FCDS_AI_Assignment_3

June 3, 2022

1 Welcome

Welcome to the first practical work of the week! In this practical, we will learn about the programming language Python as well as NumPy and Matplotlib, two fundamental tools for data science and machine learning in Python.

2 Notebooks

Click on the "play" button to execute the cell. You should be able to see the result. Alternatively, you can also execute the cell by pressing Ctrl + Enter if you are on Windows / Linux or Command + Enter if you are on a Mac.

Variables that you defined in one cell can later be used in other cells:

```
[1]: seconds_in_a_day = 60*60*24 #Seconds per minute * Minutes per hour * hours per␣
     ↪day
     seconds_in_a_day
```

```
[1]: 86400
```

```
[2]: seconds_in_a_week = 7 * seconds_in_a_day
     seconds_in_a_week
```

```
[2]: 604800
```

Note that the order of execution is important. For instance, if we do not run the cell storing *seconds_in_a_day* beforehand, the above cell will raise an error, as it depends on this variable. To make sure that you run all the cells in the correct order, you can also click on "Runtime" in the top-level menu, then "Run all".

Exercise. Add a cell below this cell: click on this cell then click on "+ Code". In the new cell, compute the number of seconds in a year by reusing the variable *seconds_in_a_day*. Run the new cell.

3 Python

[3]:
```
seconds_in_a_year = 365 * seconds_in_a_day
seconds_in_a_year
```

[3]:
```
31536000
```

Python is one of the most popular programming languages for machine learning, both in academia and in industry. As such, it is essential to learn this language for anyone interested in machine learning. In this section, we will review Python basics.

3.1 Arithmetic operations

Python supports the usual arithmetic operators: + (addition), * (multiplication), / (division), ** (power), // (integer division).

3.2 Lists

Lists are a container type for ordered sequences of elements. Lists can be initialized empty

[4]:
```
my_list = []
```

or with some initial elements

[5]:
```
my_list = [1, 2, 3]
```

Lists have a dynamic size and elements can be added (appended) to them

[6]:
```
my_list.append(4)
my_list
```

[6]:
```
[1, 2, 3, 4]
```

We can access individual elements of a list (indexing starts from 0)

[7]:
```
my_list[2]
```

[7]:
```
3
```

We can access "slices" of a list using `my_list[i:j]` where `i` is the start of the slice (again, indexing starts from 0) and `j` the end of the slice. For instance:

[8]:
```
my_list[1:3]
```

[8]:
```
[2, 3]
```

Omitting the second index means that the slice shoud run until the end of the list

[9]:
```
my_list[1:]
```

[9]:
```
[2, 3, 4]
```

We can check if an element is in the list using `in`

[10]:
```
5 in my_list
```

[10]:
```
False
```

The length of a list can be obtained using the `len` function

[11]:
```
len(my_list)
```

[11]:
```
4
```

3.3 Strings

Strings are used to store text. They can delimited using either single quotes or double quotes

[12]:
```
string1 = "some text"
string2 = 'some other text'
```

Strings behave similarly to lists. As such we can access individual elements in exactly the same way

[13]:
```
string1[3]
```

[13]:
```
'e'
```

and similarly for slices

[14]:
```
string1[5:]
```

[14]:
```
'text'
```

String concatenation is performed using the `+` operator

[15]:
```
string1 + " " + string2
```

[15]:
```
'some text some other text'
```

3.4 Conditionals

As their name indicates, conditionals are a way to execute code depending on whether a condition is True or False. As in other languages, Python supports `if` and `else` but `else if` is contracted into `elif`, as the example below demonstrates.

[16]:
```
my_variable = 5
if my_variable < 0:
  print("negative")
```

```
elif my_variable == 0:
  print("null")
else: # my_variable > 0
  print("positive")
```

```
positive
```

Here `<` and `>` are the strict `less` and `greater than` operators, while `==` is the equality operator (not to be confused with `=`, the variable assignment operator). The operators `<=` and `>=` can be used for less (resp. greater) than or equal comparisons.

Contrary to other languages, blocks of code are delimited using indentation. Here, we use 2-space indentation but many programmers also use 4-space indentation. Any one is fine as long as you are consistent throughout your code.

3.5 Loops

Loops are a way to execute a block of code multiple times. There are two main types of loops: while loops and for loops.

While loop

[17]:
```
i = 0
while i < len(my_list):
  print(my_list[i])
  i += 1 # equivalent to i = i + 1
```

```
1
2
3
4
```

For loop

[18]:
```
for i in range(len(my_list)):
  print(my_list[i])
```

```
1
2
3
4
```

If the goal is simply to iterate over a list, we can do so directly as follows

[19]:
```
for element in my_list:
  print(element)
```

```
1
2
3
4
```

3.6 Functions

To improve code readability, it is common to separate the code into different blocks, responsible for performing precise actions: functions. A function takes some inputs and process them to return some outputs.

[20]:
```python
def square(x):
  return x ** 2

def multiply(a, b):
  return a * b

# Functions can be composed.
square(multiply(3, 2)) # f(h((g(x)))
```

[20]:
```
36
```

To improve code readability, it is sometimes useful to explicitly name the arguments

[21]:
```python
square(multiply(a=3, b=2))
```

[21]:
```
36
```

3.7 Exercises ()

Exercise 1. Using a conditional, write the relu function defined as follows

$$\mathrm{relu}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}.$$

[22]:
```python
def relu(x):
  # returns x if it follows the rule
  if(x >= 0):
    return x
  # There is no need for an else statement
  return 0

# Checking for the first condition
print(relu(5))
# checking for the second condition
print(relu(-3))
```

```
5
0
```

Exercise 2. Using a for loop, write a function that computes the Euclidean norm of a vector, represented as a list.

[23]:
```python
def euclidean_norm(vector):
  # Declare a variable that will hold the result
  result = 0.0
  # Iterate over the vector elements
  for elem in vector:
    # Square each element and add it to the result
    result += elem ** 2
  # Get the square root of the result
  result = np.sqrt(result)
  return result

import numpy as np
my_vector = [0.5, -1.2, 3.3, 4.5]
three_vector = [2,6,-12]
# The result should be roughly 5.729746940310715
print(my_vector)
print("Its Euclidean Norm", euclidean_norm(my_vector))
print(three_vector)
print(euclidean_norm(three_vector))
```

```
[0.5, -1.2, 3.3, 4.5]
Its Euclidean Norm 5.729746940310715
[2, 6, -12]
13.564659966250536
```

Exercise 3. Using a for loop and a conditional, write a function that returns the maximum value in a vector.

[24]:
```python
def vector_maximum(vector):
  # Initlize the maximum value to be the first element
  max = vector[0]
  # Iterate over the rest of the vector
  for elem in vector[1:]:
    # Check if the element is the maximum so far
    if(elem > max):
      max = elem
  return max

my_vector = [0.5, -1.2, 3.3, 4.5]
print(vector_maximum(my_vector))
my_vector = [1,2,3,4,5,6]
print(vector_maximum(my_vector))
```

```
4.5
6
```

Exercise 4. Write a function that sorts a list in ascending order (from smaller to bigger) using the bubble sort algorithm.

[25]:
```python
def bubble_sort(my_list):
  swapped = False
  # Get the length of the list
  n = len(my_list)
  # Iterate over it
  for i in range(n - 1):
    # (n - i - 1) -> Don't look at the last index as it has the correct value
    for j in range(0,n - i - 1):
      if(my_list[j] > my_list[j+1]):
        my_list[j],my_list[j+1] = my_list[j+1],my_list[j]
        swapped = True
    if(not swapped):
        print("already sorted")
        break
  return my_list
my_list = [1, -3, 3, 2]
my_list_sorted = [1,2,3,3]
# Should return [-3, 1, 2, 3]
print(bubble_sort(my_list))
print(bubble_sort(my_list_sorted))
```

```
[-3, 1, 2, 3]
already sorted
[1, 2, 3, 3]
```