ILP 2022 – W4S3
Numpy library (part 1) and imports

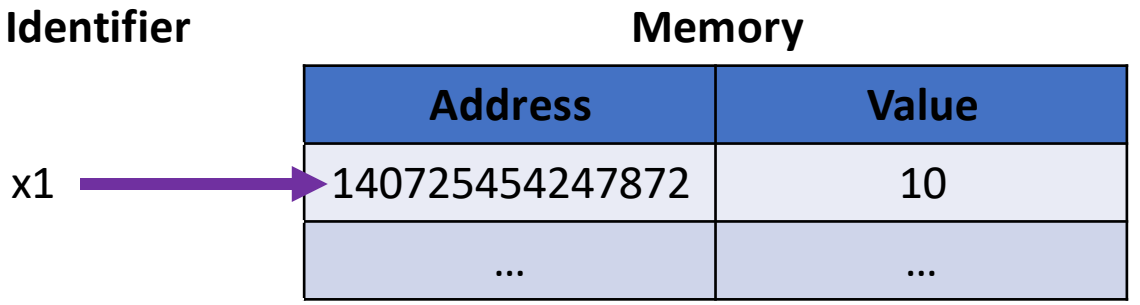
Matthieu DE MARI – Singapore University of Technology and Design

Outline (Week4, Session3 – W4S3)

- Memory management and lists: aliasing, shallow and deep copies
- The Numpy library (part 1): arrays, math functions, etc.
- About the import procedure
- Project organizing
- Mini-project

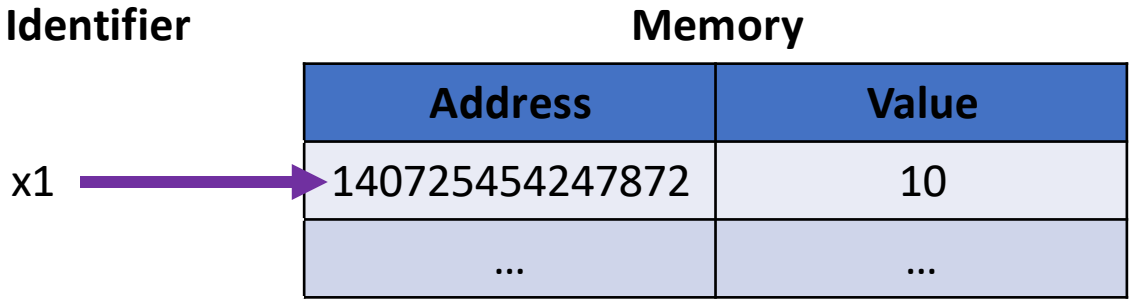
Memory of a computer

- The memory of a computer consists of several “boxes”, which can contain values for variables. Each “box” is identified by an **integer**, which corresponds to the **address/ID** of the “box”.



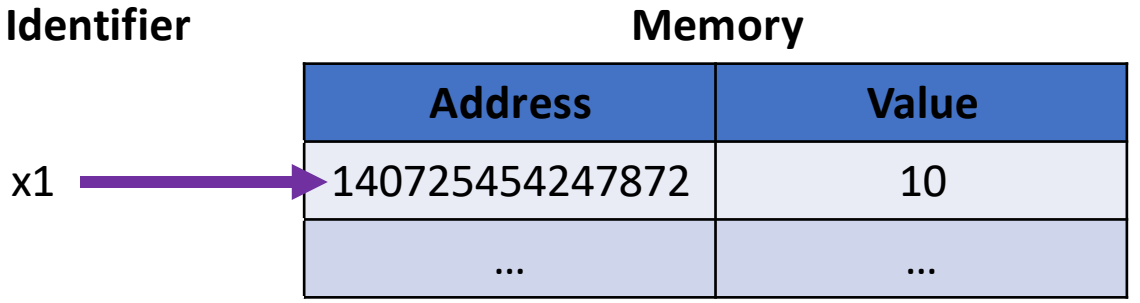
Memory of a computer

- The memory of a computer consists of several “boxes”, which can contain values for variables. Each “box” is identified by an **integer**, which corresponds to the **address/ID** of the “box”.
- When a variable is created:
  - A “box” is assigned for the variable and its value is stored in the “box”.
  - The variable name simply refers to the address/ID of the “box”.



Memory of a computer

- The memory of a computer consists of several “boxes”, which can contain values for variables. Each “box” is identified by an **integer**, which corresponds to the **address/ID** of the “box”.
- When a variable is created:
  - A “box” is assigned for the variable and its value is stored in the “box”.
  - The variable name simply refers to the address/ID of the “box”.




```
1 x1 = 10
2 print(x1)
3 print(id(x1))
```

```
10
140725454247872
```

Recall: the id() function

- The **id()** function returns an integer, which corresponds to the **address/ID**, where the variable is stored in memory.

```
1  x1 = 10
2  print(x1)
3  print(id(x1))
```

```
10
140725454247872
```

```
1  x2 = 17
2  print(x2)
3  print(id(x2))
```

```
17
140725454248096
```

Recall: the id() function

- The **id()** function returns an integer, which corresponds to the **address/ID**, where the variable is stored in memory.
- Two variables names with identical values will have the same **id().**

```
1  x1 = 10
2  print(x1)
3  print(id(x1))
```

```
10
140725454247872
```

```
1  x2 = 17
2  print(x2)
3  print(id(x2))
```

```
17
140725454248096
```

```
1  x3 = x1
2  print(x3)
3  print(id(x3))
4  print(id(x1))
```

```
10
140725454247872
140725454247872
```

Recall: the id() function

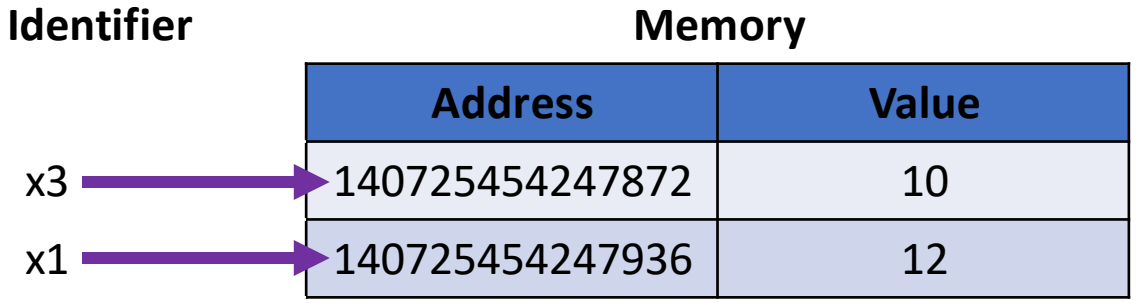
- The **id()** function returns an integer, which corresponds to the **address/ID**, where the variable is stored in memory.
- Two variables names with identical values will have the same **id().**
- **Aliasing:** Python saves memory space by having two variables names **point to the same memory ID**.


```
1  x3 = x1
2  print(x3)
3  print(id(x3))
4  print(id(x1))
```

```
10
140725454247872
140725454247872
```

Recall: the id() function

- The **id()** function returns an integer, which corresponds to the **address/ID**, where the variable is stored in memory.
- Two variables names with identical values will have the same **id().**
- **Aliasing:** Python saves memory space by having two variables names **point to the same memory ID**.




```
1  x1 = 12
2  print(x1)
3  print(x3)
4  print(id(x1))
5  print(id(x3))
```

```
12
10
140725454247936
140725454247872
```

Memory management in lists

- A **list** is a collection of variables.

```
1  list1 = [x1, x2, x3]
2  print(list1)
3  print(id(list1))
```

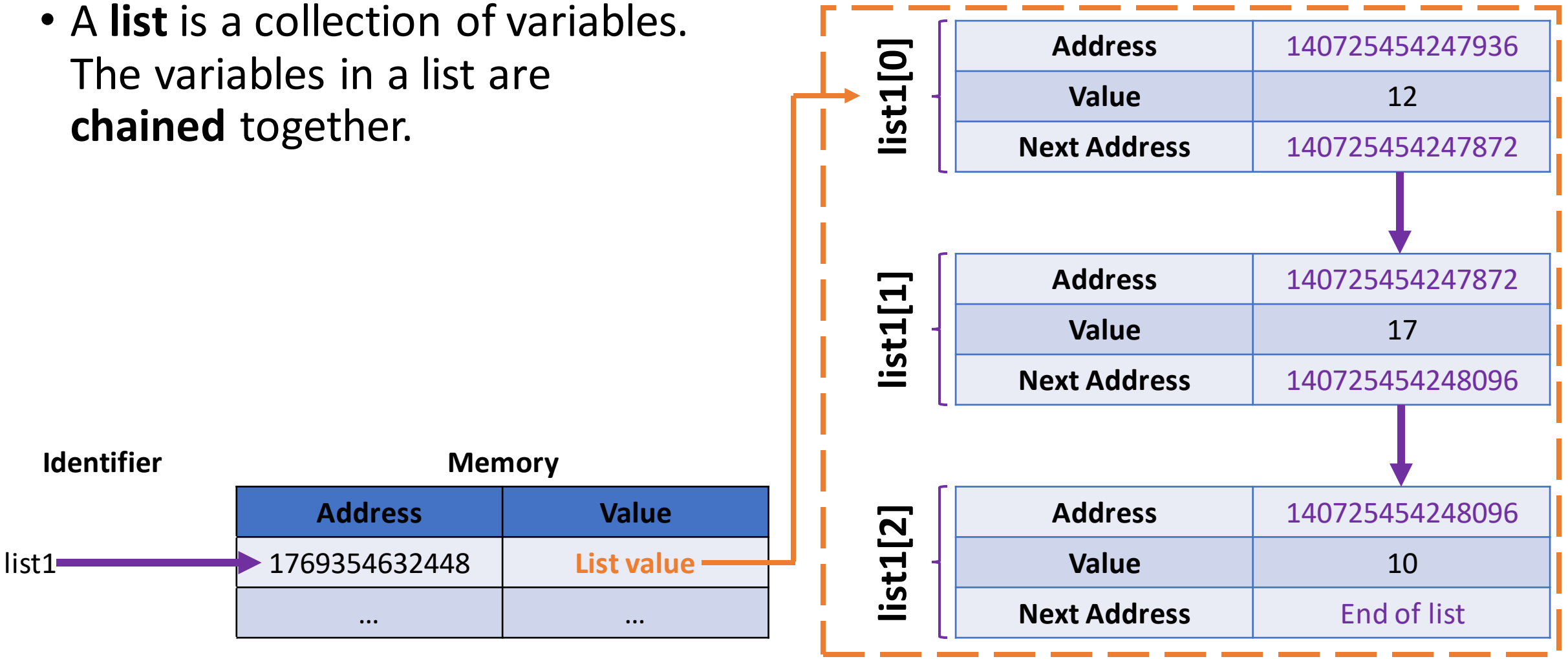
```
[12, 17, 10]
1769354632448
```

```
1  print(id(list1))
2  print("-")
3  print(id(list1[0]))
4  print(id(x1))
5  print("-")
6  print(id(list1[1]))
7  print(id(x2))
8  print("-")
9  print(id(list1[2]))
10 print(id(x3))
```

```
1769354632448
-
140725454247936
140725454247936
-
140725454248096
140725454248096
-
140725454247872
140725454247872
```

Memory management in lists

- A **list** is a collection of variables. The variables in a list are **chained** together.



Memory management in lists

- A **list** is a collection of variables. The variables in a list are **chained** together.
- If x1 is changed, Python will adjust so that the list remains unaffected.
- It simply reallocates x1 to another location in memory.

```
1  print(x1)
2  print(id(x1))
3  x1 = "Hello"
4  print(id(x1))
5  print(list1)
```

```
12
140726382041088
2120755150000
[12, 17, 10]
```

Aliasing in lists: problem

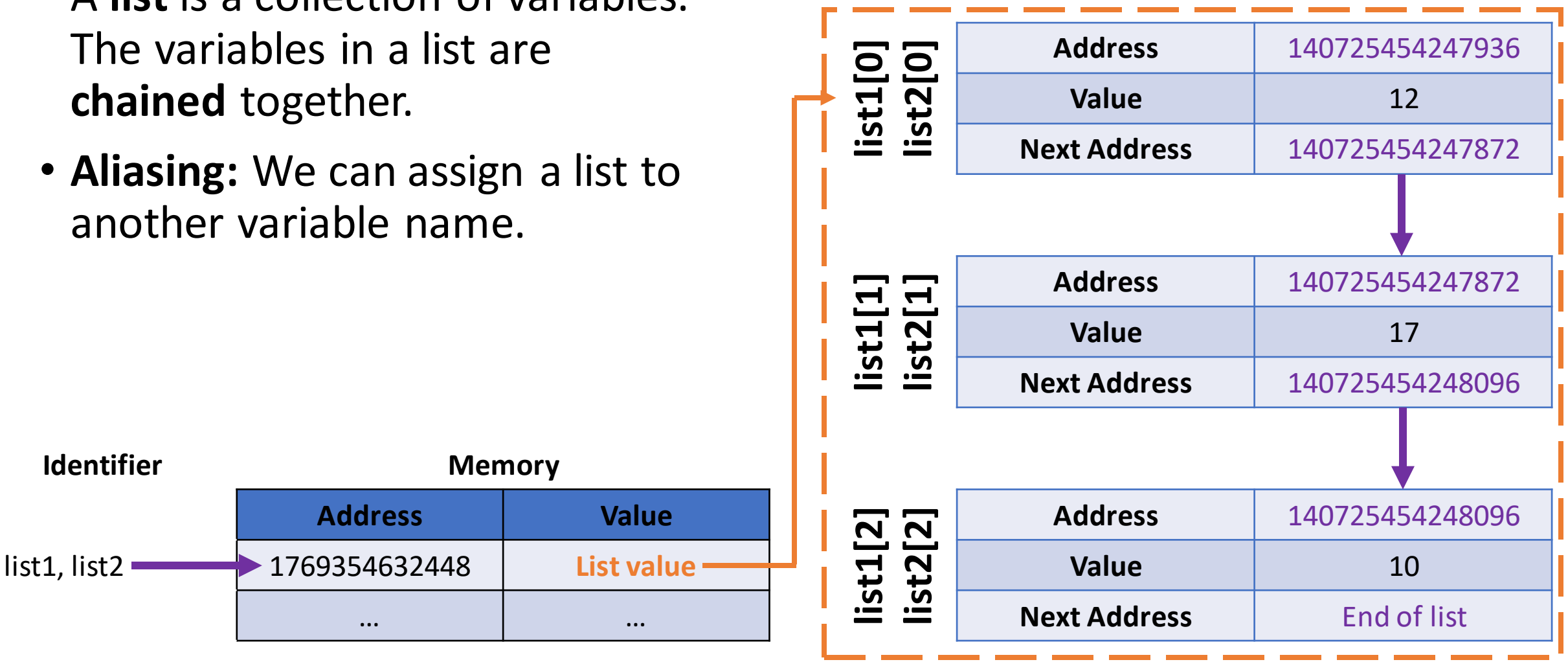
- A **list** is a collection of variables. The variables in a list are **chained** together.
- **Aliasing:** We can assign a list to another variable name.

```
1  list2 = list1
2  print(list2)
3  print(id(list1))
4  print(id(list2))
```

```
[12, 17, 10]
1983742564288
1983742564288
```

Aliasing in lists: problem

- A **list** is a collection of variables. The variables in a list are **chained** together.
- **Aliasing:** We can assign a list to another variable name.



Aliasing in lists: problem

- A **list** is a collection of variables. The variables in a list are **chained** together.
- **Aliasing:** We can assign a list to another variable name.
- **Problem:** changing list1[0] changes list1 values, but also changes list2.

```
1 print(id(list1[0]))
2 list1[0] = "SUTD"
3 print(list1)
4 print(id(list1[0]))
```

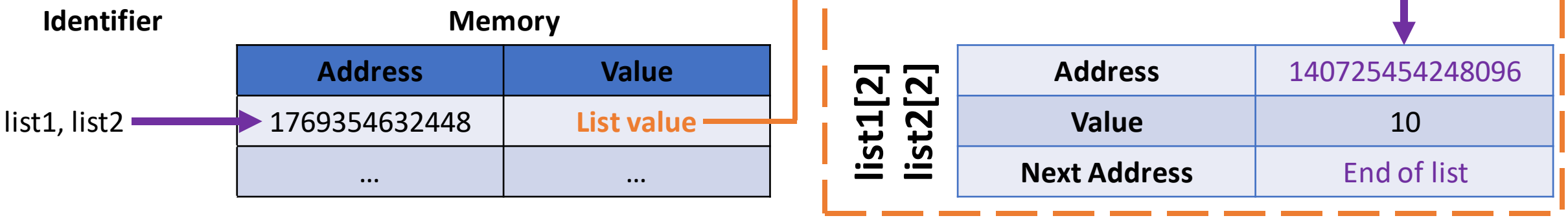
```
140726382041088
['SUTD', 17, 10]
2120755353584
```

```
1 print(list2)
2 print(id(list2[0]))
```

```
['SUTD', 17, 10]
2120755353584
```

Aliasing in lists: problem

- A **list** is a collection of variables. The variables in a list are **chained** together.
- **Aliasing:** We can assign a list to another variable name.
- **Problem:** changing list1[0] changes list1 values, but also changes list2.



Shallow copy of a list

- **Problem:** changing list1[0] changes list1 values, but also changes list2.
- **Shallow copy:** list1[:] makes list2 a shallow copy of list1. By doing so, list2 will be saved to its own location of memory.

```
1  list1 = [12, 17, 10]
2  list2 = list1[:]
3  print(list1)
4  print(list2)
5  print(id(list1))
6  print(id(list2))
```

```
[12, 17, 10]
[12, 17, 10]
2120755431296
2120755345920
```

Shallow copy of a list

- **Problem:** changing list1[0] changes list1 values, but also changes list2.
- **Shallow copy:** list1[:] makes list2 a shallow copy of list1. By doing so, list2 will be saved to its own location of memory.
- Changing a value in list1, with list1[index] = …, no longer affects list2.
- **Note:** you can also use the copy() method.

```
1 list1 = [12, 17, 10]
2 list2 = list1[:]
3 print(list1)
4 print(list2)
5 print(id(list1))
6 print(id(list2))
```

```
[12, 17, 10]
[12, 17, 10]
2120755431296
2120755345920
```

```
1 list1[0] = "SUTD"
2 print(list1)
3 print(list2)
```

```
['SUTD', 17, 10]
[12, 17, 10]
```

Shallow copy: problem

- **Note:** if an element of a list is a list (case of lists of lists), then the shallow copy will not copy the sublists to different locations of memory.
- **Problem:** changing a sublist element then affects both lists, even though these lists are shallow copies of each other.

```
1  list1 = [[8, 9, 11], 7, 4]
2  list2 = list1[:]
3  print(list1)
4  print(list2)
5  print(id(list1))
6  print(id(list2))
7  print(id(list1[0][1]))
8  print(id(list2[0][1]))
```

```
[[8, 9, 11], 7, 4]
[[8, 9, 11], 7, 4]
2120755333248
2120755332928
140726382040992
140726382040992
```

```
1  list1[0][1] = "Damn it!"
2  print(list1)
3  print(list2)
```

```
[[8, 'Damn it!', 11], 7, 4]
[[8, 'Damn it!', 11], 7, 4]
```

Deep copy

- **Solution:** make a **deep copy**, using the Python built-in **copy** library.

- A deep copy forces Python to make sure **all elements and sub-elements** are assigned to different locations in memory.

```
1 from copy import deepcopy
```

```
1 list1 = [[8, 9, 11], 7, 4]
2 list2 = deepcopy(list1)
3 print(list1)
4 print(list2)
5 print(id(list1[0]))
6 print(id(list2[0]))
```

```
[[8, 9, 11], 7, 4]
[[8, 9, 11], 7, 4]
2120754851072
2120754850304
```

```
1 list1[0][1] = "Deep copy works?"
2 print(list1)
3 print(list2)
4 print(id(list1[0][1]))
5 print(id(list2[0][1]))
```

```
[[8, 'Deep copy works?', 11], 7, 4]
[[8, 9, 11], 7, 4]
2120755409424
140726382040992
```

Matt’s Great advice #10

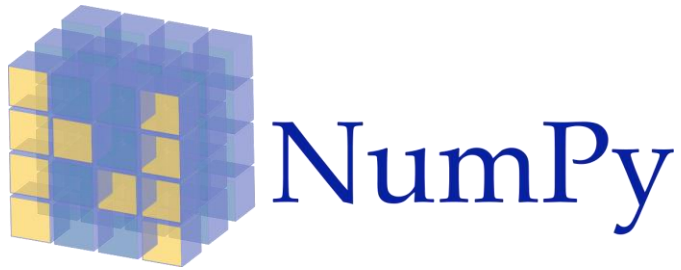
Matt’s Great Advice #10: Keep the aliasing, shallow and deep copies concepts in mind for now.

If you find that modifying a list object ends up unexpectedly changing another, then you might have an aliasing or shallow copy problem.

When in doubt, make a deep copy.

For now, do not worry about understanding all these memory concepts, these will be covered in another advanced course!

About the numpy library



- **Numpy** is one of the most common (if not the most popular) **libraries** in Python.

- Used for many applications: computing, modelling, data science, astrophysics, etc.
- **Linear algebra** (one of the main concepts of math with many applications in computing)

A new type of objects: Numpy Arrays

- **Numpy arrays** are objects from the **Numpy** library.
  - Typically used to describe **matrices** and **vectors,**
  - Or **tables** of data.
- The look very similar to (**nested lists of) lists**, which we have used earlier for many applications.
- The Numpy library, however, comes with many additional functions and methods.

```
1  import numpy as np
```

```
1  array1 = np.array([0, 2, 1, 4])
2  print(array1)
```

```
[0 2 1 4]
```

```
1  print(array1)
2  print(type(array1))
3  array1_as_list = list(array1)
4  print(array1_as_list)
5  print(type(array1_as_list))
```

```
[0 2 1 4]
<class 'numpy.ndarray'>
[0, 2, 1, 4]
<class 'list'>
```

Length, size, shape

- Just like lists, the Numpy arrays have a length, which can be checked with **len()**.
- The also have a **shape** and a **size attribute**, which give additional information, in the case of arrays with more than 1D.
- **Attribute: "sub-variable"** of an object; applies to an object using the **.** operator. (Object-oriented concept, not covered yet)

```
1  array1 = np.array([0, 2, 1, 4])
2  print(len(array1))
3  print(array1.shape)
4  print(array1.size)
```

```
4
(4,)
4
```

```
1  two_d_array = np.array([[1,2],[3,4]])
2  print(two_d_array)
3  print(len(two_d_array))
4  print(two_d_array.shape)
5  print(two_d_array.size)
```

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

Indexing an array

- Just like lists, the Numpy arrays are indexed and their element can be accessed with **[]**.
- You can equivalently use the **[i,j]** and **[i][j]** notations on arrays.
- Replacing an index with a colon symbol **:**, means “take all”.
- For instance, **[:, j]** means **all elements in column j**, whereas **[i, :]** means **all elements in row i**.

```
1 array1 = np.array([0, 2, 1, 4])
2 print(array1)
3 print(array1[0])
4 print(array1[1])
```

```
[0 2 1 4]
0
2
```

```
1 two_d_array = np.array([[1,2],[3,4]])
2 print(two_d_array)
3 print(two_d_array[0])
4 print(two_d_array[0][1])
```

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

```
1 print(two_d_array[0,1])
2 print(two_d_array[:,1])
3 print(two_d_array[0,:])
```

```
2
[2 4]
[1 2]
```

Traversing an array with for

- As with **lists**, we can traverse a Numpy array, in an element-wise manner, using a **for** loop.

```
1  array1 = np.array([0, 2, 1, 4])
2  print(array1)
3  for element in array1:
4      print(element)
```

```
[0 2 1 4]
0
2
1
4
```

```
1  my_list = [1, 4, 9, 14, 15]
2  print(my_list)
```

```
[1, 4, 9, 14, 15]
```

```
1  # Element-wise
2  for element in my_list:
3      print("--")
4      print(element)
```

```
--
1
--
4
--
9
--
14
--
15
```

The + operator on arrays

```
1 a_list = [0, 1, 2]
2 another_list = [1, 4, 7]
3 list_sum = a_list + another_list
4 print(list_sum)
```

```
[0, 1, 2, 1, 4, 7]
```

- **The + operator on lists**: On lists, the + operator will **<u>concatenate</u>** both lists into a new one.

The + operator on arrays

- **The + operator on lists**: On lists, the + operator will **concatenate** both lists into a new one.
- **The + operator on Numpy arrays – (vector sum):** On Numpy arrays, however, the + operator will **sum** the elements of both Numpy arrays.

```
1 a_list = [0, 1, 2]
2 another_list = [1, 4, 7]
3 list_sum = a_list + another_list
4 print(list_sum)
```

```
[0, 1, 2, 1, 4, 7]
```

```
1 array1 = np.array([0, 2, 1, 4])
2 array2 = np.array([1, 2, 3, 5])
3 print(array1)
4 print(array2)
5 sum_array = array1 + array2
6 print(sum_array)
```

```
[0 2 1 4]
[1 2 3 5]
[1 4 4 9]
```

The + operator on arrays

- **The + operator on lists**: On lists, the + operator will **concatenate** both lists into a new one.
- **The + operator on Numpy arrays – (vector sum):** On Numpy arrays, however, the + operator will **sum** the elements of both Numpy arrays.
- **Broadcasting:** If summed with a number instead, the elements in the Numpy array will each be incremented by the given value.

```
1 a_list = [0, 1, 2]
2 another_list = [1, 4, 7]
3 list_sum = a_list + another_list
4 print(list_sum)
```

```
[0, 1, 2, 1, 4, 7]
```

```
1 array1 = np.array([0, 2, 1, 4])
2 array2 = np.array([1, 2, 3, 5])
3 print(array1)
4 print(array2)
5 sum_array = array1 + array2
6 print(sum_array)
```

```
[0 2 1 4]
[1 2 3 5]
[1 4 4 9]
```

```
1 array1 = np.array([0, 2, 1, 4])
2 number = 7
3 print(array1)
4 sum_array = array1 + number
5 print(sum_array)
```

```
[0 2 1 4]
[ 7  9  8 11]
```

Concatenation on arrays

- Since the **+** operator cannot be used for **concatenation**, Numpy comes with a **concatenate()** function.

```
1  print(array1)
2  print(array2)
3  conc_array = np.concatenate([array1, array2])
4  print(conc_array)
```

```
[0 2 1 4]
[1 2 3 5]
[0 2 1 4 1 2 3 5]
```

The * operator on arrays

- The * operator behaves as the + operator on Numpy arrays.
- It consists of an **element-wise multiplication** of the elements in arrays.
- **Broadcasting:** if a Numpy array is multiplied by a number, the number will multiply each element in the array.

```
1 array1 = np.array([0, 2, 1, 4])
2 array2 = np.array([1, 2, 3, 5])
3 print(array1)
4 print(array2)
5 mult_arrays = array1*array2
6 print(mult_arrays)
```

```
[0 2 1 4]
[1 2 3 5]
[ 0  4  3 20]
```

```
1 n = 4
2 mult_array_int = array1*n
3 print(mult_array_int)
```

```
[ 0  8  4 16]
```

Additional functions

Additional Numpy functions

- **Min, max:** returns the minimal, resp. maximal, values in array.
- **Argmin, argmax:** returns the index where the minimal, resp. maximal, values are.
- **Mean, median:** returns the mean, resp. median, value for a given array.
- **Sum:** sums all the elements in the array together

```
array1 = np.array([0, 2, 1, 4, 7])
print(array1)
min_val = np.min(array1)
print(min_val)
argmin_val = np.argmin(array1)
print(argmin_val)
max_val = np.max(array1)
print(max_val)
argmax_val = np.argmax(array1)
print(argmax_val)
mean_val = np.mean(array1)
print(mean_val)
median_val = np.median(array1)
print(median_val)
summed_val = np.sum(array1)
print(summed_val)
```

```
[0 2 1 4 7]
0
0
7
4
2.8
2.0
14
```

Mathematical functions and constants

Numpy also contains

- Many **mathematical functions** (cosine, sine, logarithm, exponential, etc.)
- And many **mathematical constants** (pi, etc.)

```
1 print(np.cos(0))
2 print(np.sin(0))
3 print(np.pi)
4 print(np.log(1))
5 print(np.exp(0))
```

```
1.0
0.0
3.141592653589793
0.0
1.0
```

Aliasing, Shallow and Deep copies in arrays

- As with lists, Numpy arrays are subject to the same issues about aliasing, shallow and deep copies.

- If needed, use deep copies of the arrays.

```
1 two_d_array1 = np.array([[1,2],[3,4]])
2 two_d_array2 = two_d_array1
3 print(two_d_array1)
4 print(two_d_array2)
5 print(id(two_d_array1))
6 print(id(two_d_array2))
```

```
[[1 2]
 [3 4]]
[[1 2]
 [3 4]]
1750029422960
1750029422960
```

```
1 two_d_array1[0][0] = 17
2 print(two_d_array1)
3 print(two_d_array2)
```

```
[[17  2]
 [ 3  4]]
[[17  2]
 [ 3  4]]
```

And so much more! RTFM!

- Numpy has **many more functions and tools** to offer!
- **Random functions** (to be covered in an upcoming session, if time allows?)

- Learn more about Numpy (RTFM!) here:

https://numpy.org/doc/stable/

```
1  # Numpy.random.choice() mimics a dice roll
2  dice_roll = np.random.choice([1, 2, 3, 4, 5, 6])
3  print(dice_roll)
4  dice_roll = np.random.choice([1, 2, 3, 4, 5, 6])
5  print(dice_roll)
6  dice_roll = np.random.choice([1, 2, 3, 4, 5, 6])
7  print(dice_roll)
8  dice_roll = np.random.choice([1, 2, 3, 4, 5, 6])
9  print(dice_roll)
10 dice_roll = np.random.choice([1, 2, 3, 4, 5, 6])
11 print(dice_roll)
12 dice_roll = np.random.choice([1, 2, 3, 4, 5, 6])
13 print(dice_roll)
```

```
4
6
1
4
3
2
```

Activity 1 - Exam adjustments

- Let us assume, that I have **grades** from my students listed in some **np.array** variables, as shown below.

```
columns_labels_list = ["MidTerm Score", "FinalExam Score", "Average Score"]
students_list = ["Chris", "Oka", "Norman", "Natalie", "Tony"]
grades_table = np.array([[60, 80, 70], \ # Chris scored 60% on MidTerm, 80% on Finals, Average is 70%
                         [50, 80, 65], \
                         [40, 70, 55], \
                         [60, 70, 65], \
                         [60, 90, 75]])
print(grades_table)
```

```
[[60 80 70]
 [50 80 65]
 [40 70 55]
 [60 70 65]
 [60 90 75]]
```

Activity 1 - Exam adjustments

- Let us assume, that I have **grades** from my students listed in a **np.array**.
- The first line contains the column labels (student name, some scores) and the other lines will consist of entries regarding some of the students.
- Let us assume that, as a professor, I have decided to be lenient towards my students.
- I realized that the midterm was a bit too difficult compared to last year.
- To compensate for that, I would like to increase the scores of all students on the midterm by 50%.

Activity 1 - Exam adjustments

Write a function **grade_adjustment()**,

- which **receives** a grades table, **grades_table**,
- **increases** the **scores** of all students on the **midterm** by **50%**,
- **re-calculates** the **average score**, with the **new adjusted midterm score**,
- and then returns the **updated grades table** as its sole output.

Activity 1 - Exam adjustments

Write a function **grade_adjustment()**,

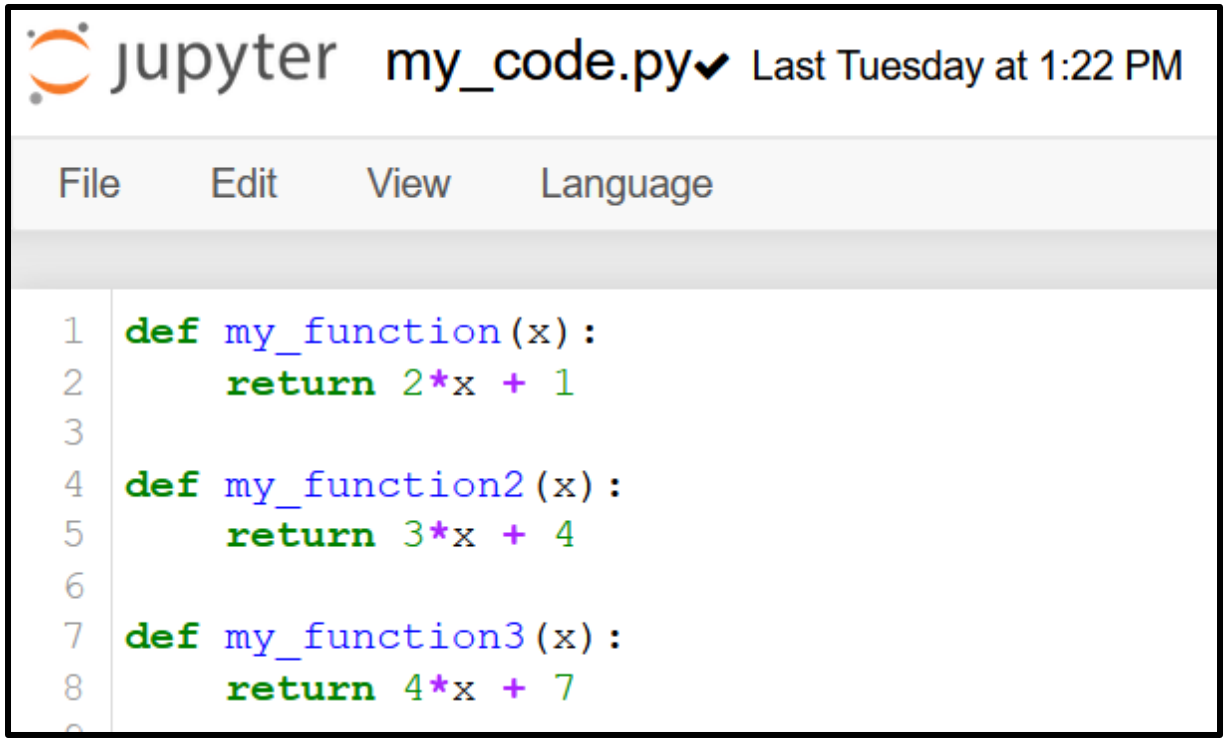
- which **receives** a grades table, **grades_table**,
- **increases** the **scores** of all students on the **midterm** by **50%**,
- **re-calculates** the **average score**, with the **new adjusted midterm score**,
- and then returns the **updated grades table** as its sole output.

- **Important note:** The **maximal score** for the midterm exam is **capped** to **100**. This means that a student which scores 80 points on the midterm, will not obtain 120 points after the adjustment, but only 100.

The import procedure

- The import procedure is used to **import functions defined in external .py files.**

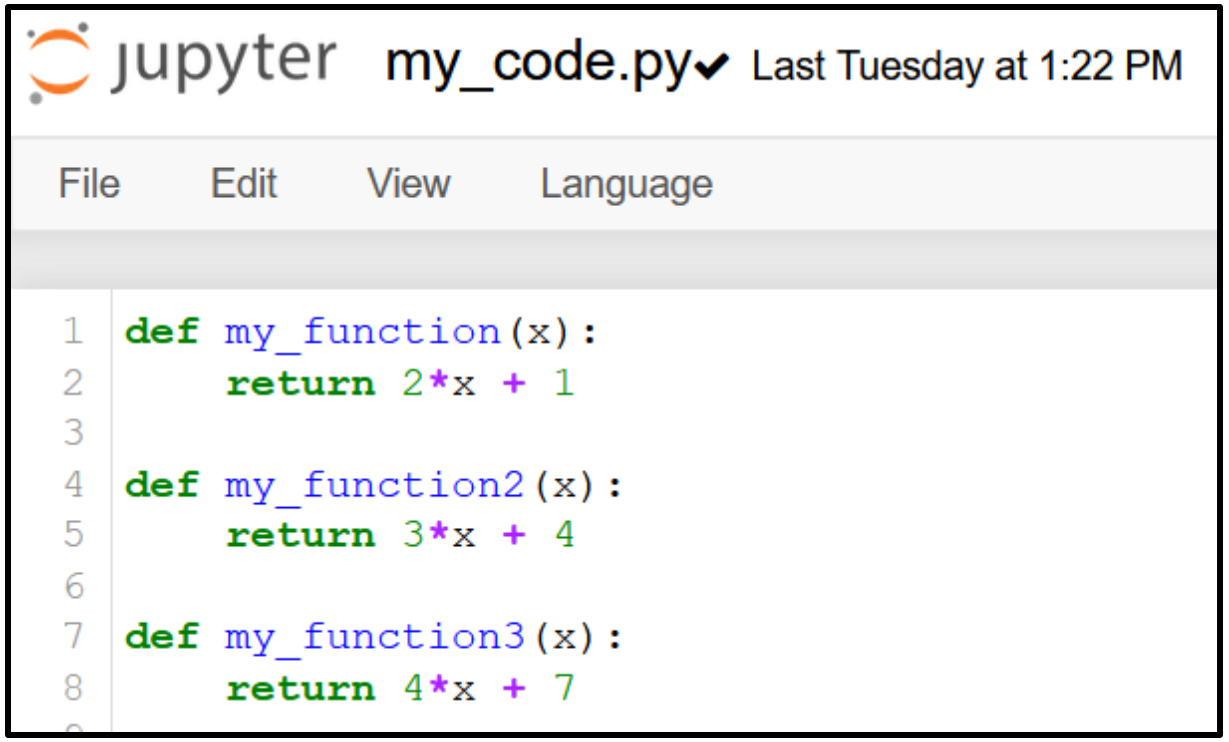
my_code.py file



The import procedure

- The import procedure is used to **import functions defined in external .py files.**
- To demonstrate, we have defined a **my_code.py** file with three functions.

my_code.py file



The import procedure

- The import procedure is used to **import functions defined in external .py files.**
- To demonstrate, we have defined a **my_code.py** file with three functions.
- We can then import one of these functions in our Notebook, by using the **from … import …** command.

my_code.py file

jupyter my_code.py ✔ Last Tuesday at 1:22 PM

File Edit View Language

```
1  def my_function(x):
2      return 2*x + 1
3
4  def my_function2(x):
5      return 3*x + 4
6
7  def my_function3(x):
8      return 4*x + 7
```

In our notebook

```
1  from my_code import my_function
```

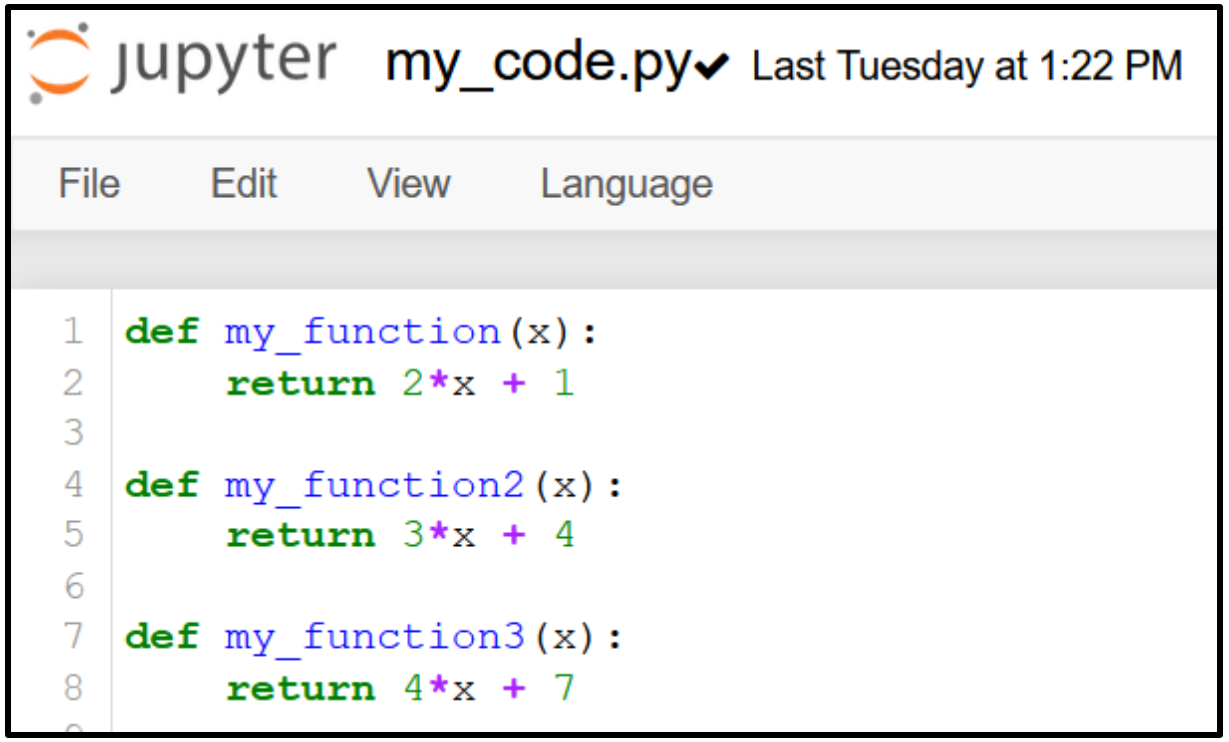
```
1  print(my_function(2))
```

```
5
```

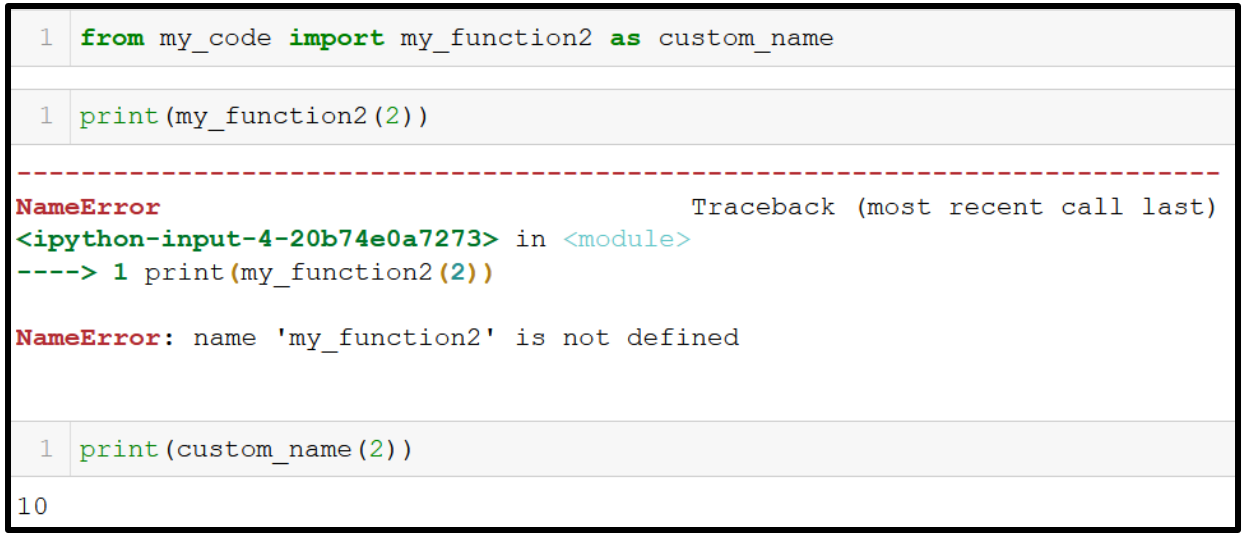
Importing as

- If needed, we can **import** and **rename** a function by using the **as** keyword.
- The whole command then reads **from … import … as …**
- **Note:** if you rename the function, its calling name changes to the alias you specified.

my_code.py file



In our notebook



Importing several functions

- If needed, you can import **multiple functions** in a single import call.
- Simply use **commas (,) symbols** to separate the different functions names.
- Or, you can also **import all functions** at once, by simply entering *.

my_code.py file

jupyter my_code.py ✔ Last Tuesday at 1:22 PM

File Edit View Language

```
1  def my_function(x):
2      return 2*x + 1
3
4  def my_function2(x):
5      return 3*x + 4
6
7  def my_function3(x):
8      return 4*x + 7
```

In our notebook

```
1  from my_code import my_function, my_function2
```

```
1  from my_code import *
```

```
1  print(my_function3(2))
```

```
15
```

Importing an entire module

- You can also import a whole file, as a **module.**
- To do so, simply start with **import** instead of **from**.
- By doing so, you import all the functions, but **they have to be called using the module name and the dot operator** (.).
- This can make the code more readable, but is also a bit more inconvenient.

more_code.py file

jupyter more_code.py ✔ Last Tuesday at 1:26 PM

File Edit View Language

```
1 def my_function4(x):
2     return x + 1
3
4 def my_function5(x):
5     return "Hello {}!".format(x)
```

In our notebook

```
1 import more_code
```

```
1 print(my_function4(2))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-9-0bc0cdce5e54> in <module>
----> 1 print(my_function4(2))

NameError: name 'my_function4' is not defined
```

```
1 print(more_code.my_function4(2))
```

```
3
```

Importing an entire module

- You can also use an alias for the module using the as keyword, as before.
- Typically, we often do it with Numpy!

import numpy as np

more_code.py file

jupyter more_code.py ✔ Last Tuesday at 1:26 PM

File Edit View Language

```
1 def my_function4(x):
2     return x + 1
3
4 def my_function5(x):
5     return "Hello {}!".format(x)
```

In our notebook

```
1 import more_code as mc
```

```
1 print(mc.my_function4(2))
```

```
3
```

```
1 import numpy as np
```

```
1 print(np.cos(0))
```

```
1.0
```

Folders to import from

- **Observation:** Python imported functions from my_code.py, which was located in the same folder as my Notebook.
- **But there was no numpy.py file in this location.**
- **What happened?**

Folders to import from

- **Observation:** Python imported functions from my_code.py, which was located in the same folder as my Notebook.
- **But there was no numpy.py file in this location.**
- **What happened?**

- Python looks for files in your current folder first,
- and then looks in your Python installation directory.

Folders to import from

- **Observation:** Python imported functions from my_code.py, which was located in the same folder as my Notebook.
- **But there was no numpy.py file in this location.**
- **What happened?**

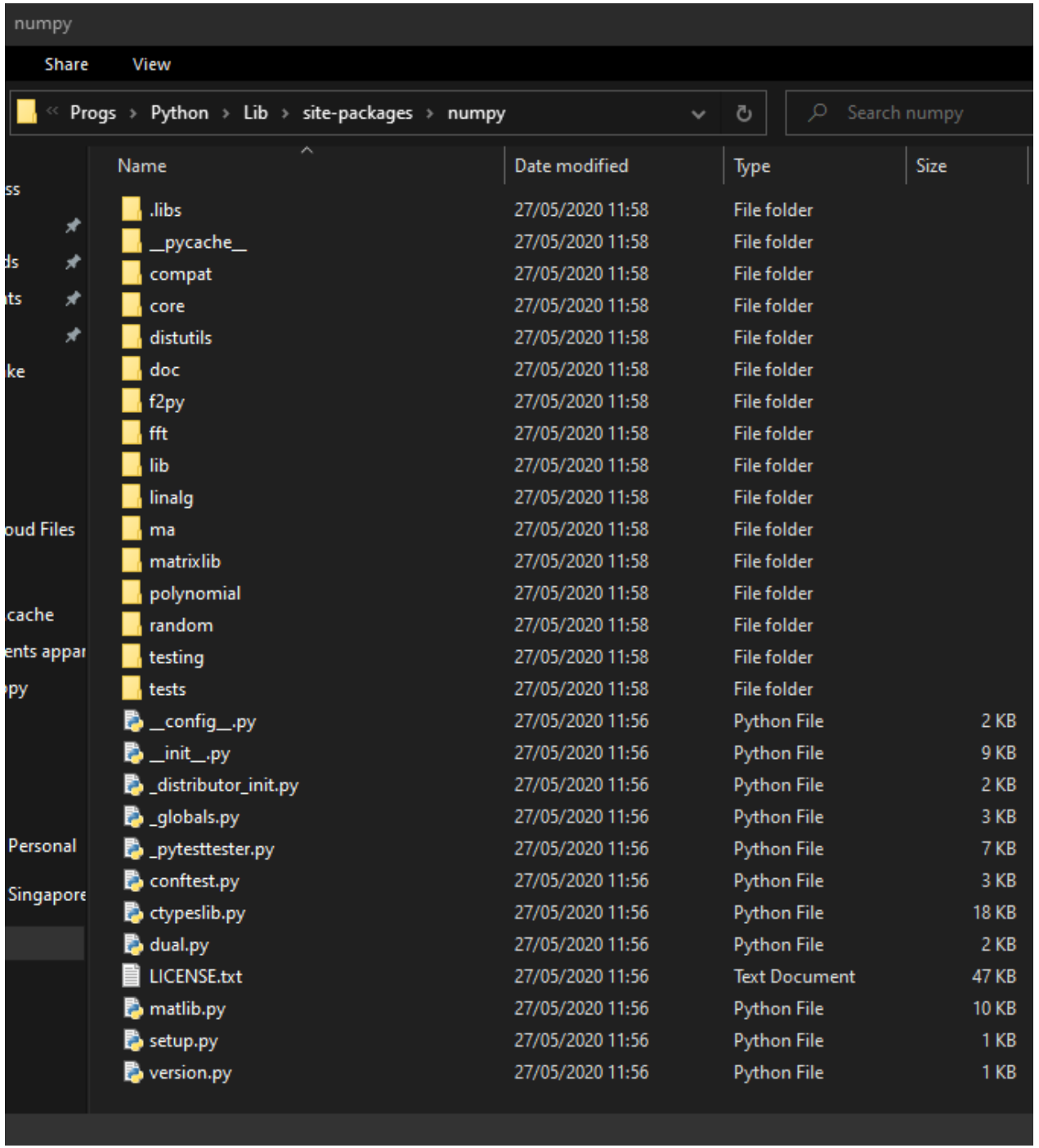
- Python looks for files in your current folder first,
- and then looks in your Python installation directory.

- When you installed the Numpy package with **pip** on Week 1, you downloaded some **numpy files** and stored them in your Python installation folder!

Folders to import from

- Python looks for files in your current folder first,
- and then looks in your Python installation directory.

- When you installed the Numpy package with **pip** on Week 1, you downloaded some **numpy** files and stored them in your Python installation folder!



Project organizing

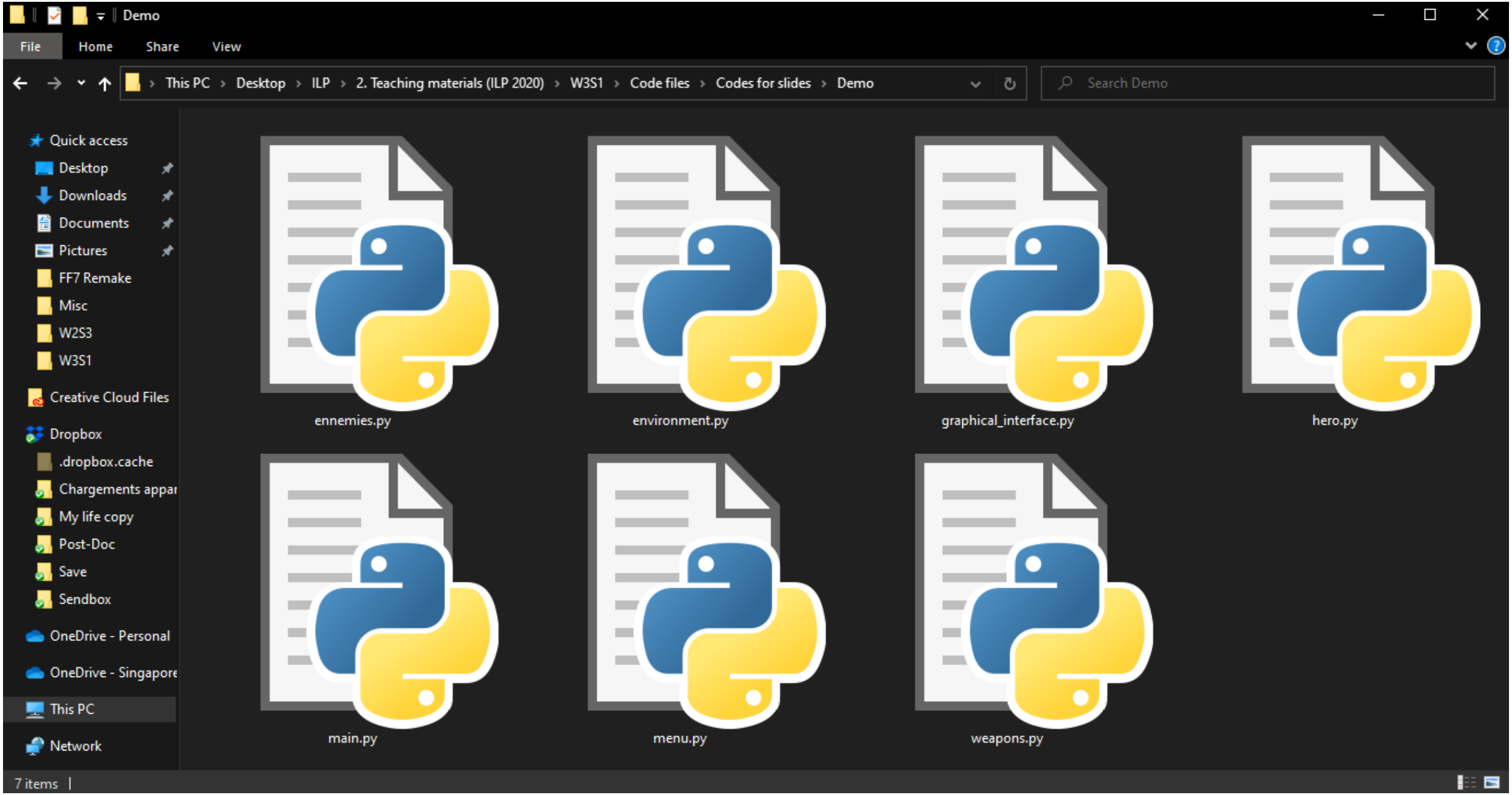
- In many projects, several developers will often collaborate on a single project.
- And each one of them will end up working on a single aspect of the project.

Project organizing

- In many projects, several developers will often collaborate on a single project.
- And each one of them will end up working on a single aspect of the project.

E.g. in a video game:

- One developer can design the main character,
- Another one will develop enemies,
- Another one will develop items and weapons,
- Another one will design the map/environment in which heroes and enemies evolve,
- Etc.


Demo
File
Home
Share
View
This PC
Desktop
ILP
2. Teaching materials (ILP 2020)
W3S1
Code files
Codes for slides
Demo
Search Demo
Quick access
Desktop
Downloads
Documents
Pictures
FF7 Remake
Misc
W2S3
W3S1
Creative Cloud Files
Dropbox
.dropbox.cache
Chargements appar
My life copy
Post-Doc
Save
Sendbox
OneDrive - Personal
OneDrive - Singapore
This PC
Network
ennemies.py
environment.py
graphical_interface.py
hero.py
main.py
menu.py
weapons.py
7 items

Project organizing

- Each developer will then work on his/her own .py file, taking care of his/her specific subtask.
- Later on, other developers might **import** functions from files created by other people.
- Eventually assemble all pieces in a **main.py** file!

Project organizing

- Each developer will then work on his/her own .py file, taking care of his/her specific subtask.
- Later on, other developers might **import** functions from files created by other people.
- Eventually assemble all pieces in a **main.py** file!

- This is something very common in programming projects.
- **Important:** you need to build the habit of documenting your functions!

Matt's Great advice #11

Matt's Great Advice #11: Good import practices

Some good practices in projects

1. As with the variables and functions names, it is a good idea to **make your file names explicit**.
2. Have **a file for each sub-concept**, and **a single main file that assembles them all** at the end.
3. It is often better to **import only what is needed** (**from** … **import** …) rather than importing everything (**from** … **import** ∗, **import** …).

Conclusion

- Memory management and lists: aliasing, shallow and deep copies
- The Numpy library (part 1): arrays, math functions, etc.
- About the import procedure
- Project organizing
- Mini-project