ILP 2024 : Computing

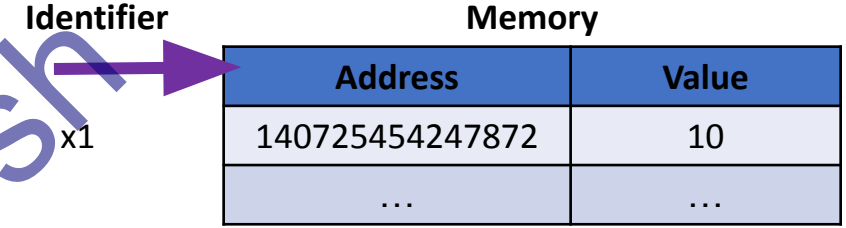
W4S2
05 June 2024

Outline

- 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”.
- 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”.



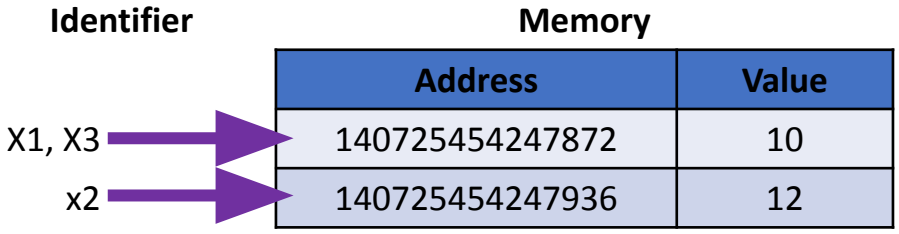

```
1  x1 = 10
2  print(x1)
3  print(id(x1))
```

```
10
140725454247872
```

`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 = 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
```

Memory management in lists

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

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

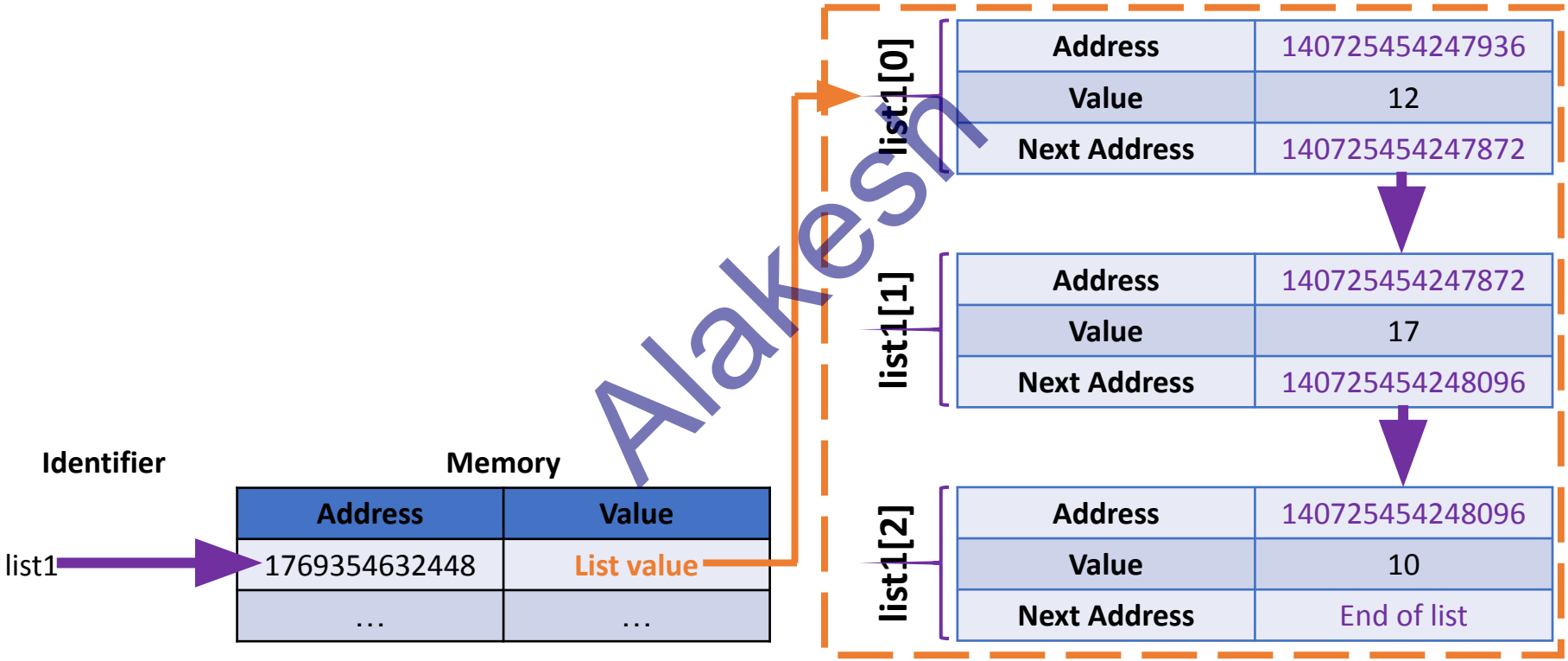
```
[12, 17, 10]
1769354632448
```



```
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



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)
```

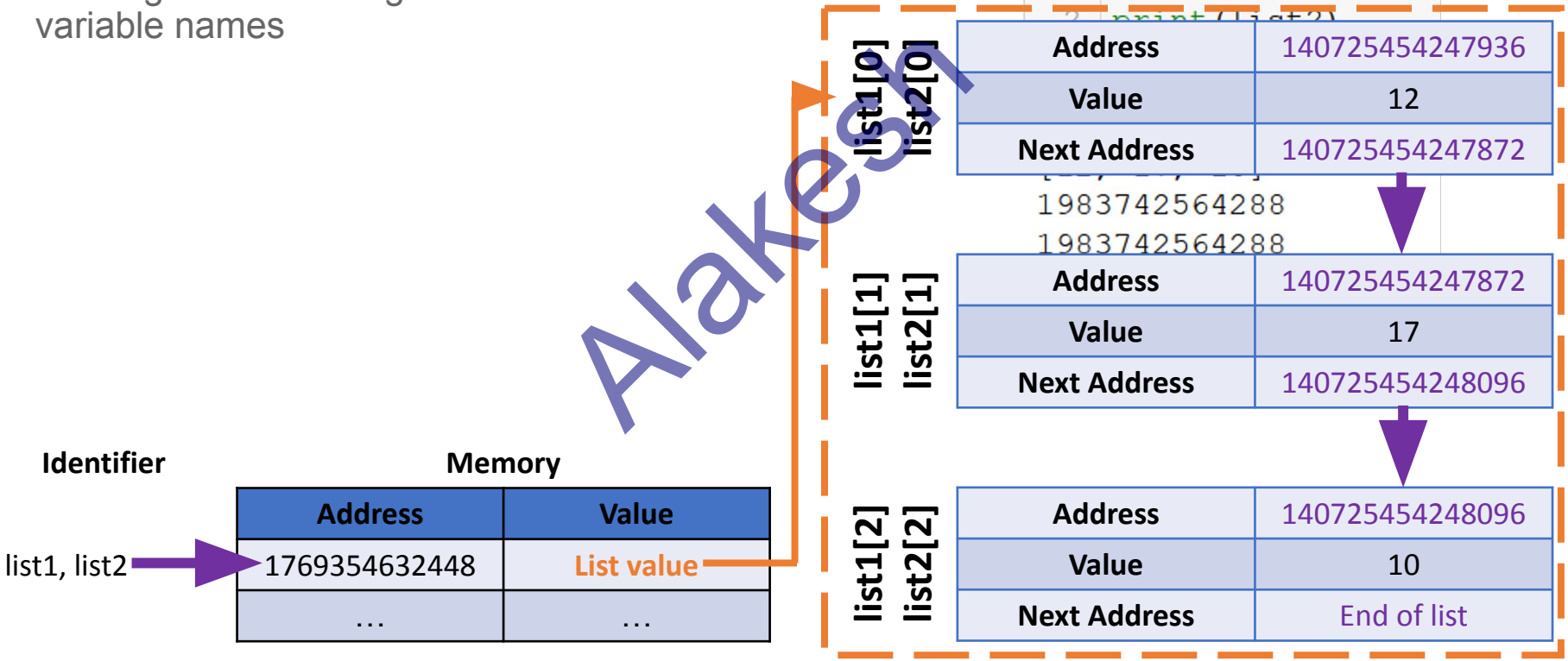
```
12
140726382041088
2120755150000
[12, 17, 10]
```

Memory management in lists

- Aliasing: We can assign a list to another variable names

```
1  list2 = list1
```

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.
- **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]))
```

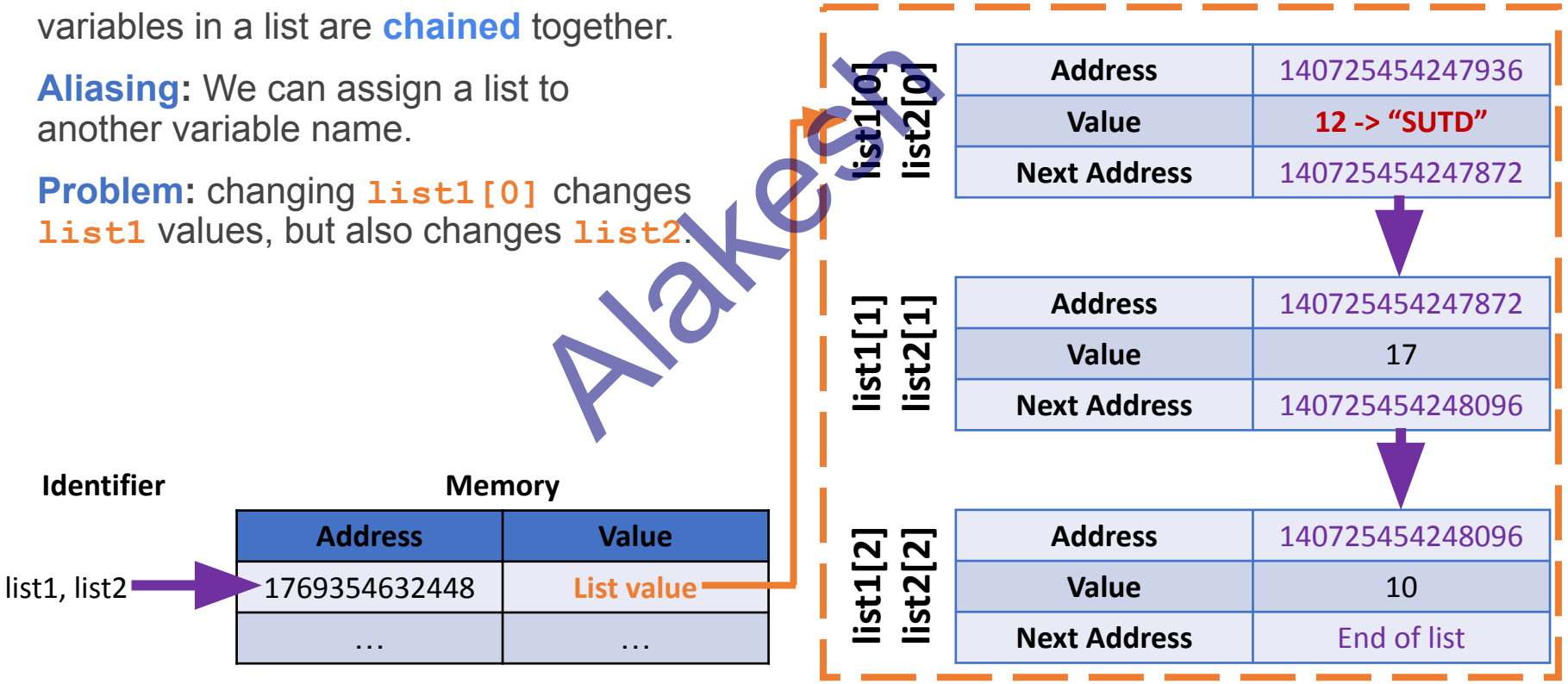
```
140726382041088
['SUTD', 17, 10]
2120755353584
```

```
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, `list1` 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
```

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)
- To install **`NumPy`**
  - **`pip install numpy`**

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 list of list, 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.


```
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.

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

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



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

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

```
# Element-wise
for element in my_list:
    print("--")
    print(element)
```

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

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.


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

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

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

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

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

- **`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]]
```

Shape of an Array

- The shape of an array is the number of elements in each dimension.

- `NumPy` arrays have an attribute called `shape` that returns a tuple with each index having the number of corresponding elements.


```
import numpy as np

arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

print(arr.shape)
```

```
(2, 4)
```

The array has 2 dimensions, where the first dimension has 2 elements and the second has 4.

Shape of an Array

Creating an array with 5 dimensions using `ndmin` using a vector with values 1, 2, 3, 4 and the last dimension has value 4:


```
arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('shape of array :', arr.shape)
```

```
[[[[[1 2 3 4]]]]]
shape of array : (1, 1, 1, 1, 4)
```

Reshaping arrays

- Reshaping means changing the shape of an array.
- The shape of an array is the number of elements in each dimension.
- By reshaping we can add or remove dimensions or change number of elements in each dimension.


```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

```
newarr = arr.reshape(2, 3, 2)

print(newarr)
```

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

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

Converts an 1-D array with 12 elements into a 2-D array, which have 4 arrays, each with 3 elements

Searching arrays

- We can search an array for a certain value, and return the indexes that get a match.
- To search an array, use the `where()` method.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 4, 4])

x = np.where(arr == 4)

print(x)
```

```
(array([3, 5, 6]),)
```



```
arr = np.array([3, 2, 0, 1])

print(np.sort(arr))
```

Can you guess the output?

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.

- **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 python `(.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.

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
```

```
1 from my_code import my_function
```

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

```
5
```

Random Numbers in NumPy

- Random number does NOT mean a different number every time. Random means something that cannot be predicted logically.
- NumPy offers the random module to work with random numbers.


```
from numpy import random

#Generate a random integer from 0 to 100:
x = random.randint(100)
print(x)
print("-")

# Define the range
lower_bound = 1
upper_bound = 100

# Generate 5 random integers within the range
random_integers = np.random.randint(lower_bound, upper_bound + 1, size=5)

print("Random integers:", random_integers)
```

```
2
-
Random integers: [39 30 32 70 30]
```

Random Numbers in NumPy

```
from numpy import random

#Generate a random integer from 0 to 100:
x = random.randint(100)
print(x)
print("-")

# Define the range
lower_bound = 1
upper_bound = 100

# Generate 5 random integers within the range
random_integers = np.random.randint(lower_bound, upper_bound + 1, size=5)

print("Random integers:", random_integers)
```

```
2
-
Random integers: [39 30 32 70 30]
```

- The `choice()` method allows you to generate a random value based on an array of values.

```
x = random.choice([3, 5, 7, 9])
print(x)
```

```
7
```

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

Coding Platforms

Most popular coding platforms

- Leetcode
- HackerRank