in **v**
- **v.reverse(n)** inverts **v** (reverses the order of elements)
- **v.count(n)** counts the occurrences of value **n** in **v**

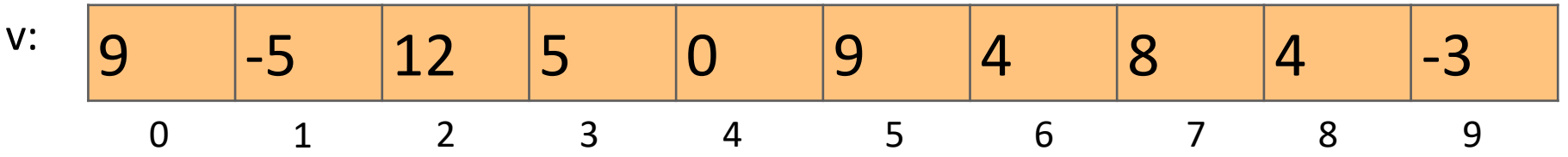
```
In  [1]: v = [10, 11, 12, 11, 14]
In  [2]: v.index(11)
Out [2]: 1
In  [3]: v.count(11)
Out [3]: 2
```

```
In  [4]: v.remove(11)
In  [5]: v
Out [5]: [10, 12, 11, 14]
In  [6]: v.reverse()
In  [7]: v
Out [7]: [14, 11, 12, 10]
```

Tuples in Python – Immutable!

- Create a list and assign it to variable v:

  v = (9, -5, 12, 5, 0, 9, 4, 8, 4, -3)



- Function len returns the size of v: len(v) is 10
- Position 3 of v: v[3] is 5
- We can't change the initial values.

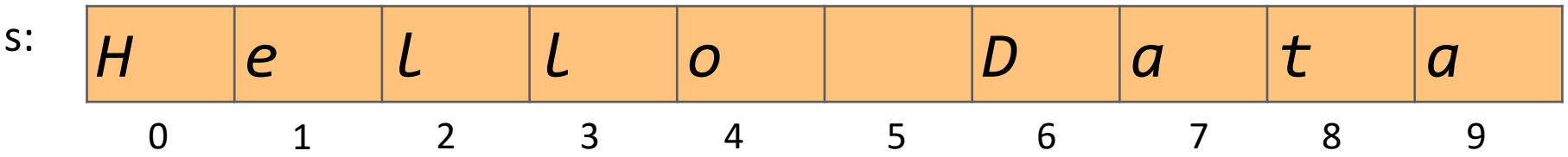
Tuples

- Like lists, tuples can:
  - Contain any and different types of elements
  - Contain duplicate elements (e.g., (1, 1, 2))
  - Be indexed exactly in the same way (i.e., using the [] brackets)
  - Be sliced exactly in the same way (i.e., using the [::] notation)
  - Be concatenated (e.g., t = (1, 2, 3) + (“a”, “b”, “c”))
  - Be repeated (e.g., t = (“a”, “b”) * 10)
  - Be nested (e.g., t = ((1, 2), (3, 4), ((“a”, “b”, ”c”), 3.4))
  - Be passed to a function, but will result in *pass-by-value* and not *pass-by-reference* outcome since it is immutable
  - Be iterated over

Strings: Sequences of characters

- Create a string and assign it to variable s:

  s = "Hello Data"



- Function len returns the length of s: len(s) is 10
- Position 2 of s: s[2] is "l"
- Cannot update the contents of a position of s.

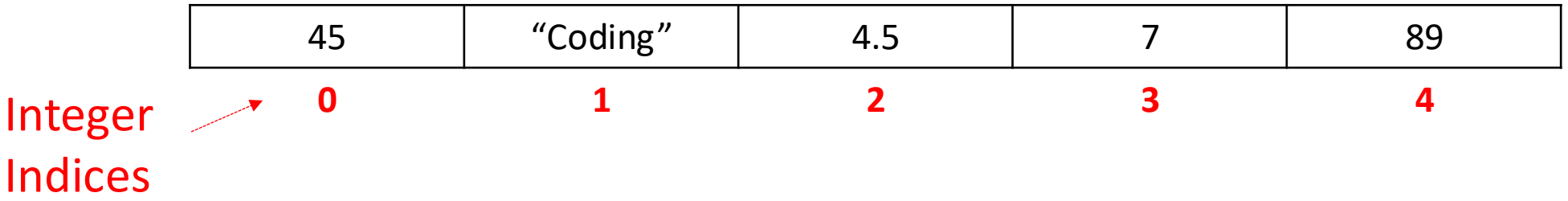
For loop: Example

Iterating over the elements of a String

```
In [1]: s = "Hello!"

In [2]: for i in s:

   ...:     print(i)

   ...:
H
e
l
l
o
!
```

Towards Dictionaries

- Lists and tuples hold elements with *only integer* indices



- So in essence, each element has an *index* (or a *key*) which can *only* be an integer, and a *value* which can be of any type (e.g., in the above list/tuple, the first element has key 0 and value 45)
  - *What if we want to store elements with non-integer indices (or keys)?*

Dictionaries

- We can use a dictionary to store elements with *keys of any types* and *values of any types* as well

keys of different types

Values of different types

- The above dictionary can be defined in Python as follows:

```
dic = {"NUM":45, 1000:"coding", 2000:4.5, 3.4:7, "XXX":89}
```

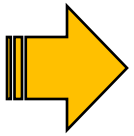
key *value*

key:*value* pairs separated by commas

Dictionaries

- Can contain any and types of elements (i.e., keys and values)
- Can contain only *unique* different keys but duplicate values

```
dic2 = {"a":1, "a":2, "b":2}
print(dic2)
```



Output: {'a': 2, 'b': 2}

The element “a”:2 will override the element “a”:1 because only ONE element can have key “a”

- Can be indexed *but only* through keys (i.e., dic2[“a”] will return 1
- dic2[0] will return an error since there is no element with key 0 in dic2 above)

Accessing an element

Python syntax to get the value associated with a **key**:

```
In [1]: balance = {"alice" : 12, "bob" : 23, "anne" : 12}
In [2]: type(balance)
Out[2]: dict
In [3]: balance["bob"]
Out[3]: 23
In [4]: balance["mary"]
Out[4]: Traceback (most recent call last):
  File "<ipython-input-19-cfcb43ec7e06>", line 1, in <module>
    balance["mary"]
KeyError: 'mary'
```

Accessing an element

Updating an element. If the element is in the dictionary, its value is updated. Otherwise, a new **key**:**value** pair is added to the dictionary.

```
In [1]: balance = {"alice" : 12, "bob" : 23, "anne" : 12}

In [2]: balance["mary"] = 20

In [3]: balance

Out[3]: {'alice': 12, 'bob': 23, 'anne': 12, 'mary': 20}

In [4]: balance["alice"] = 20

In [5]: balance

Out[5]: {'alice': 20, 'bob': 23, 'anne': 12, 'mary': 20}

In [6]: balance["mary"] = balance["mary"] + 4

In [7]: balance

Out[7]: {'alice': 20, 'bob': 23, 'anne': 12, 'mary': 24}
```

Operations over dictionaries

Check if a **key** is in the dictionary:

```
In [1]: balance = {"alice" : 12, "bob" : 23, "anne" : 12}

In [2]: "alice" in balance

Out[2]: True

In [3]: "john" in balance

Out[3]: False

In [4]: "john" not in balance

Out[4]: True
```

Operations over dictionaries

pop removes the element associated to the key and returns the value.

```
In [1]: balance = {"alice" : 12, "bob" : 23, "anne" : 12}

In [2]: balance.pop("alice")

Out[2]: 12

In [3]: balance

Out[3]: {"bob" : 23, "anne" : 12}

In [4]: balance["alice"]

Out[4]: Traceback (most recent call last):

  File "<ipython-input-19-e74b4fdfdea2>", line 1, in <module>

    balance["alice"]

KeyError: 'alice'
```

Operations over dictionaries

FOR Iterates over the **keys** of a dictionary:

```
In [1]: balance = {"alice" : 12, "bob" : 23, "anne" : 12}
In [2]: for key in balance:
   ...:     print(key)
   ...:
alice
bob
anne
```

Views over a dictionary

- **keys()** returns a view over the keys.
- **values()** returns a view over the values.
- **items()** returns a view over the pairs key:value.

```
In [1]: balance = {"alice" : 12, "bob" : 23, "anne" : 12}
In [2]: balance.keys()
Out[2]: dict_keys(['alice', 'bob', 'anne'])
In [3]: balance.values()
Out[3]: dict_values([12, 23, 12])
In [4]: balance.items()
Out[4]: dict_items([('alice',12), ('bob',23), ('anne',12)])
```

```
balance = {"alice" : 12, "bob" : 23, "anne" : 12 }
```

```
In [1]: for k in balance.keys():
   ...:     print(k)
   ...:
alice
bob
anne
```

```
In [1]: for k in balance.values():
   ...:     print(k)
   ...:
12
23
12
```

```
In [1]: for k, v in balance.items():
   ...:     print(k, v)
   ...:
alice 12
bob 23
anne 12
```

Views over a dictionary

```
In [1]: balance = {"alice" : 12, "bob" : 23, "anne" : 12}

In [2]: list_keys = list(balance.keys())

In [3]: list_keys
Out[3]: ['alice', 'bob', 'anne']

In [4]: list_values = list(balance.values())

In [5]: list_values
Out[5]: [12, 23, 12]

In [6]: list_items = list(balance.items())

In [7]: list_items
Out[7]: [('alice', 12), ('bob', 23), ('anne', 12)]
```

Dictionary Functions

- Many other functions can also be used with dictionaries

Function	Description
dic.clear()	Removes all the elements from dictionary dic
dic.copy()	Returns a copy of dictionary dic
dic.items()	Returns a list containing a tuple for each key-value pair in dictionary dic
dic.get(k)	Returns the value of the specified key k from dictionary dic
dic.keys()	Returns a list containing all the keys of dictionary dic
dic.pop(k)	Removes the element with the specified key k from dictionary dic

Dictionary Functions

- Many other functions can also be used with dictionaries

Function	Description
dic.popitem()	Removes the last inserted key-value pair in dictionary dic
dic.values()	Returns a list of all the values in dictionary dic

Hands-On Session!

Course Shared Folder

CMU Portugal Academy

Thank you!

Carnegie Mellon Portugal | TÉCNICO LISBOA | Ciências ULisboa | NOVA SCHOOL OF SCIENCE & TECHNOLOGY

This slides contain adaptations from the following references:

- Advanced Programming for Biology Course, FCT-NOVA
- 15-110: Principles of Computing course, CMU

CMU Portugal Academy

CMU Portugal
Advanced Training Program

Foundations of Data Science

DAVID SEMEDO

RAFAEL FERREIRA

NOVA SCHOOL OF SCIENCE AND TECHNOLOGY

Carnegie Mellon Portugal · TÉCNICO LISBOA · Ciências ULisboa · NOVA SCHOOL OF SCIENCE & TECHNOLOGY

Email: rah.ferreira@campus.fct.unl.pt

David Semedo– Short bio

df.semedo@fct.unl.pt

Assistant Professor @ NOVA FCT, Integrated Researcher @ NOVA LINCS

AI for vision and language. Neural approaches to conversational and contextualized media understanding.

Rafael Ferreira– Short bio

rah.ferreira@campus.fct.unl.pt

4th year PhD Student @ NOVA FCT

Conversational AI. Team leader of the award-winning TWIZ in the Alexa TaskBot Challenge.

Today's Topics

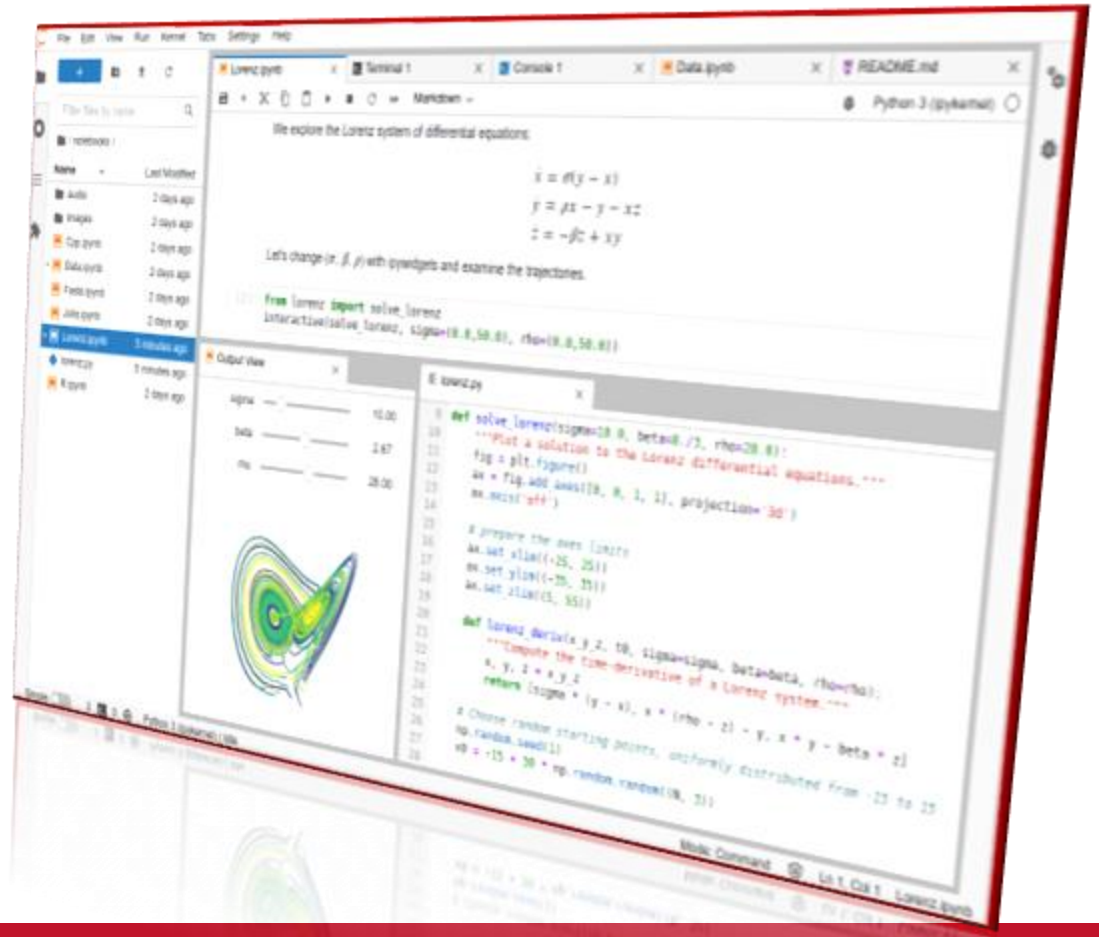
1. INTRODUCTION TO NUMPY
2. NUMPY ARRAYS
3. ARRAY DIMENSIONS
4. ARRAY INDEXING AND SLICING
5. ARRAY MANIPULATION
6. AGGREGATION FUNCTIONS
7. FILTERING
8. SORTING
9. RESHAPING AND TRANSPOSING
10. BROADCASTING
11. DOT PRODUCT
12. HANDLING MISSING VALUES

Laboratory setup

Course Shared Folder

We will use your bootcamp setup!

ANACONDA® + jupyter



https://wiki.novasearch.org/wiki/lab_setup

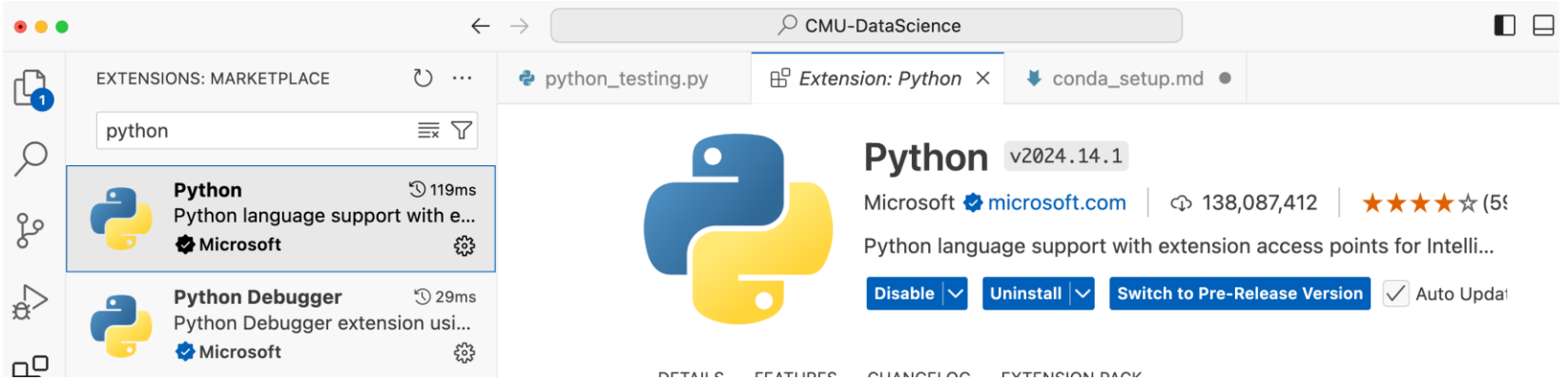
Laboratory setup – VS Code

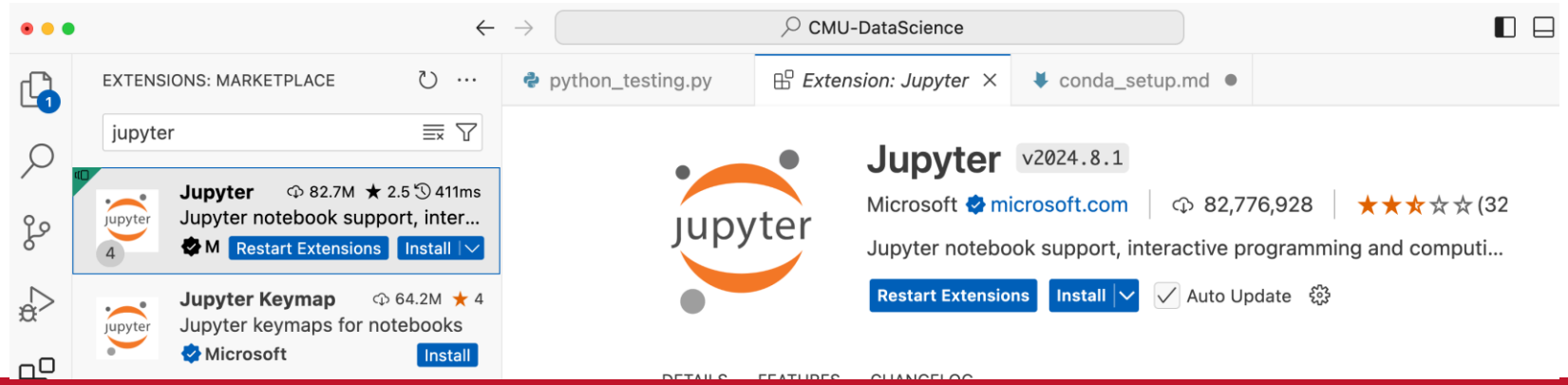
VS Code

Download VS Code

Install two extensions:

- Python
- Jupyter (Docs)





CMU Portugal Academy

03

Numerical Operations

Carnegie Mellon Portugal · TÉCNICO LISBOA · Ciências ULisboa · NOVA SCHOOL OF SCIENCE & TECHNOLOGY

What is NumPy?

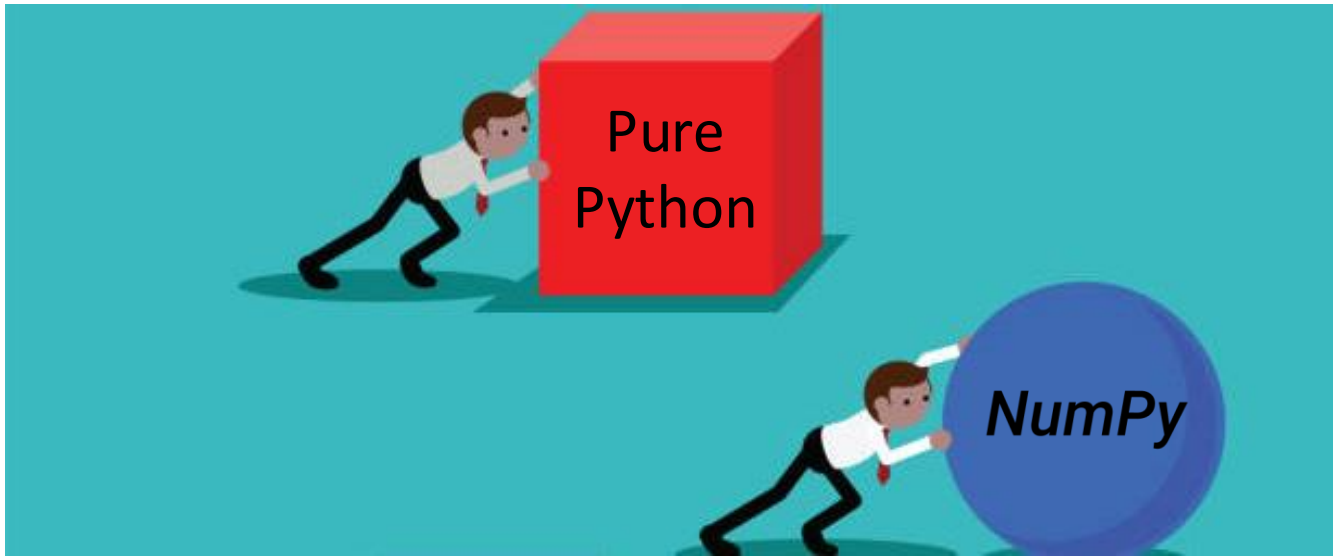
Numerical**Py**thon

Opensource library for working with **arrays** and performing numerical operations.

NumPy

Why use NumPy?

Faster, more efficient, easier syntax, and allows operations over **arrays**.



Array Operations

Why use NumPy?

Faster, more efficient, easier syntax, and allows operations over **arrays**.

```python
# Pure Python implementation
list1 = [i for i in range(10)]
list2 = [i for i in range(10)]

# Summing the two lists element-wise
result = []
for i in range(len(list1)):
    result.append(list1[i] + list2[i])

# Print the first 10 elements of the result
print(result)
```

✓ 0.0s

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

```python
import numpy as np

# Numpy implementation
array1 = np.arange(10)
array2 = np.arange(10)

# Summing the arrays element-wise
result = array1 + array2

# Print the first 10 elements of the result
print(result)
```

✓ 0.0s

```
[ 0  2  4  6  8 10 12 14 16 18]
```

NumPy and ndarray

Creating a NumPy **ndarray**:

```
import numpy as np

list_data = [1, 2, 3, 4, 5]
print("List:", list_data, type(list_data))

array_data = np.array(list_data)
print("Array:", array_data, type(array_data))
```

✓ 0.0s

```
List: [1, 2, 3, 4, 5] <class 'list'>
Array: [1 2 3 4 5] <class 'numpy.ndarray'>
```

NumPy and ndarray

Creating **other types** of NumPy arrays:

```
# creating different types of arrays

array_1d = np.array([1, 2, 3, 4, 5])  # 1D array

array_zeros = np.zeros(5)  # array filled with zeros

array_ones = np.ones(5)  # array filled with ones

array_sevens = np.full(5, 7)  # array filled with sevens


print("1D array:", array_1d)
print("Array filled with zeros:", array_zeros)
print("Array filled with ones:", array_ones)
print("Array filled with sevens:", array_sevens)
```

✓ 0.0s

```
1D array: [1 2 3 4 5]
Array filled with zeros: [0. 0. 0. 0. 0.]
Array filled with ones: [1. 1. 1. 1. 1.]
Array filled with sevens: [7 7 7 7 7]
```

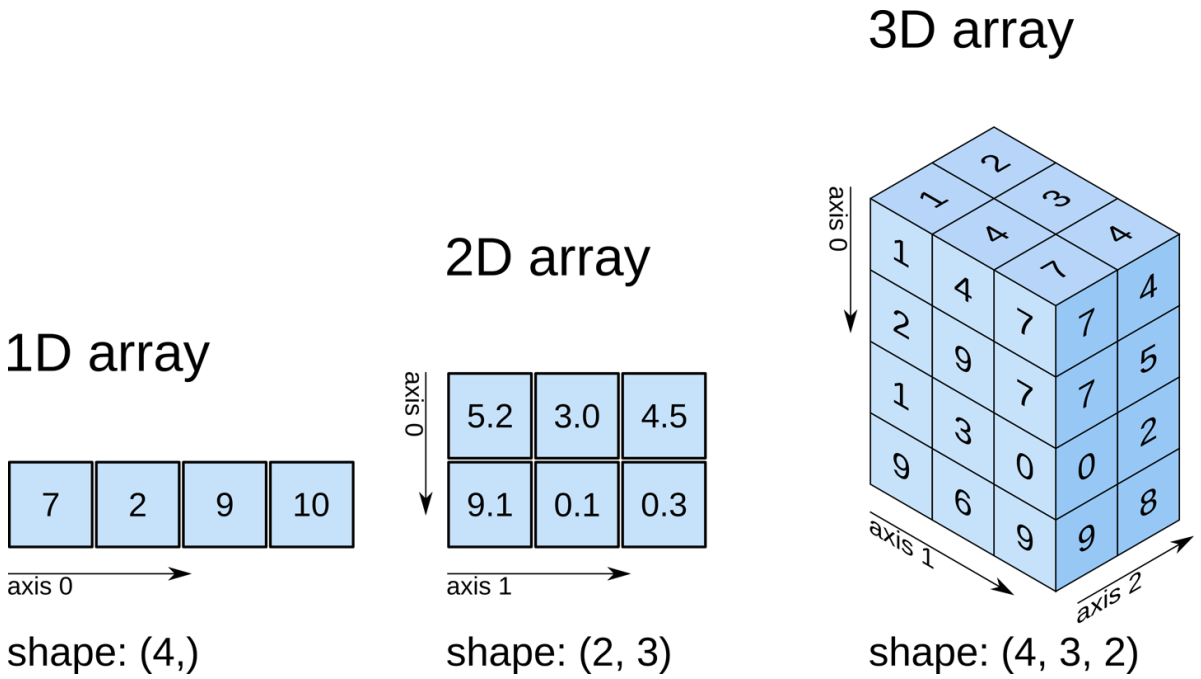
```
array_range = np.arange(0, 10, 1)  # range

array_linspace = np.linspace(0, 10, 5) # linearly spaced

array_random = np.random.rand(5)  # random numbers

print("Range:", array_range)
print("Linearly spaced:", array_linspace)
print("Random numbers:", array_random)
```

✓ 0.0s

```
Range: [0 1 2 3 4 5 6 7 8 9]
Linearly spaced: [ 0.   2.5  5.   7.5 10. ]
Random numbers: [0.81629728 0.72492204 0.63652672 0.91011596 0.16713425]
```

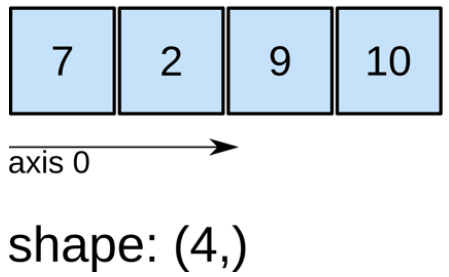
Checking dimensions:

Arrays in NumPy can have various **dimensions**:



Checking dimensions:

Arrays in NumPy can have various **dimensions**: array.shape



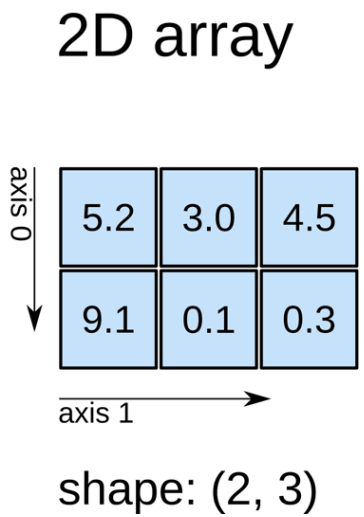

```
# showing the dimensions of an array
array = np.array([7, 2, 9, 10])
print("1D array:\n", array)
print("Number of elements in the array:", array.size)
print("Dimensions of the array:", array.ndim)
print("Shape of the array:", array.shape)
```

✓ 0.0s

```
1D array:
 [ 7  2  9 10]
Number of elements in the array: 4
Dimensions of the array: 1
Shape of the array: (4,)
```

Checking dimensions:

Arrays in NumPy can have various **dimensions**: array.shape




```
array = np.array([[5.2, 3.0, 4.5], [9.1, 0.1, 0.3]])
print("2D array:\n", array)
print("Number of elements in the array:", array.size)
print("Dimensions of the array:", array.ndim)
print("Shape of the array:", array.shape)
```

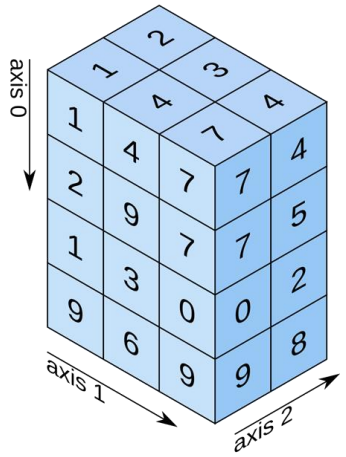
✓ 0.0s

```
2D array:
 [[5.2 3.  4.5]
 [9.1 0.1 0.3]]
Number of elements in the array: 6
Dimensions of the array: 2
Shape of the array: (2, 3)
```

Checking dimensions:

Arrays in NumPy can have various **dimensions**: array.shape

3D array



shape: (4, 3, 2)

```python
# 3D array
array = np.array([[[1, 2], [3, 4], [5, 6]],
                  [[7, 8], [9, 10], [11, 12]],
                  [[13, 14], [15, 16], [17, 18]],
                  [[19, 20], [21, 22], [23, 24]]
                  ])
print("3D array:\n", array)
print("Number of elements in the array:", array.size)
print("Dimensions of the array:", array.ndim)
print("Shape of the array:", array.shape)
```

✓ 0.0s

```
3D array:
 [[[ 1  2]
  [ 3  4]
  [ 5  6]]

 [[ 7  8]
  [ 9 10]
  [11 12]]

 [[13 14]
  [15 16]
  [17 18]]

 [[19 20]
  [21 22]
  [23 24]]]
Number of elements in the array: 24
Dimensions of the array: 3
Shape of the array: (4, 3, 2)
```

Array Indexing

Just like lists NumPy arrays can be accessed by **index**:

```
# accessing index of each element
array_1d = np.array([1, 2, 3, 4, 5])

print("First element:", array_1d[0])
print("Second element:", array_1d[1])
print("Last element:", array_1d[-1])
```

✓ 0.0s

```
First element: 1
Second element: 2
Last element: 5
```

```
# 2D array
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6]])

print("First row and second column:", array_2d[0, 1])
print("Last row and first column:", array_2d[-1, 0])
```

✓ 0.0s

```
First row and second column: 2
Last row and first column: 4
```

Array Slicing

Just like python lists NumPy arrays can be **sliced**: [start:end]

```
# examples of array slicing in 1D array
array = np.array([1, 2, 3, 4, 5])

# get index 1
print("array[1] ->", array[1])

# slice from index 1 to 3 (exclusive)
print("array[1:3] ->", array[1:3])

# slice from index 0 to 4 (exclusive)
print("array[:4] ->", array[:4])
```

✓ 0.0s

```
array[1] -> 2
array[1:3] -> [2 3]
array[:4] -> [1 2 3 4]
```

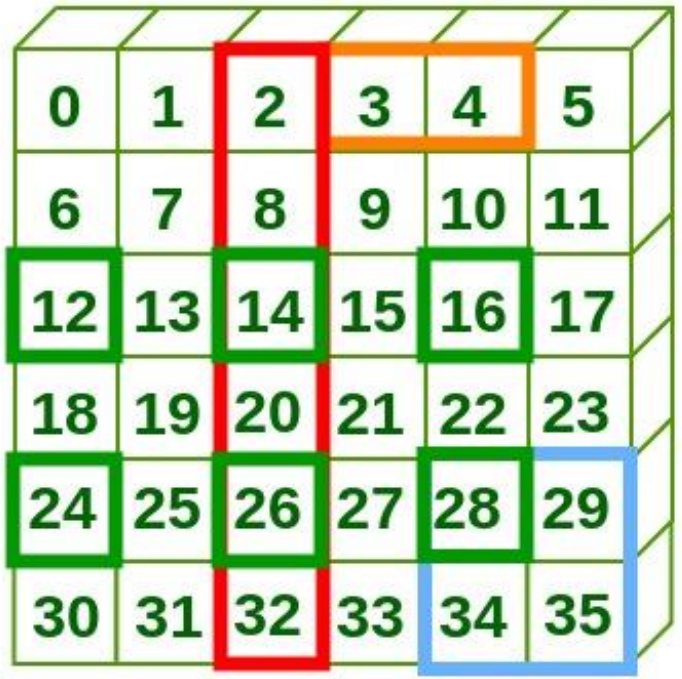
```
# examples of array slicing in 1D array
array = np.array([1, 2, 3, 4, 5])

# slice from index 1 to the end
print("array[1:] ->", array[1:])

# slice from the beginning to the end
print("array[:] ->", array[:])

# slice with a step of 2
print("array[::2] ->", array[::2])
```

✓ 0.0s

```
array[1:] -> [2 3 4 5]
array[:] -> [1 2 3 4 5]
array[::2] -> [1 3 5]
```

Array Slicing in 2D

```
>>> a[0,3:5]
array( [3,4] )

>>> a[4:, 4:]
array( [ 28, 29],
       [ 34, 35] ] )

>>> a[ :, 2]
array( [2, 8, 14, 20, 26, 32] )

>>> a[2 : : 2, : : 2]
array( [ 12, 14, 16],
       [ 24, 26, 28] ] )
```



Hands-On Session!

Indexing and Slicing

NumPy Data Types

Character	Data Type
i	integer
b	boolean
u	Unsigned integer
f	float
c	Complex float
m	timedelta
M	datetime
O	object
S	string
U	Unicode string
V	Fixed chunk of memory for other type (void)

```python
# examples of various data types
array = np.array([1, 2, 3, 4, 5])
print("Default data type:", array.dtype)

array = np.array([1.1, 2.2, 3.3, 4.4, 5.5])
print("Default data type:", array.dtype)

array = np.array([1, 2, 3, 4, 5], dtype=np.float64)
print("Custom data type:", array.dtype)

array = np.array([1, 2, 3, 4, 5], dtype=np.int32)
print("Custom data type:", array.dtype)

array = np.array([1, 2, 3, 4, 5], dtype=np.int16)
print("Custom data type:", array.dtype)

# str
array = np.array(["apple", "banana", "cherry"])
print("Default data type:", array.dtype)

# bool
array = np.array([True, False, True])
print("Default data type:", array.dtype)
```

✓ 0.0s

```
Default data type: int64
Default data type: float64
Custom data type: float64
Custom data type: int32
Custom data type: int16
Default data type: <U6
Default data type: bool
```

NumPy Converting to Other Types

Conver to another type using **astype()** function.

```python
# converting data types
array = np.array([1, 2, 3, 4, 5])
print("Original data:", array, array.dtype)

# converting from int to float
array = array.astype(np.float64)
print("New data:", array, array.dtype)
```

✓ 0.0s

```
Original data type: [1 2 3 4 5] int64
New data type: [1. 2. 3. 4. 5.] float64
```

```python
# converting from float to int
array = np.array([1.1, 2.2, 3.3, 4.4, 5.5])
print("Original data:", array, array.dtype)

array = array.astype(np.int32)
print("New data:", array, array.dtype)
```

✓ 0.0s

```
Original data: [1.1 2.2 3.3 4.4 5.5] float64
New data: [1 2 3 4 5] int32
```

```python
# converting from int to str
array = np.array([1, 2, 3, 4, 5])
print("Original data:", array, array.dtype)

array = array.astype(str)
print("New data:", array, array.dtype)
```

✓ 0.0s

```
Original data: [1 2 3 4 5] int64
New data: ['1' '2' '3' '4' '5'] <U21
```

NumPy Converting to Other Types

Applying operations between int and float results in a float.

```
1  # int summing with float
2  array_int = np.array([1, 2, 3, 4, 5])
3  array_float = np.array([1.1, 2.2, 3.3, 4.4, 5.5])
4
5  result = array_int + array_float
6  print("Result:", result, result.dtype)
```

✓ 0.0s

```
Result: [ 2.1  4.2  6.3  8.4 10.5] float64
```

NumPy Converting to Other Types

Applying operations between int and str without converting results in an error.

```
1  # int summing with str
2  array_int = np.array([1, 2, 3, 4, 5])
3  array_str = np.array(["1", "2", "3", "4", "5"])
4
5  result = array_int + array_str  # this will result in an error!
6  print("Result:", result, result.dtype)
```

⊗ 0.0s

```
---------------------------------------------------------------------------
UFuncTypeError                            Traceback (most recent call last)
Cell In[57], line 5
      2 array_int = np.array([1, 2, 3, 4, 5])
      3 array_str = np.array(["1", "2", "3", "4", "5"])
----> 5 result = array_int + array_str
      6 print("Result:", result, result.dtype)

UFuncTypeError: ufunc 'add' did not contain a loop with signature matching types (dtype('int64'), dtype('<U1'))
```

NumPy Operations over Arrays

Element-wise operations considering an array.

```
# Element-wise operations
arr = np.array([1, 2, 3, 4])
arr_sum = arr + 10  # Add 10 to each element
arr_square = arr ** 2  # Square each element

print("Original Array:\n", arr)
print("\nArray after adding 10:\n", arr_sum)
print("\nArray after squaring:\n", arr_square)
```

✓ 0.0s

```
Original Array:
 [1 2 3 4]

Array after adding 10:
 [11 12 13 14]

Array after squaring:
 [ 1  4  9 16]
```

NumPy Operations over Arrays

Operations over arrays work **element-wise**.

Sizes must match.

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

addition = arr1 + arr2
subtraction = arr1 - arr2

print("Array 1:", arr1)
print("Array 2:", arr2)
print("\nAddition:", addition)
print("Subtraction:", subtraction)
```

✓ 0.0s

```
Array 1: [1 2 3]
Array 2: [4 5 6]

Addition: [5 7 9]
Subtraction: [-3 -3 -3]
```

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

mult = arr1 * arr2
division = arr1 / arr2

print("Array 1:", arr1)
print("Array 2:", arr2)
print("Mult:", mult)
print("Division:", division)
```

✓ 0.0s

```
Array 1: [1 2 3]
Array 2: [4 5 6]
Mult: [ 4 10 18]
Division: [0.25 0.4  0.5 ]
```

NumPy Operations over 2D-Arrays

Element-wise operations considering a 2D-array.

```
# element-wise operations with 2D arrays
arr = np.array([[1, 2], [3, 4]])
arr_sum = arr + 10  # Add 10 to each element

print("Original Array:\n", arr)
print("\nArray after adding 10:\n", arr_sum)
```

✓ 0.0s

```
Original Array:
 [[1 2]
 [3 4]]

Array after adding 10:
 [[11 12]
 [13 14]]
```

NumPy Operations over 2D-Arrays

Operations over arrays work **element-wise**.

Sizes must match.

```python
# element-wise operations with 2D arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

addition = arr1 + arr2
subtraction = arr1 - arr2

print("Array 1:\n", arr1)
print("Array 2:\n", arr2)
print("\nAddition:\n", addition)
print("Subtraction:\n", subtraction)
```

✓ 0.0s

```
Array 1:
 [[1 2]
 [3 4]]
Array 2:
 [[5 6]
 [7 8]]

Addition:
 [[ 6  8]
 [10 12]]
Subtraction:
 [[-4 -4]
 [-4 -4]]
```

```python
# element-wise operations with 2D arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

mult = arr1 * arr2
division = arr1 / arr2

print("Array 1:\n", arr1)
print("Array 2:\n", arr2)
print("\nMult:\n", mult)
print("Division:\n", division)
```

✓ 0.0s

```
Array 1:
 [[1 2]
 [3 4]]
Array 2:
 [[5 6]
 [7 8]]

Mult:
 [[ 5 12]
 [21 32]]
Division:
 [[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

NumPy Mathematical Functions

NumPy defines various operations:

- max, min, sum, median, average, standard deviation, etc
- These **aggregate** the values over the array

```python
# Aggregation functions
arr = np.array([[1, 2, 3], [4, 5, 6]])
total_sum = np.sum(arr)  # Sum of all elements
column_sum = np.sum(arr, axis=0)  # Sum along columns
min_value = np.min(arr)  # Minimum value in the array
arg_min = np.argmin(arr)  # Index of minimum value
max_value = np.max(arr)  # Maximum value in the array
arg_max = np.argmax(arr)  # Index of maximum value

print("Array:\n", arr)
print("Total sum of array elements:", total_sum)
print("Column-wise sum:", column_sum)
print("Minimum value in array:", min_value)
print("Index of minimum value:", arg_min)
print("Maximum value in array:", max_value)
print("Index of maximum value:", arg_max)
```

✓ 0.0s

```
Array:
 [[1 2 3]
 [4 5 6]]
Total sum of array elements: 21
Column-wise sum: [5 7 9]
Minimum value in array: 1
Index of minimum value: 0
Maximum value in array: 6
Index of maximum value: 5
```

NumPy Mathematical Functions

NumPy defines various operations:

- max, min, sum, median, average, standard deviation, etc
- These **aggregate** the values over the array

```python
# Aggregation functions
arr = np.array([[1, 2, 3], [4, 5, 6]])
mean_value = np.mean(arr)  # Mean of all elements
median_value = np.median(arr)  # Median of all elements
mean = np.mean(arr, axis=0)  # Mean along columns
std_value = np.std(arr)  # Standard deviation of all elements

print("Array:\n", arr)
print("Mean of array elements:", mean_value)
print("Mean along columns:", mean)
print("Median of array elements:", median_value)
print("Standard deviation of array elements:", round(std_value, 3))
```

✓ 0.0s

```
Array:
 [[1 2 3]
 [4 5 6]]
Mean of array elements: 3.5
Mean along columns: [2.5 3.5 4.5]
Median of array elements: 3.5
Standard deviation of array elements: 1.708
```

NumPy Filtering

1. Define a **condition** for which to filter which creates a **mask**

2. Apply the **mask** to the original array to get the **elements**

```
# Boolean masking and conditional selection
arr = np.array([10, 20, 30, 40, 50])
mask = arr > 30  # Create a boolean mask where values are greater than 30
selected_elements = arr[mask]  # Use the mask to select elements

print("Original array:\n", arr)
print("\nBoolean mask (arr > 30):\n", mask)
print("\nSelected elements (arr[mask]):\n", selected_elements)
```

✓ 0.0s

```
Original array:
 [10 20 30 40 50]

Boolean mask (arr > 30):
 [False False False  True  True]

Selected elements (arr[mask]):
 [40 50]
```

NumPy Filtering – np.where

1. Define a **condition** for which to filter which creates a mask

2. Apply the **mask** to the original array to get the **elements**

```
# using np.where() to select elements based on condition
arr = np.array([10, 20, 30, 40, 50])
selected_elements = np.where(arr > 30)  # Select elements where value > 30

print("Original array:\n", arr)
print("\nSelected elements (arr > 30):\n", selected_elements)
print("\nSelected elements:\n", arr[selected_elements])
```

✓ 0.0s

```
Original array:
 [10 20 30 40 50]

Selected elements (arr > 30):
 (array([3, 4], dtype=int64),)

Selected elements:
 [40 50]
```

NumPy Filtering – Chaining Operations

Apply **multiple operations**.

Combine with logical conditions:

- and - &
- or - |

```
# chaining multiple conditions
arr = np.array([10, 20, 30, 40, 50])
# Select elements where 20 < value < 50
selected_elements = arr[(arr > 20) & (arr < 50)]

print("Original array:\n", arr)
print("\nSelected elements (20 < arr < 50):\n", selected_elements)

# logical or operation
# Select elements where value < 20 or value > 40
selected_elements = arr[(arr < 20) | (arr > 40)]
print("\nSelected elements (arr < 20 or arr > 40):\n", selected_elements)
```

✓ 0.0s

```
Original array:
 [10 20 30 40 50]

Selected elements (20 < arr < 50):
 [30 40]

Selected elements (arr < 20 or arr > 40):
 [10 50]
```

NumPy Sort

Sorts the array in **ascending** order of its values.

```
# example of sorting
array = np.array([3, 1, 2, 4, 5])

# Sort the array
sorted_array = np.sort(array)
print("Original array:", array)
print("Sorted array:", sorted_array)
```

✓ 0.0s

```
Original array: [3 1 2 4 5]
Sorted array: [1 2 3 4 5]
```

NumPy Sort

Sorts the array in **ascending** order of its values.

In 2D it sorts by the index passed.

```
# example of sorting a 2D array
array = np.array([[3, 4, 5], [1, 3, 2]])

# Sort the array along the rows
sorted_array_rows = np.sort(array, axis=0)
print("Original array:\n", array)
print("Sorted array along the rows:\n", sorted_array_rows)

# Sort the array along the columns
sorted_array_columns = np.sort(array, axis=1)
print("Sorted array along the columns:\n", sorted_array_columns)
```

✓ 0.0s

```
Original array:
 [[3 4 5]
 [1 3 2]]
Sorted array along the rows:
 [[1 3 2]
 [3 4 5]]
Sorted array along the columns:
 [[3 4 5]
 [1 2 3]]
```

NumPy Argsort

This function **gets the indices** that when applied would sort the array.

```
# argsort() function
array = np.array([3, 1, 2, 4, 5])

# Get the indices that would sort the array
indices = np.argsort(array)
print("Original array:", array)
print("Indices of sorted array:", indices)

# sort using the indices
sorted_array = array[indices]
print("Sorted array:", sorted_array)
```

✓ 0.0s

```
Original array: [3 1 2 4 5]
Indices of sorted array: [1 2 0 3 4]
Sorted array: [1 2 3 4 5]
```

Hands-On Session!

The Students

NumPy Flatten

Flatten converts a multi-dimensional array into a one-dimensional array.

Combines all elements into a **single row or column**.

```
# flattening a 2D array
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
flattened_arr = arr2d.flatten()

print("Original 2D array:\n", arr2d)
print("\nFlattened array:\n", flattened_arr)
```

✓ 0.0s

```
Original 2D array:
 [[1 2 3]
 [4 5 6]
 [7 8 9]]

Flattened array:
 [1 2 3 4 5 6 7 8 9]
```

NumPy Reshape

Reshape is used to change the shape of an array without altering its data.

It allows you to **rearrange the elements** into a new configuration.

Total number of elements must match.

```
# Reshaping
arr = np.array([[1, 2, 3], [4, 5, 6]])
reshaped = arr.reshape(3, 2)  # Reshape 2x3 array to 3x2

print("Original array:\n", arr)
print("\nReshaped array (3x2):\n", reshaped)
```

0.0s

```
Original array:
 [[1 2 3]
 [4 5 6]]

Reshaped array (3x2):
 [[1 2]
 [3 4]
 [5 6]]
```

NumPy Transpose

Transposing a matrix involves **swapping the rows and columns**.

Shape also changes.

Transposing can be done by:

- arr.T
- np.transpose(arr)

```
# Transposing arrays
arr = np.array([[1, 2, 3],
                [4, 5, 6]])
transposed = arr.T  # Transpose the array

print("Original array:\n", arr)
print("Shape of original array:", arr.shape)
print("\nTransposed array:\n", transposed)
print("Shape of transposed array:", transposed.shape)
```

✓ 0.0s

```
Original array:
 [[1 2 3]
 [4 5 6]]
Shape of original array: (2, 3)

Transposed array:
 [[1 4]
 [2 5]
 [3 6]]
Shape of transposed array: (3, 2)
```

NumPy Broadcasting

Broadcasting allows you to perform **element-wise operations on arrays of different shapes**.

Applies (broadcasts) the operation to each element.

```
# Broadcasting example
arr = np.array([[1, 2, 3], [4, 5, 6]])
vector = np.array([1, 2, 3])
broadcasted_sum = arr + vector  # Adds vector to each row of arr

print("Original array:\n", arr)
print("\nVector:\n", vector)
print("\nArray after broadcasting:\n", broadcasted_sum)
```

✓ 0.0s

```
Original array:
 [[1 2 3]
 [4 5 6]]

Vector:
 [1 2 3]

Array after broadcasting:
 [[2 4 6]
 [5 7 9]]
```

NumPy Joining arrays - Vertically

Joining arrays vertically - **vstack**:

- Appends the arrays **one below the other**.
- The arrays must have the same number of columns.

```
# Stacking and concatenating arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
vertical_stack = np.vstack((arr1, arr2))  # Stack arrays vertically

print("Array 1:\n", arr1)
print("\nArray 2:\n", arr2)
print("\nVertically stacked:\n", vertical_stack)
```

✓ 0.0s

```
Array 1:
 [1 2 3]

Array 2:
 [4 5 6]

Vertically stacked:
 [[1 2 3]
 [4 5 6]]
```

NumPy Joining arrays - Horizontally

Joining arrays vertically - **hstack**:

- Appends the arrays **side by side**.
- The arrays must have the same number of rows.

```
# Stacking and concatenating arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
horizontal_stack = np.hstack((arr1, arr2))  # Stack arrays horizontally

print("Array 1:\n", arr1)
print("\nArray 2:\n", arr2)
print("\nHorizontally stacked:\n", horizontal_stack)
```

✓ 0.0s

```
Array 1:
 [1 2 3]

Array 2:
 [4 5 6]

Horizontally stacked:
 [1 2 3 4 5 6]
```

Dot Product – 1D-Arrays

Calculates the **scalar product** of corresponding elements.

$$\begin{bmatrix} a & b \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax + by \end{bmatrix}$$

Number of columns in the first matrix is equal to the number of rows in the second matrix.

In this case it has a (1,2) . (2,1) which results in a (1,1)

NumPy Dot Product -1D-Arrays

```
# dot product of two 1d arrays
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])

dot_product = np.dot(arr1, arr2)

print("Array 1:", arr1)
print("Array 2:", arr2)
print("\nDot product:", dot_product)
```

0.0s

```
Array 1: [1 2 3]
Array 2: [4 5 6]

Dot product: 32
```

Dot Product – 2D-arrays

Multiply **each line** of the first matrix element-wise with **each column** of the second matrix and sum these values at the end.

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \bullet \begin{bmatrix} w & x \\ y & z \end{bmatrix} = \begin{bmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{bmatrix}$$

Again, the number of columns in the first matrix is equal to the number of rows in the second matrix.

In this case it has a (2,2) . (2,2) which results in a (2,2)

Animation link.

NumPy Dot Product – 2D-Arrays

Matrix multiplication is **not commutative** – i.e., order matters

```
# numpy dot product of two arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

dot_product = np.dot(arr1, arr2)

print("Array 1:\n", arr1)
print("\nArray 2:\n", arr2)
print("\nDot product of the two arrays:\n", dot_product)
```

0.0s

```
Array 1:
 [[1 2]
 [3 4]]

Array 2:
 [[5 6]
 [7 8]]

Dot product of the two arrays:
 [[19 22]
 [43 50]]
```

```
# numpy dot product of two arrays
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])

dot_product = np.dot(arr2, arr1)

print("Array 1:\n", arr1)
print("\nArray 2:\n", arr2)
print("\nDot product of the two arrays:\n", dot_product)
```

✓ 0.0s

```
Array 1:
 [[1 2]
 [3 4]]

Array 2:
 [[5 6]
 [7 8]]

Dot product of the two arrays:
 [[23 34]
 [31 46]]
```

NumPy Handling Missing data

In computing, **NaN** (Not a Number) is a value used to represent a numeric value that is **undefined or unrepresentable**.

```
# handling nans
arr = np.array([1, np.nan, 3, 4])
print("Array with NaN:\n", arr)
```

✓ 0.0s

```
Array with NaN:
 [ 1. nan  3.  4.]
```

NumPy Handling Missing data - Replacing

We can **replace** nan by another number:

```
# handling nans
arr = np.array([1, np.nan, 3, 4])
nan_index = np.isnan(arr)  # Find indices of NaN values

print("Array with NaN:\n", arr)
print("\nIndices of NaN values:", nan_index)

# handling nans
arr = np.array([1, np.nan, 3, 4])
arr[nan_index] = 0  # Replace NaN values with 0

print("Array with NaN replaced by 0:\n", arr)
```

✓ 0.0s

```
Array with NaN:
 [ 1. nan  3.  4.]

Indices of NaN values: [False  True False False]
Array with NaN replaced by 0:
 [1. 0. 3. 4.]
```

```
# handling nans
arr = np.array([1, np.nan, 3, 4])

arr_filled = np.nan_to_num(arr)  # Replace NaN values with 0

print("Array with NaN:\n", arr)

print("\nArray with NaN replaced by 0:\n", arr_filled)
```

✓ 0.0s

```
Array with NaN:
 [ 1. nan  3.  4.]

Array with NaN replaced by 0:
 [1. 0. 3. 4.]
```

NumPy Handling Missing data - Removing

We also might want to **remove** nan values:

```
    # handling nans
    arr = np.array([1, np.nan, 3, 4])

    arr_without_nan = arr[~np.isnan(arr)]  # Remove NaN values from array

    print("Array with NaN:\n", arr)

    print("\nArray without NaN values:", arr_without_nan)
```

✓ 0.0s

```
Array with NaN:
 [ 1. nan  3.  4.]

Array without NaN values: [1. 3. 4.]
```

Conclusion

In this part we learned various NumPy Operations.

NumPy's powerful array handling and vectorized operations make it an essential library for numerical computing in Python.

Hands-On Session!

Course Shared Folder

CMU Portugal Academy

Thank you!

Carnegie Mellon Portugal | TÉCNICO LISBOA | Ciências ULisboa | NOVA SCHOOL OF SCIENCE & TECHNOLOGY