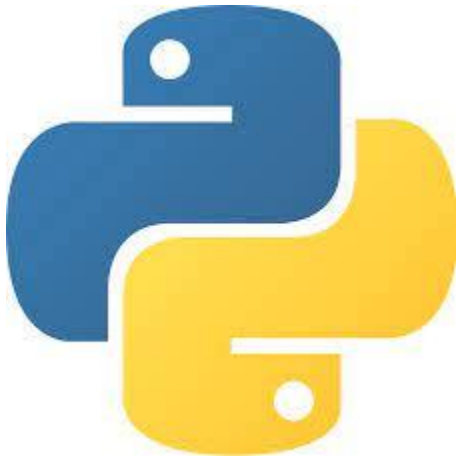
Data Processing and Visualizations using Python

Day 1 – Introduction to Python



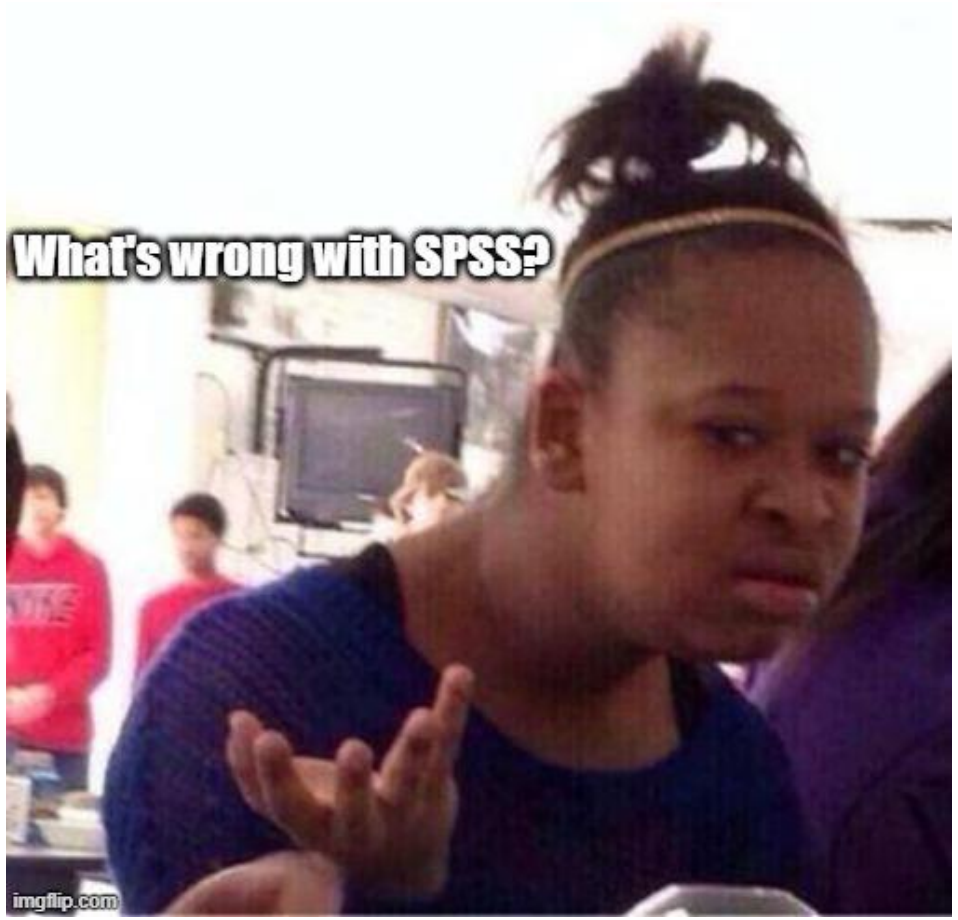
SICSS 2022 – Haifa University

Amit Donner

What will we learn?

1. Introduction and motivation.
2. Basic programming :
   - Variables, comments, reserved keywords.
   - Conditions, nested conditions.
   - Lists, tuples and dictionaries.
   - Methods and no more than 5 sentences on OOP.
   - Loops and functions (arguments and return values).
   - Importing additional modules.
3. Basic visualization using matplotlib:
   - Histogram, scatter plot, bar plot, etc.
   - Colors, legend, axis ticks and additional figure attributes.
   - Visualizing results of important statistical theorems.
4. numpy – section 2, but better.
   - Vectorization, nd-arrays, indexing, element-wise operations, linspace, arange, repeat, where, unique, time comparison, arg methods, random.choice + apply along axis and bootstrap.
5. Pandas (Part 1) – Let’s get this data started!
   - Impot csv and excel files, data frames, index, columns, indexing, filtering, rename, dtypes, select_dtypes, unique, count_values, cut, sort_values, crosstabs, group by, reset_index, cov, corr, mean, std, plot.
6. Seaborn – advanced visualizations:
   - catplot, relplot, implot, displot, jointplot, pairplot, heatmaps and set_theme.
   - Bootstrap revisited.
7. Pandas (Part 2)
   - Transform, assign, fillna, dropna, drop_duplicates, get dummies, pivot, concat, melt, merge.
   - Covid 19 data from Israel

Motivation

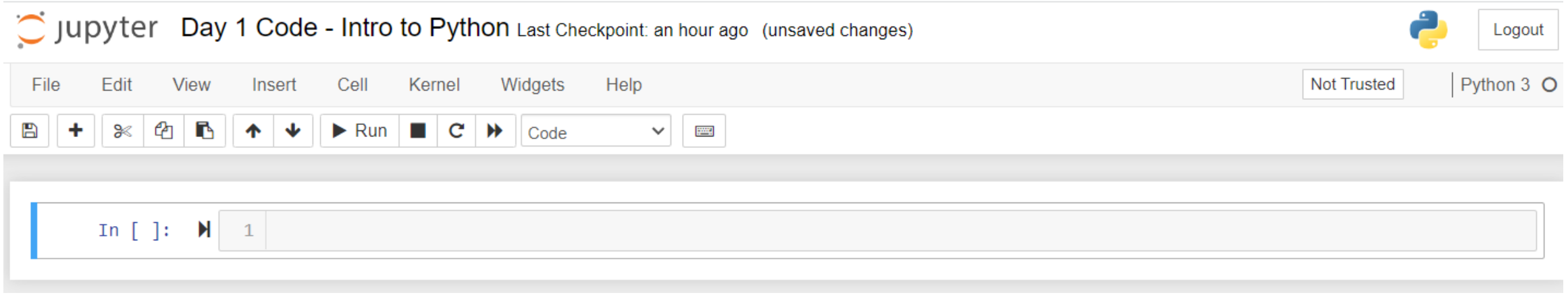


Introduction

- Python is an open-source programming language that became popular in recent years mainly due to its simplicity and high capabilities.
- It was designed by Guido van Rossum around 1990 in order to make coding accessible for everybody.
- Some of Python’s strongest skills are data processing, visualizations and deploying machine learning algorithms.

Jupyter Notebooks

- In our meetings we will use the Jupyter Notebooks.
- Very popular workspace, especially in data science fields.
- Consists of code and text (markdown) chunks.
- Easy to use, share and design.



First steps…

- In every intro to programming course, we do the same thing…

```
1 print("Hello world!")
```

```
Hello world!
```

- Thus, we will be original, and do the following:

```
1 print("Hello SICSS 2022 participants!")
```

```
Hello SICSS 2022 participants!
```

- Congratulations – you have written your first Python code!

- Let’s try something a little bit more complicated. Let’s assume that we wish to print a full sentence but with a line break. Using the character “\n” we can force the print function to add a line break.

```
1 print("Hello SICSS 2022 participants! \nHaving fun yet?")
```

```
Hello SICSS 2022 participants!
Having fun yet?
```

- What about multiple texts? Well, no problem at all!

```
1 print("Hello SICSS 2022 participants!",
2       "\nHaving fun yet?",
3       "Still prefer SPSS?")
```

```
Hello SICSS 2022 participants!
Having fun yet? Still prefer SPSS?
```

- Observe that line breaks aren’t working without “\n”.
- What does “ “ mean? We will see shortly.

Variables

- Python enables us to assign different values to arbitrary ‘variables’. For example,

```
a = 1
print(a)
```

```
1
```

We defined the variable a with the value 1, and now Python recognizes a as 1.

- Additional examples:

```
a = 1
b = 4.689
c = "Example"
print(a)
print(b)
print(c)
```

```
1
4.689
Example
```

Variable types

- Each variable we define has a unique type (out of some collection of types which are familiar to Python). In the example before,

```
print(type(a))
print(type(b))
print(type(c))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
```

- Another important type is ‘Boolean’ (True or False) – which is the result of a ‘logical test’. For example,

```
print(b > a)
print(type(b > a))
```

```
True
<class 'bool'>
```

Logical tests

On many occasions we seek to compare different values, usually using logical tests.

```
print(5 > 3)
```
```
True
```
Is 5 greater than 3

```
print(3 > 3)
```
```
False
```
Is 3 greater than 3

```
print(3 >= 3)
```
```
True
```
Is 3 greater or equal to 3

```
print(5 == 3)
```
```
False
```
Is 5 equal to 3

```
print(3 == 3)
```
```
True
```
Is 3 equal to 3

```
print("A" == "a")
```
```
False
```
Is the letter A equal to the letter a

```
print("A" != 3)
```
```
True
```
Is the letter A not equal to 3

```
print(3.0 == 3)
```
```
True
```
Is 3.0 equal to 3

Reserved Keywords in Python

Although we have infinite options for variable names, there are a few reserved words which cannot be defined differently.

Keywords in Python programming language				
False	await	else	import	pass
None	break	except	in	raise
True	class	finally	is	return
and	continue	for	lambda	try
as	def	from	nonlocal	while
assert	del	global	not	with
async	elif	if	or	yield

```
True = 3
```

```
  File "<ipython-input-25-679355094e21>", line 1
    True = 3
    ^
SyntaxError: cannot assign to True
```

If and else Statements

- Let’s say that that your bus to work arrives exactly at 09:00, so you must wake up before 08:30 in order to get ready. Sadly, from time to time waking up in the morning doesn’t go so smoothly and you wake up after 08:30. When this is the case you must leave instantly so you don’t miss the bus. Otherwise, you just get up, consume your precious morning coffee and leave to work.
- How can we describe this ‘process’ in Python?

```
1 if wake_up_time > "08:30":
2     print("Run Forrest run!")
3 else:
4     print("All good, get yourself some coffee")
```

Example continued

- Now, assume that your day off is on Wednesday so you can wake up later that day. How can we consider this option as well?

```
if Day == "Wednesday":
    print("Go back to sleep")
else:
    if wake_up_time > "08:30":
        print("Run Forrest run!")
    else:
        print("All good, get yourself some coffee")
```

- Or equivalently,

```
if Day == "Wednesday":
    print("Go back to sleep")
elif wake_up_time > "08:30":
    print("Run Forrest run!")
else:
    print("All good, get yourself some coffee")
```

Meaningful Variable Names and Comments

- Anytime we write and run our code we need to keep in mind that someday someone (including us) will go over this code in order to use it, do some modifications and many other reasons.
- Using meaningful variable names makes the code much more interpretable and ‘user friendly’.
- Example – consider the following chunk of code and try to understand what’s its goal.

```
1  d1 = 0.5*a + 0.3*b + 0.2*c
2  d2 = 0.8*a + 0.2*c
3  if d1 < d2:
4      d = d2
5  else:
6      d = d1
```

- Now, check out the following version:

```
Final1 = 0.5*Final_test + 0.3*Mid_term + 0.2*HW
Final2 = 0.8*Final_test + 0.2*HW
if Final1 <= Final2:
    Final = Final2
else:
    Final = Final1
```

- Another helpful tool is comments. Using the # sign we can add comments to our code and make it easier to interpret by others (and by us if we forget what we did).
- Python will ignore everything from the location of the # sign up to the end of the row.

```
Final1 = 0.5*Final_test + 0.3*Mid_term + 0.2*HW # Final grade considering the mid term score
Final2 = 0.8*Final_test + 0.2*HW # Final grade without considering the mid term score
if Final1 <= Final2: # If the final grade is higher without mid term
    Final = Final2   # we set it to be Final2
else:
    Final = Final1   # Otherwise, Final1 is the course's final grade
```

Lists

- Lists are used to store multiple items in a single variable.
- We define lists using [ ] and separate between different items using a comma.
- Lists can store items of different types!

```
1 li = ["Buy milk", 1, -3.7, True]
2 print(li)
3 print(type(li))
```

```
['Buy milk', 1, -3.7, True]
<class 'list'>
```

Lists - continued

- Indexing – By using parentheses once again we can approach specific values withing the list.
- Beware! The first element's index is 0, so don't get confused.

```
1 print(li[0]) # Printing the first element
2 print(li[2]) # Printing the third element
```

```
Buy milk
-3.7
```

- We can even approach multiple values at once by specifying a range of indices (observe that Python ignores the last index).

```
1 print(li[0:2]) # Printing the first and second element
2 print(li[2:]) # Printing the second element up to the last
3 print(li[:3]) # Printing from the start up the third element
4 print(li[-3:-1]) # Printing the third from the right up to the second last
```

```
['Buy milk', 1]
[-3.7, True]
['Buy milk', 1, -3.7]
[1, -3.7]
```

Methods

- We already noticed the output of the print(type(…)) command includes the word *class* and then the variable's type.
- Python is an *object-oriented* programming language.
- In simple words, each variable is assigned with a value and a class, and every class has its own methods.
- Methods are a set of operations that can be applied on variables with their corresponding class.
- In order to apply those methods, we write (for some variable a)

a.method()

List Methods

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

Examples

```
1 li.append("Methods!")
2 print(li)
```

```
['Buy milk', 1, -3.7, True, 'Methods!']
```

```
1 print(li.index(1))
2 print(li.index(-3.7))
```

```
1
2
```

```
1 li.reverse()
2 print(li)
```

```
['Methods!', True, -3.7, 1, 'Buy milk']
```

```
1 li1 = [1, -3, 10, -2, 4.356]
2 li1.sort()
3 print(li1)
```

```
[-3, -2, 1, 4.356, 10]
```

```
1 li2 = [1, 2, 1, 4, 10, -4, "cat"]
2 print(li2.count(1))
3 print(li2.count("cat"))
4 print(li2.count(5))
```

```
2
1
0
```

```
1 li2.insert(1, 0)
2 print(li2)
```

```
[1, 0, 2, 1, 4, 10, -4, 'cat']
```

Tuples and Dictionaries

- Additional data types that can be used to store many items are tuples and dictionaries.
- Tuples are quite identical to lists and defined using ( ) with the exception that they're an immutable object, i.e. we can change items values when they are stored in lists but can't when they are stored in tuples.

```
1 list1 = [1, 2, 3]
2 list1[1] = 10 # Assigning different value to the second element in list1
3 print(list1)
```

```
[1, 10, 3]
```

```
1 tuple1 = (1, 2, 3)
2 tuple1[1] = 10 # Assigning different value to the second element in tuple1
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-44-0681bb54a3ac> in <module>
      1 tuple1 = (1, 2, 3)
----> 2 tuple1[1] = 10 # Assigning different value to the second element in tuple1

TypeError: 'tuple' object does not support item assignment
```

- Dictionaries are more ‘complicated’ objects which consist of Key:Value pairs. Each item in the dictionary has a key with corresponding values.

```
1 dict1 = {"Car": ["Toyota Corolla", "Mazda 3", "BMW X6"],
2          "Price": [130, 115, 200]}
3 print(dict1)
```

```
{'Car': ['Toyota Corolla', 'Mazda 3', 'BMW X6'], 'Price': [130, 115, 200]}
```

- Keys can be numbers, strings and more, where values can be almost every object.
- We won’t dwell on these objects, but we will encounter them in the upcoming days.

Nested lists

- Lists items can be lists as well, i.e. it is possible to have a list like that

[1, 4, 10, -3, “cat”, [4, -3, “Nested”], 0]

- Observe that the sixth element is a list as well. We can use this to store for examples pairs of [Student, Grade]

```
1 Grades = [["Student 1", 100], ["Student 2", 90], ["Student 3", 75], ["Student 4", 83]]
2 print(Grades)
```

```
[['Student 1', 100], ['Student 2', 90], ['Student 3', 75], ['Student 4', 83]]
```

Loops

- Assume we have a list of numbers representing students' grades and we wish to know how many of them passed the course (grade higher than 55).
- Sadly, we can't do things like this:

```
1 Grades = [85, 67, 90, 100, 40, 26, 58, 32, 100, 21, 74, 30, 52, 49, 1, 84, 46]
2 print(Grades > 55)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-55-86fc21d074d0> in <module>
      1 Grades = [85, 67, 90, 100, 40, 26, 58, 32, 100, 21, 74, 30, 52, 49, 1, 84, 46]
----> 2 print(Grades > 55)

TypeError: '>' not supported between instances of 'list' and 'int'
```

- Obviously, we can count by hand how many passed the course but what will we do when we have 100 students? 1000?
- Using *for loops* we can iterate through the list's elements and check for each one of them whether the student passed or not.

```
for grade in Grades: # remark - the name grade is arbitrary
    print(grade)
```

```
85
67
90
100
40
26
58
32
100
21
74
30
52
49
1
84
46
```

```
num_of_passes = 0
for grade in Grades:
    if grade > 55:
        num_of_passes += 1 # equivalent to num_of_passes = num_of_passes + 1
print(num_of_passes)
```

```
8
```

- We can also iterate across nested lists!

```
Grades = [["Student 1", 100], ["Student 2", 90], ["Student 3", 75], ["Student 4", 83]]
for student, grade in Grades:
    print(student, "Final grade is", grade)
```

```
Student 1 Final grade is 100
Student 2 Final grade is 90
Student 3 Final grade is 75
Student 4 Final grade is 83
```

Range(start, stop, step)

- Range is a function that is usually combined with for loops and enables us to iterate across a range of numbers which start at *start*, ends at *stop* (not including) with steps of *step* (The default is *step*=1).

```
for i in range(0, 10):
    print(i, end = " ") # end = " " enable printing in a row, without line breaks
```

```
0 1 2 3 4 5 6 7 8 9
```

```
for i in range(0, 10, 2):
    print(i, end = " ") # end = " " enable printing in a row, without line breaks
```

```
0 2 4 6 8
```

```
for i in range(-5, 3, 3):
    print(i, end = " ") # end = " " enable printing in a row, without line breaks
```

```
-5 -2 1
```

```
for i in range(10, 0, -1):
    print(i, end = " ") # end = " " enable printing in a row, without line breaks
```

```
10 9 8 7 6 5 4 3 2 1
```

- Note: *range(n)* is equivalent to *range(0,n).*

Creating lists using *for loop*

- Consider we are given a list of grades and we wish to create another one, where each value in the new list is 1 if the corresponding students passed the course, and 0 otherwise.

```
Grades = [85, 67, 90, 100, 40, 26, 58, 32, 100, 21, 74, 30, 52, 49, 1, 84, 46]
Pass = []
for i in range(len(Grades)):
    Pass.append(int(Grades[i] >= 55))
print(Grades)
print(Pass)
```

```
[85, 67, 90, 100, 40, 26, 58, 32, 100, 21, 74, 30, 52, 49, 1, 84, 46]
[1, 1, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0]
```

List Comprehension

- Instead of predefining an empty list and then iteratively filling it, we can do all of that in one row:

```
a = [x for x in range(1, 11)]
print(a, "\n---------------")

a = [x for x in range(1, 11) if x % 2 == 0]
print(a, "\n---------------")

a = [[x,x**2] for x in range(1, 11) if x % 2 == 0]
print(a, "\n---------------")

import math
a = [[x,round(math.log(x**2),3)] for x in range(1, 11) if x % 2 == 0]
print(a, "\n---------------")
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
---------------
[2, 4, 6, 8, 10]
---------------
[[2, 4], [4, 16], [6, 36], [8, 64], [10, 100]]
---------------
[[2, 1.386], [4, 2.773], [6, 3.584], [8, 4.159], [10, 4.605]]
---------------
```

```
Pass1 = [int(Grades[i] >= 55) for i in range(len(Grades))]
print(Pass1)
```

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

Functions

- Function apply some operation. We already encountered *print()* and *type()*. Another example is the function *sum()* which computes the sum of elements inside a list (or tuple)

```
print(sum([1,10,-2,3,4])) # sum of list
print(sum((1,10,-2,3,4))) # sum of tuple
```

```
16
16
```

- Another useful function is *round()*

```
print(round(4.28601, 3))
```

```
4.286
```

- *len()* takes one argument (for example a list) and returns the number of elements in it.
- If the argument is a string, the function returns the number of characters.

```
li = [1, 2, 4, 5, 7, 10]
print(len(li))
```

```
6
```

```
print(len("SICSS 2022"))
```

```
10
```

Functions - continued

- Functions are divided into 4 groups:
  - *With* argument(s) and *with* return value(s)
  - *Without* argument(s) and *with* return value(s)
  - *With* argument(s) but *without* return value(s)
  - *Without* argument(s) and *without* return value(s)
- For example, the function *sum()* takes 1 argument, a list or tuple and returns one value which is the sum of elements.
- The function *round()* takes 2 arguments, a number and integer *k* and returns the number rounded up to *k* decimals.

Defining functions

- Occasionally, we wish to perform operations that have no predefined function.
- Fortunately, we can define our own function.
- For example, we wish to create a function that calculates the final grade of a student given its final test, mid term and HW scores.
- This function has 3 arguments and returns one value – the final grade.

```
def Calc_final_grade(Final_test, Mid_term, HW):
    Final1 = 0.5*Final_test + 0.3*Mid_term + 0.2*HW # Final grade considering the mid term score
    Final2 = 0.8*Final_test + 0.2*HW # Final grade without considering the mid term score
    if Final1 <= Final2: # If the final grade is higher without mid term
        Final = Final2   # we set it to be Final2
    else:
        Final = Final1   # Otherwise, Final1 is the course's final grade
    return Final
```

```
print(Calc_final_grade(85, 90, 100))
print(Calc_final_grade(85, 50, 100))
print(Calc_final_grade(85, 10, 100))
```

```
89.5
88.0
88.0
```

- We can also create the same function but without a return value.

```
def Calc_final_grade_print(Final_test, Mid_term, HW):
    Final1 = 0.5*Final_test + 0.3*Mid_term + 0.2*HW # Final grade considering the mid term score
    Final2 = 0.8*Final_test + 0.2*HW # Final grade without considering the mid term score
    if Final1 <= Final2: # If the final grade is higher without mid term
        Final = Final2   # we set it to be Final2
    else:
        Final = Final1   # Otherwise, Final1 is the course's final grade
    print(Final)
```

```
Calc_final_grade_print(85, 90, 100)
Calc_final_grade_print(85, 50, 100)
Calc_final_grade_print(85, 10, 100)
```

```
89.5
88.0
88.0
```

- Function can be defined also without arguments.

```
def no_arg_fun():
    return 5**5 # this function just return the value 5 to the power of 5
print(no_arg_fun())
```

```
3125
```

Default arguments

- Assume that we wish to define a function that gets one numerical variable, an integer *k* and return its absolute value, rounded up to *k* decimals, but if *k* is not specified then we will use 3 by default.

```
1 def round_abs(num, k = 3):
2     return round(abs(num), k)
```

```
1 print(round_abs(-3.234825))
2 print(round_abs(-3.234825, 3))
3 print(round_abs(-3.234825, 5))
```

```
3.235
3.235
3.23482
```

Modules

- A module is a collection of additional function and objects.
- The *math* module includes mathematical functions such as *sqrt, exp, cos, sin* etc…
- The *random* module includes many functions that enable us to generating pseudo-random numbers.
- Later in this course we will focus mainly on the *pandas* and *numpy* modules.

Importing modules

- In order to use the contents of different modules, we need to *import* them using the syntax:

module.function()

```
1 import math
2 math.sqrt(4)
```

```
2.0
```

- We can also import only a specific function and then we don't need to specify the module name when calling the function.

```
1 from math import cos
2 cos(3)
```

```
-0.9899924966004454
```

- Sometimes modules have long names, or we are using their content a lot, so writing their name constantly can be very exhausting, hence we can “modify” the module’s name after importing it.

```
1 import math as m
2 m.sqrt(4)
```

```
2.0
```

```
1 import random as r
2 print(r.random()) # pseudo - random number between 0 to 1
```

```
0.9455770542185054
```

Jupyter Time!