intermediate_workshop_1

June 29, 2019

1 Intermediate Workshop 1: numpy, loops and *if statements*

1.1 1.1 Numpy

Numpy's main object is the multidimensional array, known as an ndarray. It can also be referred as an array.
Indeed, numpy.array is just a convenience function to create an ndarray.
So, an ndarray should always be constructed using *array*, *zeros*, or *empty* as per numpy's documentation.

```
In [3]: import numpy as np

        # instantiating an ndarray using array
        arr = np.array([[1,2,3],[10,11,12]])
        arr, type(arr)

Out[3]: (array([[ 1,  2,  3],
                [10, 11, 12]]), numpy.ndarray)

In [4]: # instantiating an ndarray using zeros
        zeros_arr = np.zeros((3,2)) # the input is a tuple to determine the desired shape
        zeros_arr

Out[4]: array([[0., 0.],
               [0., 0.],
               [0., 0.]])

In [5]: # instantiating an ndarray using empty
        empty_arr = np.empty((3,2))
        empty_arr

Out[5]: array([[0., 0.],
               [0., 0.],
               [0., 0.]])
```

The main attributes of an ndarray are
* ndim: the number of axes (dimensions) of the array * shape: the dimension of the array (for an n*m array, the dimension is the tuple (n,m))* size: number of elements in the array (n times m) * dtype: all elements in an numpy have the same type (dtype)

```
In [6]: {'ndim': arr.ndim, 'shape': arr.shape, 'size': arr.shape, 'dtype': arr.dtype}
```

```
Out[6]: {'ndim': 2, 'shape': (2, 3), 'size': (2, 3), 'dtype': dtype('int32')}
```

We can use arithmetic operations on arrays, these operations are applied *elementwise*.

```
In [7]: arr2 = np.arange(6).reshape(2,3) # the arange function just creates a array with
          # elements from 0 to size
        print(arr)
        print(arr2)
        arr + arr2
```

```
[[ 1  2  3]
 [10 11 12]]
[[0 1 2]
 [3 4 5]]
```

```
Out[7]: array([[ 1,  3,  5],
               [13, 15, 17]])
```

```
In [8]: arr * arr2
```

```
Out[8]: array([[ 0,  2,  6],
               [30, 44, 60]])
```

There exists a bunch of unary function that works on arrays. For instance, the square root function, the exponential function and so on.

```
In [9]: print(np.sqrt(arr))
        np.sqrt(arr)
```

```
[[1.         1.41421356 1.73205081]
 [3.16227766 3.31662479 3.46410162]]
```

```
Out[9]: array([[1.        , 1.41421356, 1.73205081],
               [3.16227766, 3.31662479, 3.46410162]])
```

In the previous workshop, we saw how indexing and slicing can be used on lists and strings. Now, we can use these same concepts on arrays.

```
In [10]: print(arr)
         print(arr[1:, 1:])
         print(arr[0, 0])
         print(arr[-1, -1])
```

```
[[ 1  2  3]
 [10 11 12]]
[[11 12]]
1
12
```

For arrays, there is a new character ... that can be particularly useful when indexing arrays with lots of dimensions.
$x[1,2...] = x[1,2,:,:,:]$

```
In [11]: arr[..., 0] # select only the first "column"
```

```
Out[11]: array([ 1, 10])
```

We can stack arrays on top of eachother to extend our defined arrays. We can use the *vstack* to stack data vertically or *htstack* to stack data horizontally.

```
In [12]: new_row = np.array([1000, 2000, 3000])
         np.vstack((arr, new_row)) # inputs have the shape of a tuple
```

```
Out[12]: array([[   1,    2,    3],
                [  10,   11,   12],
                [1000, 2000, 3000]])
```

```
In [13]: new_column = np.array([999, 888])
         new_column = new_column.reshape(2,1)
         print(new_column)
         np.hstack((new_column, arr))
```

```
[[999]
 [888]]
```

```
Out[13]: array([[999,   1,   2,   3],
                [888,  10,  11,  12]])
```

Something you must always keep in mind in numpy, and often in Python is that you access variable by reference, and not by value. In other words, when I re-assign an array, it is the same one re-assigned twice.

```
In [14]: new_arr = arr
         print(arr is new_arr)
         arr[0,0] = 999999
         new_arr
```

```
True
```

```
Out[14]: array([[999999,      2,      3],
                [    10,     11,     12]])
```

If I want to actually copy an array to modify one variable independently of the other, you can use the *copy* function.

```
In [15]: new_arr = arr.copy()
         print(arr is new_arr)
         arr[0,0] = 22222
         new_arr
```

```
False
```

```
Out[15]: array([[999999,      2,      3],
                [    10,     11,     12]])
```

Let's create a bigger array with the *arange* function.

```
In [16]: big_arr = np.arange(100).reshape(10, 10)
         big_arr
```

```
Out[16]: array([[ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9],
                [10, 11, 12, 13, 14, 15, 16, 17, 18, 19],
                [20, 21, 22, 23, 24, 25, 26, 27, 28, 29],
                [30, 31, 32, 33, 34, 35, 36, 37, 38, 39],
                [40, 41, 42, 43, 44, 45, 46, 47, 48, 49],
                [50, 51, 52, 53, 54, 55, 56, 57, 58, 59],
                [60, 61, 62, 63, 64, 65, 66, 67, 68, 69],
                [70, 71, 72, 73, 74, 75, 76, 77, 78, 79],
                [80, 81, 82, 83, 84, 85, 86, 87, 88, 89],
                [90, 91, 92, 93, 94, 95, 96, 97, 98, 99]])
```

1.1.1 Exercise 1

Retrieve the last row of the *big_arr*.

```
In [17]: def exercise1(arr):
             x = None
             return x

         assert (exercise1(big_arr) == np.array([90, 91, 92, 93, 94, 95, 96, 97, 98, 99])).all
```

```
        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-17-52d47cc00606> in <module>
          3   return x
          4
    ----> 5 assert (exercise1(big_arr) == np.array([90, 91, 92, 93, 94, 95, 96, 97, 98, 99])).a


        AssertionError:
```

1.2 1.2 *If statements*

As the goal of this workshop is to make you very comfortable with looping in Python, the second building block necessary for this goal is to understand control flow tools and control statements. We will start with the *if statement*.

In [18]:
```
condition = True # boolean variable set to true

if condition == True:
  print('this statement runs')

condition = False

if condition == True:
  print('this statement does not')
```

```
this statement runs
```

In the previous cell, a lot of new keywords and concepts were introduced, let's describe them

```
if condition == True
```

The $==$ is a comparison operator which compares the value to its right to the value to its left. If the two elements are equal, then the whole expression evaluates to true and can be understood as

```
if True == True
if True: # True == True reduces to True
```

Now, the *if statement*, being *True*, executes the piece of code after the colon. In our case,

```
print('this statement runs')
```

executes.
In the second case,

```
if condition == True
```

evaluates to

```
if False == True
```

and since $False \neq True$, then the expression evaluates to *False* just like

```
if False
```

and so the block of code delimated by this *if statement* does not execute.
To summarize, *if statements* only execute their code if their condition is *true*.

```
if condition:
  their code
```

Where their code is indented more than the if line.

Beside the equality comparison operator (==) seen in the previous cell, there exists a lot more logic operators in python. The opposite of the equality comparison operator is the *not equal* operator (!=). There exists also the *greater* and *greater or equal* operators (>=, >), the *less than* and *less or equal* operators (<=, <), and the negation operator (not).

In [19]:
```
condition = True
small_number = 5
huge_number = 100000

if not condition:
  print('will not print')

if condition != False:
  print('will print')

if small_number > huge_number:
  print('will not print')

if small_number < huge_number:
  print('will print')

if huge_number >= small_number:
  print('will print')
```

```
will print
will print
will print
```

The equal sign with a boolean value is optional

```
if condition == True:
  print('hi')
```

is exactly the same statement as

```
if condition:
  print('hi')
```

and

```
if condition == False:
  print('hi')
```

is exactly the same statement as

```
if not condition:
  print('hi')
```

```
In [20]: condition = False
         if not condition:
           print('not False == True, so I print')

         if condition:
           print('will not print')
```

```
not False == True, so I print
```

We can mix conditions together for more flexibility in logical expressions such as *if statements*. We have the *and* and the *or* statements

Condition 1	Condition 2	or	and
False	False	False	False
False	True	True	False
True	False	True	False
True	True	True	True

We can interpret this table with the following code

```
if False and True:
  print('will not print')

if False or True:
  print('will print')
```

```
In [21]: if False or False:
           print('this does not print')

         if True and True:
           print('this will print')
```

```
this will print
```

The examples seen in the previous cells all used boolean values or numbers . However, we can create more complex conditions involving strings and functions.

```
In [22]: name = 'Jack O'

         if name == 'Jack O':
           print('this will print')
```

```
        if len(name) == 6:
          print('this will print')

        if name == 'Jack O' and len(name) == 6:
          print('this will print')
```

```
this will print
this will print
this will print
```

1.2.1 Exercise 2

Make sure that either number1 or number2 lie between their respective thresholds

```
In [23]: def exercise2(number1, low_threshold1, high_threshold1,
                       number2, low_threshold2, high_threshold2):
           pass

         assert exercise2(10, 15, 20, 12, 10, 11) == False

         assert exercise2(8, 6, 9, 10, 11, 22) == True
```

```
        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-23-655aa52a7410> in <module>
          3   pass
          4
    ----> 5 assert exercise2(10, 15, 20, 12, 10, 11) == False
          6
          7 assert exercise2(8, 6, 9, 10, 11, 22) == True


        AssertionError:
```

1.3 1.3 Loops

Loops are very important in any good programming languages, and Python is no exception. They are often used to iterate over some lists, arrays or iterable data structure, or to repeat a task multiple times. Python features two main types of loops: the *for loop* and the *while loop*.

```
In [24]: for element in big_arr:
           print("current element: {}".format(element))
```

```
current element: [0 1 2 3 4 5 6 7 8 9]
current element: [10 11 12 13 14 15 16 17 18 19]
current element: [20 21 22 23 24 25 26 27 28 29]
current element: [30 31 32 33 34 35 36 37 38 39]
current element: [40 41 42 43 44 45 46 47 48 49]
current element: [50 51 52 53 54 55 56 57 58 59]
current element: [60 61 62 63 64 65 66 67 68 69]
current element: [70 71 72 73 74 75 76 77 78 79]
current element: [80 81 82 83 84 85 86 87 88 89]
current element: [90 91 92 93 94 95 96 97 98 99]
```

Let's understand the previous statement.

```
for element in big_arr:
```

since *big_arr* is a multidimensional array (a 2 dimension array in our case), we iterate over the rows. Everything is taken care of under the hood such that the loop stops automatically. We can appreciate the simplicity of Python when we compare to other programming languages, such as C.

```
int[][] big_arr = new int[10][10];
for (int i = 0; i < 10; i++){
  printf("current element:");
  for (int j = 0; j < 10; j++)}
    printf("%f", big_arr[i][j]);
  }
  printf("\n");
}
```

We might want the index for numerous purposes such as printing the current iteration. We use the *enumerate* method to do so.

```
In [25]: for index, element in enumerate(big_arr):
             print('current index: {}, current element: {}'.format(index, element))
```

```
current index: 0, current element: [0 1 2 3 4 5 6 7 8 9]
current index: 1, current element: [10 11 12 13 14 15 16 17 18 19]
current index: 2, current element: [20 21 22 23 24 25 26 27 28 29]
current index: 3, current element: [30 31 32 33 34 35 36 37 38 39]
current index: 4, current element: [40 41 42 43 44 45 46 47 48 49]
current index: 5, current element: [50 51 52 53 54 55 56 57 58 59]
current index: 6, current element: [60 61 62 63 64 65 66 67 68 69]
current index: 7, current element: [70 71 72 73 74 75 76 77 78 79]
current index: 8, current element: [80 81 82 83 84 85 86 87 88 89]
current index: 9, current element: [90 91 92 93 94 95 96 97 98 99]
```

Notice that the loop starts at index 0 and not 1.
We can pile up loops on top of each other. The first loop is called the *outer* loop while the second loop is calle dthe *inner* loop.

```
In [26]: for i, row in enumerate(big_arr): # outer loop
           for j, element in enumerate(row): # inner loop
             if i * big_arr.shape[0] + j < 20: # I print only the first 20 elements
               print(element)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
```

1.3.1 Exercise 3

Compute the average value of the *x* array.

```
In [27]: x = np.random.randn(100)

         def exercise3(arr):
           _sum = 0
           return _sum


         assert '{:3f}'.format(exercise3(x)) == '{:3f}'.format(np.average(x))
```

```
        ---------------------------------------------------------------------------

        AssertionError                            Traceback (most recent call last)

        <ipython-input-27-daaba776d938> in <module>
          6
          7
    ----> 8 assert '{:3f}'.format(exercise3(x)) == '{:3f}'.format(np.average(x))
```

```
AssertionError:
```

There exists another way of dealing with *for loops* in Python which is named list comprehensions. The basic syntax is

```
[ expression for item in list if condition ]
```

notice the two bracked surrounding the line, as well as the few keywords.
* for: the variable name following the for is an individual element coming from the list which can be accessed in the expression. * in: the variable following the in keyword represents where we are looping on. * if: the if keyword as well as its condition that follows it are optional.

```
In [37]: random_values = np.random.randn(12)
         print(np.average(random_values))
         top_x_values = [el for el in random_values if el > np.average(random_values)]
         top_x_values
```

```
-0.019922292112282376
```

```
Out[37]: [0.6414412389252723,
          0.2099006295640668,
          0.300886351097635,
          0.05492078564340846,
          0.8319457661383078,
          1.008280837606465]
```

```
In [40]: # creating a simple 2d list
         twodim_list = [[] for whatever in range(5)]
         twodim_list
```

```
Out[40]: [[], [], [], [], []]
```

There is another type of loop which is the *while* loop that has a different structure. The *while* loop uses the same structure as the *if statement* seen previously. It executes until its condition evaluates to *False*.

```
while condition:
  expression
```

The expression runs only if the condition evaluates to *True*.

```
In [41]: i = 0
         while i < 5:
           print('i value: {}'.format(i))
           i = i + 1 # alternatively i += 1
```