Welcome to the 2nd Python and ML Bootcamp!

Evangelia Drakopoulou

PhD students: Dimitris Stavropoulos
Vasilis Tsourapis
George Zarpapis

Konstantinos Paschos
Leda Liogka

Who we are

The Astroparticle Group of INPP:

Researchers:

C. Markou, E. Tzamariudaki, E. Drakopoulou

Under Work Contract:

C. Bagatelas

Technical and Support personnel:

V. Tsagli, A. Vougioukas, S. Bakou

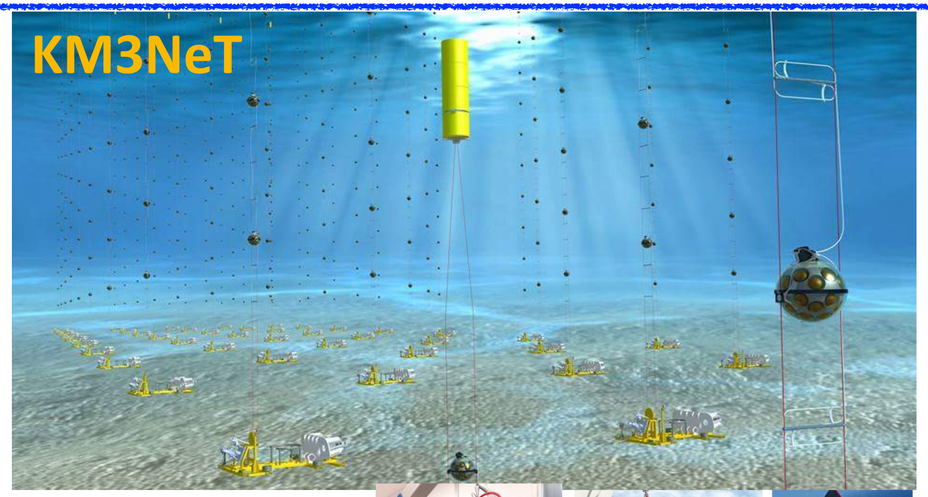
Ph.D. Students:

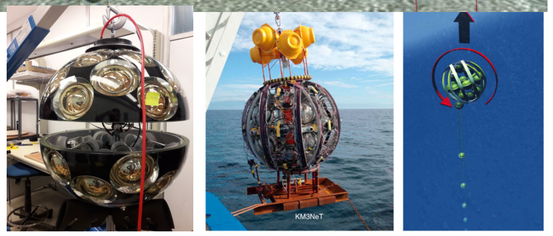
D. Stavropoulos, V. Tsourapis, G. Zarpapis

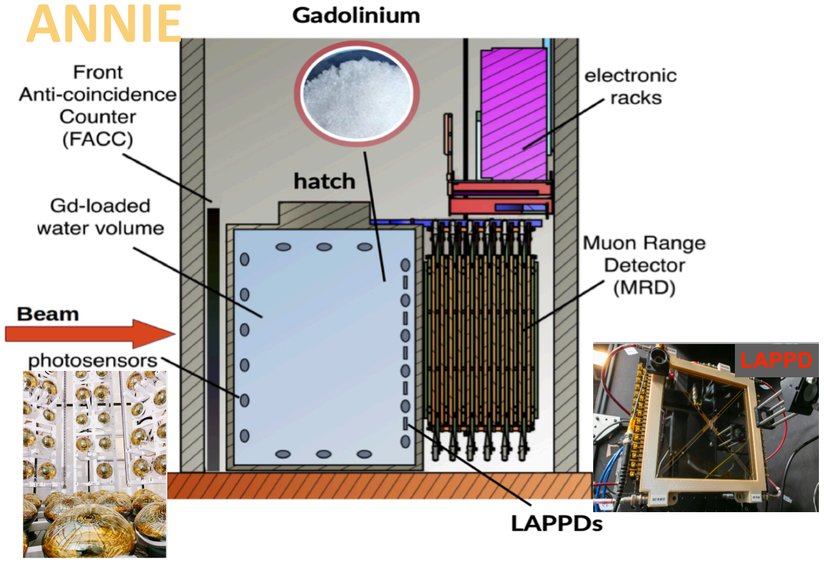
and under/postgraduate students.

Research Interests:

- Astroparticle physics with neutrinos - KM3NeT(km3net)
- Neutrino physics - ANNIE (Fermilab)







Lots of Data - Excellent playfield for Machine Learning techniques

- Neutrino experiments have large datasets:

i. Need to keep only important data

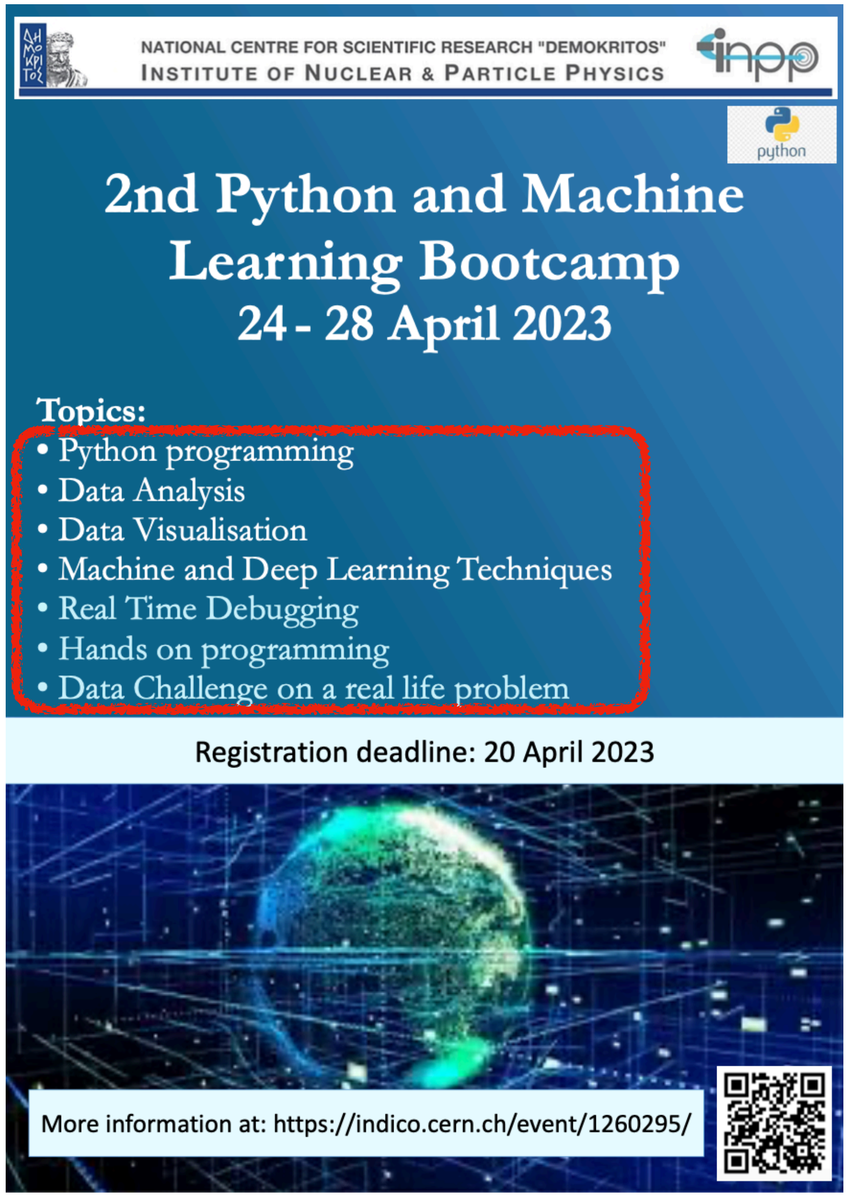
ii. Get the most out of these data - explore most possible combinations

iii. Fast pacing experiments: Be efficient - not just effective

iv. Use state of the art techniques

Extensive use of Machine Learning algorithms: from simple BDTs to GNNs.

Goal of this bootcamp



- The ultimate goal is to provide the basic knowledge of python → use and understand Machine Learning (ML) Algorithms.

How to achieve that:

- presenting the basics of Python and ML
- Use real life examples (industry, physics)
- Hands on exercises

An early take home message:

- excellence comes with practising/programming

Why Python?

According to several popular programming language indices, TIOBE [1], Stack Overflow [2], PYPL [3], and RedMonk, [4] Python is far and away the more popular language across the broader tech community.

Python is becoming more and more popular in research.

Worldwide, Apr 2023 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	27.43 %	-0.8 %
2		Java	16.41 %	-1.7 %
3		JavaScript	9.57 %	+0.3 %
4		C#	6.9 %	-0.3 %
5		C/C++	6.65 %	-0.5 %
6		PHP	5.17 %	-0.5 %
7		R	4.22 %	-0.4 %
8		TypeScript	2.89 %	+0.5 %
9	↑	Swift	2.31 %	+0.2 %
10	↓	Objective-C	2.09 %	-0.1 %

What is Python?

Python is a high-level, general-purpose programming language known for its intuitive syntax that mimics natural language. You can use Python code for a wide variety of tasks, but three popular applications include:

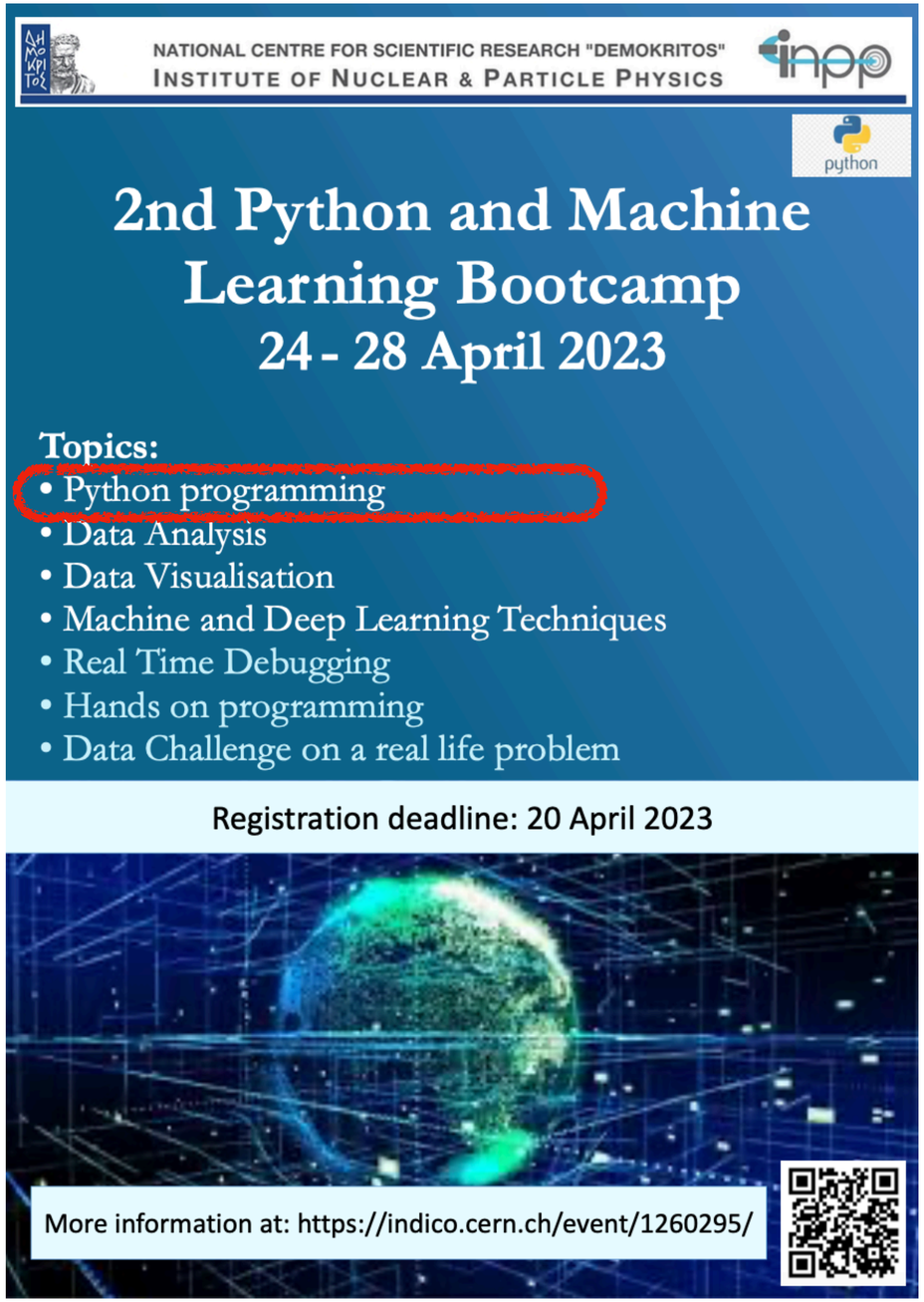
- Data science and data analysis
- Web application development
- Automation/scripting

Let's check the software

i) type: python3

```
edrakopo@Sopas-MacBook-Pro ~ % python3
Python 3.10.8 (main, Oct 13 2022, 10:19:1
Type "help", "copyright", "credits" or "l
>>> import numpy
>>> import pandas
>>> import matplotlib
>>> import seaborn
>>> import keras
2023-04-11 14:18:20.645297: I tensorflow/
se the following CPU instructions in perf
To enable them in other operations, rebui
>>> import sklearn
>>> exit()
```

Ready to go..



Python Programming

Ways of Learning Python

- Examples : read other people's code, try to understand, and then modify from it. ← **faster**

- Step by step: Learn the language rules and build up a program from scratch.

Tips & Tricks:

* Do some exercises while you read a book/website
* Try out to replicate the codes and
* make up your own.

Let's start with the basics and then proceed with examples..

Variables

```
# Create a variable called test_var and give it a value of 4+5
test_var = 4 + 5

# Print the value of test_var
print(test_var)

# Set the value of a new variable to 3
my_var = 3

# Print the value assigned to my_var
print(my_var)

# Change the value of the variable to 100
my_var = 100

# Print the new value assigned to my_var
print(my_var)
print(test_var)
```

Over time, you'll learn how to select good names for Python variables.

Both my_var and test_var exist.

Manipulating variables

```
# Increase the value by 3
my_var = my_var + 3

# Print the value assigned to my_var
print(my_var)
```

```
# Create variables
num_years = 4
days_per_year = 365
hours_per_day = 24
mins_per_hour = 60
secs_per_min = 60
```

- When we use variables (such as num_years, days_per_year, etc), we can better keep track of each part of the calculation and more easily check for and correct any mistakes.
- It is particularly useful to use variables when the values of the inputs can change.

```
# Calculate number of seconds in four years
total_secs = secs_per_min * mins_per_hour * hours_per_day * days_per_year * num_years
print(total_secs)
print('total_secs: ', total_secs)
```

printing a short description.

Debugging

Attention!!! Debugging is as important as coding.

Adding comments to your code and keeping it tidy always helps.
-> It can be a life/time-saver!!

- One common error when working with variables is to accidentally introduce typos.
- Try to read the message carefully before typing…

```
print(hours_per_dy)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipykernel_18/142450907.py in <module>
----> 1 print(hours_per_dy)
NameError: name 'hours_per_dy' is not defined
```

```
print(hours_per_day)
```

Useful Tools

slack

Useful Tools

- slack for instant messaging

https://slack.com Workspace **Python&ML Bootcamp**

- Github: platform for software development and version control

https://github.com/edrakopo/pythonMLBootcamp

- In terminal type:

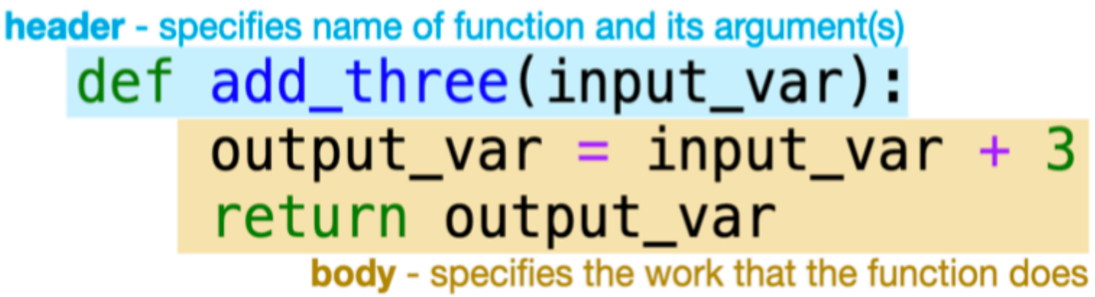
```
git clone https://github.com/edrakopo/pythonMLBootcamp.git
```

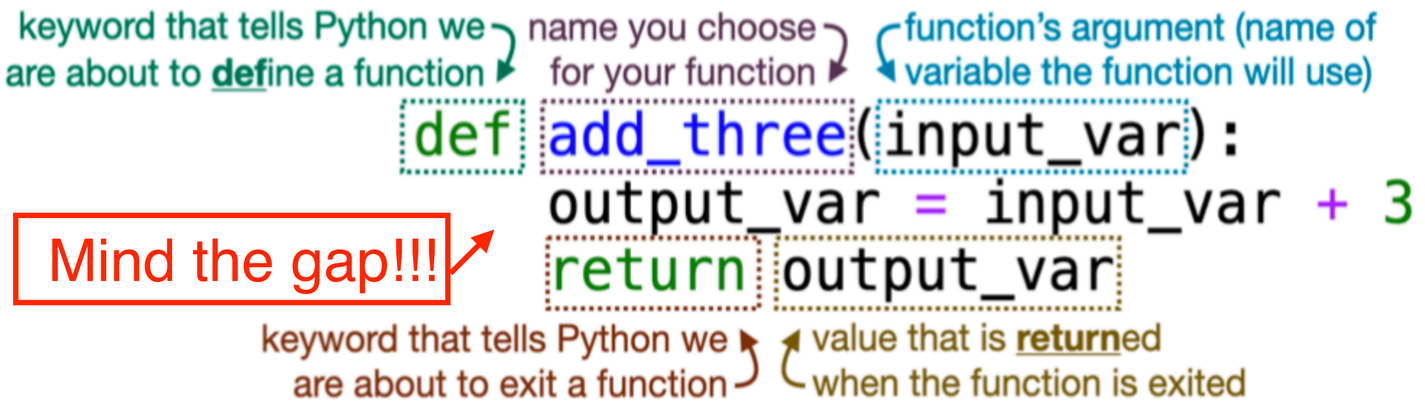
```
cd pythonMLBootcamp
```

Functions with one argument

- The add_three() function below accepts any number, adds three to it, and then returns the result.

```
# Define the function
def add_three(input_var):
    output_var = input_var + 3
    return output_var
```





Run ("call") a function

```
# Run the function with 10 as input
new_number = add_three(10)

# Check that the value is 13, as expected
print(new_number)
```

Functions with multiple arguments

Function calculating a weekly paycheck based on three arguments:

i. num_hours - number of hours worked in one week

ii. hourly_wage - the hourly wage (in $/hour)

iii. tax_bracket - percentage of your salary that is removed for taxes (in decimal)

```
def get_pay_with_more_inputs(num_hours, hourly_wage, tax_bracket):
    # Pre-tax pay
    pay_pretax = num_hours * hourly_wage
    # After-tax pay
    pay_aftertax = pay_pretax * (1 - tax_bracket)
    return pay_aftertax
```

Task:

How can we calculate the pay after taxes for someone who works 40 hours, makes $24/hour, and is in a 22% tax bracket? Print the result.

Functions with multiple arguments

Function calculating a weekly paycheck based on three arguments:

i. num_hours - number of hours worked in one week

ii. hourly_wage - the hourly wage (in $/hour)

iii. tax_bracket - percentage of your salary that is removed for taxes (in decimal)

```
def get_pay_with_more_inputs(num_hours, hourly_wage, tax_bracket):
    # Pre-tax pay
    pay_pretax = num_hours * hourly_wage
    # After-tax pay
    pay_aftertax = pay_pretax * (1 - tax_bracket)
    return pay_aftertax
```

Answer:

```
higher_pay_aftertax = get_pay_with_more_inputs(40, 24, .22)
print(higher_pay_aftertax)
```

Functions with no arguments

- Functions with no arguments printing a standard message:

```
# Define the function with no arguments and with no return
def print_hello():
    print("Hello, you!")
    print("Good morning!")

# Call the function
print_hello()
```

Examples (I)

```
# multiple arguments are passed

# function definition
def displayMessage(argument1, argument2, argument3):
        print(argument1+" "+argument2+" "+argument3)


# function call
displayMessage("testing", "my", "code")
```

```
testing my code
```

Task:

Create a function with two arguments: i) string ii) integer, that prints this message:

Hello your_name , visitor 99!

Hands On

Task:

Create a function with two arguments: i) string ii) integer, that prints this message:

Hello your_name , visitor 99!

Answer:

```
# function definition
def displayMessage(argument1, argument2):
    print("Hello ",argument1," , visitor ",argument2,"!")
```

```
# function call
displayMessage("you",99)
```

```
Hello  you  , visitor  99 !
```

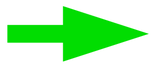
Examples(II)

- We can pass multiple arguments to a python function without predetermining the formal parameters using: def functionName(*argument)

```
# variable number of non keyword arguments passed

# function definition
def calculateTotalSum(*arguments):
    totalSum = 0
    for number in arguments:
        totalSum += number
    print(totalSum)


# function call
calculateTotalSum(5, 4, 3, 2, 1)
```



15

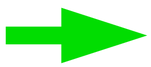
Conditions

Conditions are statements that are either `True` or `False`.

Symbol	Meaning
==	equals
!=	does not equal
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

```
var_one = 1
var_two = 2

print(var_one < 1)
print(var_two >= var_one)
```



```
False
True
```

Conditional Statements

"if" statements

```
def evaluate_temp(temp):
    # Set an initial message
    message = "Normal temperature."
    # Update value of message only if temperature greater than 38
    if temp > 38:
        message = "Fever!"
    return message
```

Check the space!!! the return statement is not indented under the "if" statement, it is always executed

The function accepts a body temperature (in Celcius) as input.

- Initially, message is set to "Normal temperature".
- Then, if temp > 38 is True (e.g., the body temperature is greater than 38°C), the message is updated to "Fever!". Otherwise, if temp > 38 is False, then the message is not updated.
- Finally, message is returned by the function.

```
print(evaluate_temp(37))
```
→ Normal temperature.

```
print(evaluate_temp(39))
```
→ Fever!

Conditional Statements

“if…else” statements

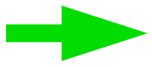
We can use "else" statements to run code if a statement is False. The code under the "if" statement is run if the statement is True, and the code under "else" is run if the statement is False.

```
def evaluate_temp_with_else(temp):
    if temp > 38:
        message = "Fever!"
    else:
        message = "Normal temperature."
    return message
```

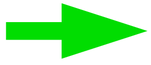
Task:

Predict the result!

```
print(evaluate_temp_with_else(37))
```



```
print(evaluate_temp(39))
```



Conditional Statements

“if…else” statements

We can use "else" statements to run code if a statement is False. The code under the "if" statement is run if the statement is True, and the code under "else" is run if the statement is False.

```
def evaluate_temp_with_else(temp):
    if temp > 38:
        message = "Fever!"
    else:
        message = "Normal temperature."
    return message
```

Task:

Predict the result!

```
print(evaluate_temp_with_else(37))
```
→ Normal temperature.

```
print(evaluate_temp(39))
```
→ Fever!

Conditional Statements

“if...elif…else” statements

We can use "elif" (which is short for "else if") to check if multiple conditions might be true.

```
def evaluate_temp_with_elif(temp):
    if temp > 38:
        message = "Fever!"
    elif temp > 35:
        message = "Normal temperature."
    else:
        message = "Low temperature."
    return message
```

```
print(evaluate_temp_with_elif(36))
```
→ Normal temperature.

```
print(evaluate_temp_with_elif(34))
```
→ Low temperature.

Hands On

Task:

Create a function with one argument: a number (i.e. num). It will check if the number is greater/smaller than zero and it will print 3 lines:

```
Check your number:
num is a positive number.
Job completed.
```

```
Check your number:
num is a negative number.
Job completed.
```

Hands On

Task:

Create a function with one argument: a number (i.e. num). It will check if the number is greater/smaller than zero and it will print 3 lines:

Answer:

```
def check_num(num):
    print("Check your number:")
    if num > 0:
        print(num, "is a positive number.")
    if num < 0:
        print(num, "is a negative number.")
    print("Job completed.")
```

```
# function call
check_num(3)
```

```
Check your number:
3 is a positive number.
Job completed.
```

```
check_num(-2)
```

```
Check your number:
-2 is a negative number.
Job completed.
```

Debugging (I)

Attention!!! Debugging is as important as coding.

Adding comments to your code and keeping it tidy always helps.

-> It can be a life/time-saver!!

- Let's debug some examples - Common mistakes…

```
def evaluate_temp(temp):
    # Set an initial message
    message = "Normal temperature."
    # Update value of message only if temperature greater than 38
    if temp > 38:
        message = "Fever!"
    return message
```

```
 print(evaluate_temp_with_else(37))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'evaluate_temp_with_else' is not defined
```

Why 

Debugging (I)

Attention!!! Debugging is as important as coding.

Adding comments to your code and keeping it tidy always helps.

-> It can be a life/time-saver!!

- Let's debug some examples - Common mistakes…

```
def evaluate_temp(temp):
    # Set an initial message
    message = "Normal temperature."
    # Update value of message only if temperature greater than 38
    if temp > 38:
        message = "Fever!"
    return message
```

```
 print(evaluate_temp_with_else(37))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'evaluate_temp_with_else' is not defined
```

Answer

use of wrong name

Debugging (II)

Attention!!! Debugging is as important as coding.

Adding comments to your code and keeping it tidy always helps.

-> It can be a life/time-saver!!

- Let's debug some examples - Common mistakes…

```
def evaluate_temp0(temp):
    # Set an initial message
    message = "Normal temperature."
    # Update value of message only if temperature greater than 38
     if temp > 38:
        message = "Fever!"
     return message


print(evaluate_temp0(37))
    if temp > 38:
IndentationError: unexpected indent
```

Why 

Debugging (II)

Attention!!! Debugging is as important as coding.

Adding comments to your code and keeping it tidy always helps.

-> It can be a life/time-saver!!

- Let's debug some examples - Common mistakes…

```
def evaluate_temp0(temp):
    # Set an initial message
    message = "Normal temperature."
    # Update value of message only if temperature greater than 38
     if temp > 38:
        message = "Fever!"
     return message
```

Answer

wrong space

```
print(evaluate_temp0(37))
    if temp > 38:
IndentationError: unexpected indent
```

Loops

Loops are a way to repeatedly execute some code. Here's an example:

for loop

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

list of strings

```
for planet in planets:
    print(planet, end=' ') # print all on same line
```

Mercury Venus Earth Mars Jupiter Saturn Uranus Neptune

The **for** loop specifies

- the variable name to use (in this case, planet)
- the set of values to loop over (in this case, planets)

You use the word "in" to link them together.

Loops

range()

range() is a function that returns a sequence of numbers. It is useful for writing loops.

```
for i in range(5):
    print("Doing important work. i =", i)
```

```
Doing important work. i = 0
Doing important work. i = 1
Doing important work. i = 2
Doing important work. i = 3
Doing important work. i = 4
```

till 4 (i.e. not 5 inclusive)

while loop

The other type of loop in Python is a while loop, which iterates until some condition is met:

```
i = 0
while i < 10:
    print(i, end=' ')
    i += 1 # increase the value of i by 1
```

0 1 2 3 4 5 6 7 8 9

Hands On

Task:

Use **for** and **range** to return a sequence of numbers until 10. Then, using **if** statements print "… is greater than 3" if the number is greater than 3 and "… is less than 3" if it is less than 3.

Hands On

Task:

Use **for** and **range** to return a sequence of numbers until 9 (9 inclusive). Then, using **if** statements print "… is greater than 3" if the number is greater than 3 and "… is less than 3" if it is less than 3.

Answer:

```
for i in range(10):
    if i>3:
        print(i," is greater than 3")
    if i<3:
        print(i," is less than 3")
```

```
0  is less than 3
1  is less than 3
2  is less than 3
4  is greater than 3
5  is greater than 3
6  is greater than 3
7  is greater than 3
8  is greater than 3
9  is greater than 3
```

← 3 is not printed

Why ❓

Lists

Lists in Python represent ordered sequences of values. Here is an example of how to create them:

```
primes = [2, 3, 5, 7]
```

list of integers

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

list of strings

```
hands = [
    ['J', 'Q', 'K'],
    ['2', '2', '2'],
    ['6', 'A', 'K'], # (Comma after the last element is optional)
]
# (I could also have written this on one line, but it can get hard to read)
hands = [['J', 'Q', 'K'], ['2', '2', '2'], ['6', 'A', 'K']]
```

A list can contain a mix of different types of variables:

```
my_favourite_things = [32, 'raindrops on roses', help]
```

Lists - Indexing

List of planets:

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

```
planets[0]
```

'Mercury'

Elements at the end of the list can be accessed with negative numbers, starting from -1:

```
planets[-1]
```

'Neptune'

What's the next closest planet to the Sun?

Which planet is the 2nd furthest from the sun?

Lists - Indexing

List of planets:

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

`planets[0]` → 'Mercury'

Elements at the end of the list can be accessed with negative numbers, starting from -1:

`planets[-1]` → 'Neptune'

Answer:

What's the next closest planet to the Sun? `planets[1]` → 'Venus'

Which planet is the 2nd furthest from the sun? `planets[-2]` → 'Uranus'

Lists - Slicing

You can also pull a segment of a list (for instance, the first three entries or the last two entries). This is called **slicing**. For instance:

- to pull the first x entries, you use **[:x]**
- to pull the last y entries, you use **[-y:]**

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

What are the first three planets?

```
planets[0:3]
```
→ ['Mercury', 'Venus', 'Earth']

```
planets[:3]
```
→ ['Mercury', 'Venus', 'Earth']

```
planets[3:]
```
→ ['Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

i.e. the expression above means "give me all the planets from index 3 onward"

```
# All the planets except the first and last
planets[1:-1]
```
→ ['Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus']

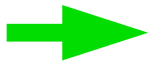
```
# The last 3 planets
planets[-3:]
```
→ ['Saturn', 'Uranus', 'Neptune']

Lists - Removing/Adding items

Remove an item from a list with .remove(), and put the item you would like to remove in parentheses.

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']
```

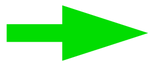
```
planets.remove('Neptune')
print(planets)
```



```
['Mercury', 'Venus', 'Earth', 'Mars',
'Jupiter', 'Saturn', 'Uranus']
```

Add an item to a list with .append(), and put the item you would like to add in parentheses.

```
planets.append('Neptune2')
>>> print(planets)
```



```
['Mercury', 'Venus', 'Earth', 'Mars',
'Jupiter', 'Saturn', 'Uranus', 'Neptune2']
```

Replace an item:

```
planets[2] = “Test"
print(planets)
```

```
['Mercury', 'Venus', 'Test', 'Mars',
'Jupiter', 'Saturn', 'Uranus', 'Neptune2']
```

Lists - Useful commands

```
values = [139, 128, 172, 139, 191, 168, 170]
```

list of integers

```
print("Length of the list:", len(values))
```

```
Length of the list: 7
```

```
print("Entry at index 2:", values[2])
```

```
Entry at index 2: 172
```

- You can also get the minimum with min() and the maximum with max().

```
print("Minimum:", min(values))
```

```
Minimum: 128
```

```
print("Maximum:", max(values))
```

```
Maximum: 191
```

- To add every item in the list, use sum()

```
print("Sum of values:", sum(values))
```

```
Sum of values: 1107
```

Note: If you apply sum on a list of booleans (True, False) True counts for 1 and False for 0.

- We can also take the sum from the first five elements (sum(values[:5])), and then divide by five to get the average.

```
print("Average:", sum(values[:5])/5)
```

```
Average: 153.8
```

Lists - Useful commands

```
values = [139, 128, 172, 139, 191, 168, 170, 128, 128]
```

list of integers

```
print("Length of the list:", len(values))
```

→ `Length of the list: 9`

```
print("count: ", values.count(values[1]))
```

→ `count:  3`

- counts the occurrences of the 2nd element in this list

```
listA = [19,19,15,5,3,5,5,2]
```

```
>>> listA = [19,19,15,5,3,5,5,2]
>>> print("count: ", values.count(values[1]))
count:  3
>>> print("count: ", values.count(values[2]))
count:  1
>>> print("check: ", values.count(values[2])==2)
check:  False
>>> print("check: ", values.count(values[2])==1)
check:  True
>>> print("check: ", values.count(values[1])==3)
check:  True
```

counts the occurrences of the 2nd element in this list

checks if the occurrences of the 3rd element in this list are 2

Hands On

Task 1: a restaurant with five food dishes, organised in the Python list `menu` below. One day, you decide to:

(1)

- remove bean soup (`'bean soup'`) from the menu, and
- add roasted beet salad (`'roasted beet salad'`) to the menu.

Implement this change to the list below. While completing this task,

- do not change the line that creates the `menu` list.
- your answer should use `.remove()` and `.append()`.

(2)

- remove bean soup (`'bean soup'`) from the menu, and add roasted beet salad (`'roasted beet salad'`) at the same place of the menu.

menu = ['stewed meat with onions', 'bean soup', 'risotto with trout and shrimp', 'fish soup with cream and onion', 'gyro']

Hands On

Answer:

(1)

```
>>> menu = ['stewed meat with onions', 'bean soup', 'risotto with trout and shrimp','fish soup with cream and onion', ‘gyro']
>>> menu.remove("bean soup")
>>> print(menu)
['stewed meat with onions', 'risotto with trout and shrimp', 'fish soup with cream and onion', 'gyro']
>>> menu.append('roasted beet salad')
>>> print(menu)
['stewed meat with onions', 'risotto with trout and shrimp', 'fish soup with cream and onion', 'gyro', 'roasted beet salad']
```

(2)

```
menu = ['stewed meat with onions', 'bean soup', 'risotto with trout and shrimp','fish soup with cream and onion', 'gyro']
>>> menu[1] = "roasted beet salad"
>>> print(menu)
['stewed meat with onions', 'roasted beet salad', 'risotto with trout and shrimp', 'fish soup with cream and onion', 'gyro']
```

Hands On

Task 2:

In Python, you can quickly turn a string into a list with `.split()`. In the parentheses, we need to provide the character should be used to mark the end of one list item and the beginning of another, and enclose it in quotation marks.

For example, in the sting below that character is a comma.

```
flowers = "pink primrose,hard-leaved pocket orchid,canterbury bells,sweet pea"
print(flowers.split(","))
```

```
['pink primrose', 'hard-leaved pocket orchid', 'canterbury bells', 'sweet pea']
```

Can you convert these strings into lists?

```
alphabet = "A.B.C.D.E.F.G.H.I.J.K.L.M.N.O.P.Q.R.S.T.U.V.W.X.Y.Z"
address = "Mr. H. Potter,The cupboard under the Stairs,4 Privet Drive,Little Whinging,Surrey"
```

Hands On

Task 2:

```
flowers = "pink primrose,hard-leaved pocket orchid,canterbury bells,sweet pea"
print(flowers.split(","))
```

```
['pink primrose', 'hard-leaved pocket orchid', 'canterbury bells', 'sweet pea']
```

Can you convert these strings into lists?

alphabet = "A.B.C.D.E.F.G.H.I.J.K.L.M.N.O.P.Q.R.S.T.U.V.W.X.Y.Z"

address = "Mr. H. Potter,The cupboard under the Stairs,4 Privet Drive,Little Whinging,Surrey"

Answer:

```
print(alphabet.split("."))
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z']
>>> print(address.split(","))
['Mr. H. Potter', 'The cupboard under the Stairs', '4 Privet Drive', 'Little Whinging', 'Surrey']
```

List Comprehensions

List comprehensions are one of Python's most beloved and unique features.

```
squares = [n**2 for n in range(10)]
squares
```

→ [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

squares of int: [0,10)

Here's how we would do the same thing *without* a list comprehension:

```
squares = []
for n in range(10):
    squares.append(n**2)
squares
```

→ [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

We can also add an **if** condition:

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter', 'Saturn', 'Uranus', 'Neptune']

short_planets = [planet for planet in planets if len(planet) < 6]
short_planets
```

↓

['Venus', 'Earth', 'Mars']

List Comprehensions

List comprehensions combined with functions like min, max, and sum can lead to impressive one-line solutions for problems that would otherwise require several lines of code.

```
def count_negatives(nums):
    """Return the number of negative numbers in the given list.
   >>> count_negatives([5, -1, -2, 0, 3])
    2
    """
    n_negative = 0
    for num in nums:
        if num < 0:
            n_negative = n_negative + 1
    return n_negative
```

Here's a solution using a list comprehension:

```
def count_negatives(nums):
    return len([num for num in nums if num < 0])
```

Further minimizing the length of our code:

```
def count_negatives(nums):
    # Reminder: True counts for 1 and False for 0.
    return sum([num < 0 for num in nums])
```

Hands On

Task 3: **List comprehensions:** Create a list based on the values of another list.

```
test_ratings = [1, 2, 3, 4, 5]
```

Then we can use this list (`test_ratings`) to create a new list (`test_liked`) where each item has been turned into a boolean, depending on whether or not the item is greater than or equal to four.

```
test_liked = [i>=4 for i in test_ratings]
print(test_liked)
```

`ratings`: list of ratings that people gave to a movie, where each rating is a number between 1-5, inclusive

We say someone liked the movie, if they gave a rating of either 4 or 5. Create a function (`percentage_liked`) which gets these `ratings` and returns the percentage of people who liked the movie.

For instance, if we supply a value of

`percentage_liked([1, 2, 3, 4, 5, 4, 5, 1])`

then 50% (4/8) of the people liked the movie, and the function should return **0.5**.

Hands On

Task 3:

Your function (`percentage_liked`) should get these ratings and return the percentage of people who liked the movie.

Hint 1:

```
def percentage_liked(ratings):
    list_liked = [i>=4 for i in ratings]
    print("list_liked ",list_liked)
    print("len(list_liked) ",len(list_liked))
    print("len(ratings) ",len(ratings))
    percentage_liked = ……….
    return percentage_liked
```

Define the function

create the new list (`list_liked`)

Print some variables to understand

Calculate the percentage

Return the result

Hands On

Task 3:

Your function (`percentage_liked`) should get these ratings and return the percentage of people who liked the movie.

Hint2:

```
def percentage_liked(ratings):
    list_liked = [i>=4 for i in ratings]
    print("list_liked ",list_liked)
    print("len(list_liked) ",len(list_liked))
    print("len(ratings) ",len(ratings))
    percentage_liked = ……….
    return percentage_liked
```

Define the function

create the new list (`list_liked`)

Print some variables to understand

Calculate the percentage

Return the result

Note: If you apply sum on a list of booleans (True, False) True counts for 1 and False for 0.

Hands On

Task 3:

Your function (`percentage_liked`) should get these ratings and return the percentage of people who liked the movie.

Answer:

```
def percentage_liked(ratings):
    list_liked = [i>=4 for i in ratings]
    print("list_liked ",list_liked)
    print("len(list_liked) ",len(list_liked))
    print("len(ratings) ",len(ratings))
    percentage_liked = sum(list_liked)/len(ratings)
    return percentage_liked
```

```
percentage_liked([1, 2, 3, 4, 5, 4, 5, 1])
```

```
0.5
```

List Comprehensions - lambda function

A lambda function is a small anonymous function. A lambda function can take any number of arguments, but can only have one expression.

Example:

Write a Python program to create a lambda function that adds 15 to a given number passed in as an argument, also create a lambda function that multiplies argument x with argument y and prints the result.

```
r = lambda a : a + 15
print(r(10))
r = lambda x, y : x * y
print(r(12, 4))
```

```
>>> r = lambda a : a + 15
>>> print(r(10))
25
>>> r = lambda x, y : x * y
>>> print(r(12, 4))
48
```

Task:

Write a Python program to create a function that takes one argument, and that argument will be multiplied with an unknown given number (n).

List Comprehensions - lambda function

Task:

Write a Python program to create a function that takes one argument, and that argument will be multiplied with an unknown given number (n).

Answer:

```
def func_compute(n):
 return lambda x : x * n
result = func_compute(2)
print("Double the number of 15 =", result(15))
result = func_compute(3)
print("Triple the number of 15 =", result(15))
result = func_compute(4)
print("Quadruple the number of 15 =", result(15))
result = func_compute(5)
print("Quintuple the number 15 =", result(15))
```

```
>>> result = func_compute(2)
>>> print("Double the number of 15 =", result(15))
Double the number of 15 = 30
>>> result = func_compute(3)
>>> print("Triple the number of 15 =", result(15))
Triple the number of 15 = 45
>>> result = func_compute(4)
>>> print("Quadruple the number of 15 =", result(15))
Quadruple the number of 15 = 60
>>> result = func_compute(5)
>>> print("Quintuple the number 15 =", result(15))
Quintuple the number 15 = 75
```

Dictionaries

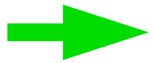
Dictionaries are a built-in Python data structure for mapping keys to values.

```
numbers = {'one':1, 'two':2, 'three':3}
```

In this case 'one', 'two', and 'three' are the **keys**, and 1, 2 and 3 are their corresponding values.

Values are accessed via square bracket syntax similar to indexing into lists and strings.

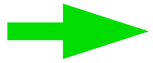
```
numbers['one']
```



1

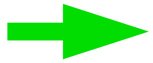
We can use the same syntax to add another key, value pair

```
numbers['eleven'] = 11
numbers
```



{'one': 1, 'two': 2, 'three': 3, 'eleven': 11}

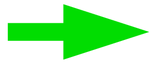
```
numbers['one'] = 'Pluto'
numbers
```



{'one': 'Pluto', 'two': 2, 'three': 3, 'eleven': 11}

Python has dictionary comprehensions with a syntax similar to the list comprehensions

```
planets = ['Mercury', 'Venus', 'Earth', 'Mars', 'Jupiter',
'Saturn', 'Uranus', 'Neptune']
planet_to_initial = {planet: planet[0] for planet in planets}
planet_to_initial
```



```
{'Mercury': 'M',
 'Venus': 'V',
 'Earth': 'E',
 'Mars': 'M',
 'Jupiter': 'J',
 'Saturn': 'S',
 'Uranus': 'U',
 'Neptune': 'N'}
```

Working with External Libraries

One of the best things about Python is the vast number of high-quality custom libraries that have been written for it.

Some of these libraries are in the "standard library", meaning you can find them anywhere you run Python. Other libraries can be easily added, even if they aren't always shipped with Python.

Use **import** to access the libraries.

```
import math
print("It's math! It has type {}".format(type(math)))
```

```
It's math! It has type <class 'module'>
```

math is a module. A module is just a collection of variables defined by someone else. We can see all the names in math using the built-in function dir().

```
print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', 'acos', 'acosh', 'asin',
'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf', 'isclose', 'isfinite', 'isinf',
'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'remainder',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

Working with External Libraries

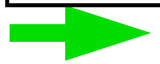
```
import math as mt
mt.pi
```

3.141592653589793

The as simply renames the imported module.

```
from math import *
print(pi, log(32, 2))
```

import * makes all the module's variables directly accessible to you



3.141592653589793 5.0

But:

```
from math import *
from numpy import *
print(pi, log(32, 2))
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_19/3018510453.py in <module>
      1 from math import *
      2 from numpy import *
----> 3 print(pi, log(32, 2))

TypeError: return arrays must be of ArrayType
```

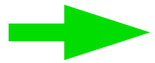
What Why

What has happened? It worked before!

Working with External Libraries

But:

```
from math import *
from numpy import *
print(pi, log(32, 2))
```



```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_19/3018510453.py in <module>
      1 from math import *
      2 from numpy import *
----> 3 print(pi, log(32, 2))

TypeError: return arrays must be of ArrayType
```

What Why

These kinds of "star imports" can occasionally lead to weird, difficult-to-debug situations.

The problem in this case is that the math and numpy modules both have functions called log, but they have different semantics. Because we import from numpy second, its log overwrites (or "shadows") the log variable we imported from math.

Use that instead:

```
from math import log, pi
from numpy import asarray
```

Create you own lib/class

Suppose you need to perform a calculation/plot etc. several times or you want your code to be used by others.

Step 1 Start by writing a script

```
# Set the input
number = 5


# Multiply by two
result = number * 2


# Display the result
print(result)
```

Step 2 Turn your script into a documented function

```
1  def multiply(number, muiltiplier):
2      """
3      Multiply a given number by a given multiplier.
4
5      :param number: The number to multiply.
6      :type number: int
7
8      :param muiltiplier: The multiplier.
9      :type muiltiplier: int
10     """
11
12     return number * muiltiplier
13
14 # Call the function
15 print(multiply(5, 2))
```

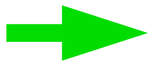
Use you lib

You can use your library already at this step.

1. Create a python script: mylib.py and store your function.
2. Create another script: test_mylib.py to import the first one as library.

test_mylib.py

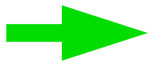
```
1 import mylib as lib
2
3 print(lib.multiply(4, 2))
```



```
10  This print comes from your lib: mylib.py
8   This print comes from your script: test_mylib.py
```

Common Bug:

```
1 import mylib
2
3 print(multiply(4, 2))
```



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

Using Classes

In object oriented programming classes and objects are the main features. A class creates a new data type and objects are instances of a class which follows the definition given inside the class.

Example:

Let's create a simple class using class keyword followed by the class name (Student) which follows an indented block of segments (student class, roll no., name).

```
class Student:
    """A simple example class"""
    def __init__(self, sclass, sroll, sname):
        self.c = sclass
        self.r = sroll
        self.n = sname
    def messg(self):
            return 'New Session will start soon.'
```

__init__ method: this methods does some initialization work and serves as a constructor for the class.

In Python **self** is a name for the first argument of a method which is different from ordinary function. Rather than passing the object as a parameter in a method the word self refers to the object itself.

Using Classes

Example:

Let's create a simple class using class keyword followed by the class name (Student) which follows an indented block of segments (student class, roll no., name).

```
class Student:
    """A simple example class"""
    def __init__(self, sclass, sroll, sname):
        self.c = sclass
        self.r = sroll
        self.n = sname
    def messg(self):
            return 'New Session will start soon.'
```

Task:

Save the class to a python script: classStudent.py

Debug this script *use_classStudent_withBug.py* to create the first class object and print its attributes.

Create you own class

The Object-Oriented Programming makes the maintainability of your code easier for you, and for the developers who will contribute to your library.

Step 3

Transform to an Object

```python
class Multiplication:
    """
    Instantiate a multiplication operation.
    Numbers will be multiplied by the given multiplier.

    :param multiplier: The multiplier.
    :type multiplier: int
    """

    def __init__(self, multiplier):
        self.multiplier = multiplier

    def multiply(self, number):
        """
        Multiply a given number by the multiplier.

        :param number: The number to multiply.
        :type number: int

        :return: The result of the multiplication.
        :rtype: int
        """

        return number * self.multiplier

# Instantiate a Multiplication object
multiplication = Multiplication(2)

# Call the multiply method
print(multiplication.multiply(5))
```

Store it in a python script:

myclass.py

and run it by: *python3 myclass.py*

```
10
```

Use your class

You can now use your class.

Create another script: test_myclass.py and import your class.

test_myclass.py

```
1 # Import Multiplication from your library
2 from myclass import Multiplication
3
4 # Instantiate a Multiplication object
5 multiplication = Multiplication(2)
6
7 # Call the multiply method
8 print(multiplication.multiply(4))
```

```
10  This print comes from your lib: myclass.py
8   This print comes from your script: test_myclass.py
```

Note: if your class is stored in a different directory than test_myclass.py, you call it by:

```
1 # Import Multiplication from your library
2 from storeclass.myclass import Multiplication
```