Python cheat sheet

May 11, 2019

1 This is a Jupyter notebook.

Jupyter notebooks can be used to interactively run Python commands. It becomes very handy if you want to track or structure your pipelines.

Install it using:
`pip install jupyter`
The language for this kind of comments is markdown.
If you want to run jupyter, type:
`jupyter notebook`
into your shell terminal.

2 Python essentials

All details about using the Python language can be found here:
https://docs.python.org/3/contents.html

Please note that this cheat sheet uses Python 3 notation. There are some subtle differences to Python 2, but since Python 2 will not be supported anymore from 2020 on, switching to Python 3 might be a good idea anyway.

2.1 Mathematical operations

Addition / Subtraction / Multiplication / Division / Power / square root / modulo

Below you can find the mathematical operators commonly used in Python. The four "common" operators are similar to what are you used to from MatLab. Only raising a value to a power uses ** instead of ^ In native Python you further don't have a square root operator. Instead raise the value to the power of 1/2. The modulo operator (%) returns the remaining value similar to MatLab's 'mod' function.

It is also possible to use the "math" package (`import math`) to e.g. use the math.sqrt() function.

[1]:
```
40 + 2
```

[1]:
```
42
```

[2]:
```
142 - 100
```

[2]:
```
42
```

[3]:
```
6 * 7
```

[3]:
```
42
```

[4]:
```
84 / 2
```

[4]:
```
42.0
```

[5]:
```
42**2
```

[5]:
```
1764
```

[6]:
```
1764**.5 # yes, you don't have to write the zero
```

[6]:
```
42.0
```

[7]:
```
26 % 7
```

[7]:
```
5
```

2.2 outputting results

Printing an output to the command line is done by using `print()`, which is similar to MatLab's `disp()` function. Note, that Python does not use an end-of-line statement, such as `;` in MatLab and thus natively does not print any output to the command line, if not indicated by the `print` statement.

[8]:
```
print("things I want to display")
```

```
things I want to display
```

2.3 assigning and using simple variables

[9]:
```
a = 5
print(a)
```

```
5
```

[10]:
```
a = a + 1
print(a)
```

```
6
```

2.3.1 really handy

In Python you can use the < operator > = < value > notation to calculate and re-assign values. This becomes handy when e.g. using counters in loops etc.

[11]:
```
a = 41
a += 1 # add and (re-) assign variable in one go

print(a)

a = 1764
```

```
a **=.5

print(a)
```

```
42
42.0
```

2.4 variable types

You will come across many functions in Python that either require `int` (integer) or `float` (float) operators. Integers are much more efficient in terms of memory, but of course restricted to whole numbers. In general it's a good idea to use `int` whenever possible. Further there exist `string` (character strings), `bool` (boolean values, either `True` or `False`) and the special type `None`.

[12]:
```
print(42) # integer
print(42.) # float - again you don't have to write the zero

my_integer = int(42)
my_float = float(42)

print(my_integer, my_float)
```

```
42
42.0
42 42.0
```

[13]:
```
my_string = 'my string'

print(my_string)
print(str(42))

my_boolean = True
my_NoneType = None

print(my_boolean, my_NoneType)
```

```
my string
42
True None
```

2.5 concatenate output

[14]:
```
my_integer = 42
print("I can combine this string with the number " + str(my_integer) +
      " by converting the number into a string first.")
```

```
I can combine this string with the number 42 by converting the number into a
string first.
```

2.6 representing data

Note that Python (similar to far most of all programming languages) uses indexing starting from 0. Hence the first value in every assembly of values can be accessed with the value 0

2.6.1 tuple

Tuple can hold different data types and cannot be changed once assigned. However this behavior makes them very fast as compared to lists (below). Creating a tuple is indicated using normal parentheses `tuple = (val1, val2, etc)`. You can access values within a tuple using square brackets [index]

[15]:
```
a = (4, 2)
print(a)

print(a[0])
```

```
(4, 2)
4
```

Since tuple can hold multiple datatypes you can store e.g. an integer and a string within the same tuple.

[16]:
```
year_name = (1990, "Tommy")
print(year_name)
```

```
(1990, 'Tommy')
```

Since tuples can not be changed after initialization, a re-assignment of values within a tuple is not possible.

[17]:
```
year_name[0] + 1 # this works
```

[17]:
```
1991
```

[18]:
```
year_name[0] = 1990 # this doesn't
```

```
␣
↪---------------------------------------------------------------------------

      TypeError                                 Traceback (most recent call␣
↪last)

      <ipython-input-18-04b95c6e3f1d> in <module>
    ----> 1 year_name[0] = 1990 # this doesn't


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

2.6.2 lists

Lists in turn can be changed but are slower. To create a list indicate it's initialization using square brackets `list = [val1, val2, etc]` Indexing works the same as for tuples using square brackets as well.

[19]:
```
a= [4, 2]
print(a)

print(a[0])
```

```
[4, 2]
4
```

[20]:
```
year_name = [1990, "Tommy"]
print(year_name)
```

```
[1990, 'Tommy']
```

[21]:
```
year_name[0] + 1
```

[21]:
```
1991
```

As already mentioned, list values can be re-assigned

[22]:
```
year_name[0] = 1990
```

[23]:
```
year_name.append(1993)
year_name.append("Francy")
print(year_name)
```

```
[1990, 'Tommy', 1993, 'Francy']
```

[24]:
```
year_name = [1990, "Tommy"]
year_name.extend([1993, "Francy"])
print(a)
```

```
[4, 2]
```

2.6.3 No matrices in Python

You might wonder how Python handles matrices. In fact there is no such a thing like a matrix that can be handled by a computer. Since computers always handle input serially, a matrix is nothing more than a rearranged list. In fact MatLab displays matrices, but internally uses the same logic as Python does, namely using a list of lists.

[25]:
```
column1 = [1, 2, 3]
column2 = [4, 5, 6]
column3 = [7, 8, 9]
```

```
my_matrix = [column1, column2, column3]

print(my_matrix)

print(my_matrix[0])
print(my_matrix[0][1])

# print(my_matrix[:,1]) # this does not work
```

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
[1, 2, 3]
2
```

Since scientists often use matrices to represent data, there is a workaround to handled this. A package called `numpy` wraps this behavior efficiently. That's why numpy became one of the most commonly used packages. In numpy a matrix is represented as `array` Since numpy is not part of the core libraries in Python it has to be installed before being able to import it. Importing packages will be discussed later, but for the sake of demonstrating arrays here is a short spoiler.

[26]:
```
import numpy as np # convention that is used by almost everyone

my_array = np.array(my_matrix) # convert list of lists from before into a numpy␣
 ↪array

print(my_array)

print(my_array[0])
print(my_array[0][1])
```

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

[27]:
```
print(my_array[:,1]) # now matrices can be sliced in the same way as in MatLab
```

```
[2 5 8]
```

[28]:
```
print(my_array[range(0,2),:]) # or even
```

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

2.6.4 dictionaries

Those are objects that can hold Key - Value pairs. Which means once created a string is chosen as a Key and paired with a Value. Dictionaries can be created using curly brackets { } Key and Value have to be separated by a colon.

[29]:
```
my_dict = {"Key":"Value", "Tommy":1990, "Francy":1993}
```

To access a certain value use square brackets and the respective Key as an "index", which will return the corresponding value. Further it is possible to return all keys or key-value pairs, which can become handy when iterating over a dictionary. In general dictionaries (unsorted) are faster than lists (sorted), because items are accessed via a key and not via the respective item position.

[30]:
```
# return values
print(my_dict["Key"])
print(my_dict["Francy"])

# return all possible keys
print(my_dict.keys())

# return all Key-Value pairs
print(my_dict.items())
```

```
Value
1993
dict_keys(['Key', 'Tommy', 'Francy'])
dict_items([('Key', 'Value'), ('Tommy', 1990), ('Francy', 1993)])
```

2.7 basic statements

Any kind of higher order statement such as `if` or `for` need a `:` and a tabular indent like so:

2.7.1 for loop

[31]:
```
for digit in (4, 2): # finishing line by colon
    print(digit) # indenting to indicate enclosing functionality

print(digit) # if no indent was used
```

```
4
2
2
```

2.7.2 if statement

[32]:
```
list1 = []
list2 = []
list3 = list1

if list1 == list2: # list 1 equals (is contains the same as) list 2
    print("True")
else:
    print("False")
```

```
if list1 is list2: # list 1 IS NOT the very same object as list 2 at the same␣
 ↪memory location
    print("True")
else:
    print("False")

if list1 is list3: # list 1 IS the very same object as list 3 at the same␣
 ↪memory location
    print("True")
else:
    print("False")

if list1 is list2:
    print("list 1 is list 2")
elif list2 is list3: # else if statement
    print("list 2 is list 3")
else:
    print("list 1 is list 3")
```

```
True
False
True
list 1 is list 3
```

2.7.3 while statement

[33]:
```
counter = 10
while counter > 0:
    counter -= 1
    print(counter)
```

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

2.7.4 continue, pass, break

[34]:
```
for number in range(0, 10):

    if number == 3: # skips this loop iteration
        continue

    elif number == 5: # just goes on
        pass

    elif number == 8: # exits loop
        break

    print(number)
```

```
0
1
2
4
5
6
7
```

2.7.5 with statement (reading files)

This statement is easy to understand, when considering opening a file. As you will know from MatLab, opening a file involves assigning a fileID, opening the file, doing something and afterwards closing it. Now the closing part is necessary to no longer hold the respective file in memory. However if something happens and an error occurs, the file might not be closed properly.

The with statement solves this problem by internally wrapping a `try` and `finally` statement. That is 'trying' something and even if something happens executing a 'final' statement (e.g. closing the file).

So `with` is thus mostly used for opening files.

[35]:
```
# crashes due to missing file
with open("my_file.ext", "r") as my_data: # opens file for reading ("r") as the␣
 ↪variable "my_data"
    data = my_data.read() # reads the actual file data into a new variable
```

```
      ␣
↪---------------------------------------------------------------------------

       FileNotFoundError                         Traceback (most recent call␣
↪last)

       <ipython-input-35-1492469eff26> in <module>
         1 # crashes due to missing file
```

```
   ----> 2 with open("my_file.ext", "r") as my_data: # opens file for reading␣
↪("r") as the variable "my_data"
         3     data = my_data.read() # reads the actual file data into a new␣
↪variable


FileNotFoundError: [Errno 2] No such file or directory: 'my_file.ext'
```

2.8 really handy

In general it's good to know that in Python you can loop over multiple variables at the same time. Below you'll find some examples why this can be useful.

2.8.1 enumerate

This function returns the respective value of an assembly of data and it's index. Thus using enumerate you could e.g. get rid of counter variables.

[36]:
```
my_list = range(30, 40)
for entry, number in enumerate(my_list):
    print(str(number) + " is the " + str(entry) + " entry in the list")
```

```
30 is the 0 entry in the list
31 is the 1 entry in the list
32 is the 2 entry in the list
33 is the 3 entry in the list
34 is the 4 entry in the list
35 is the 5 entry in the list
36 is the 6 entry in the list
37 is the 7 entry in the list
38 is the 8 entry in the list
39 is the 9 entry in the list
```

2.8.2 zip

This function creates an iterable object, that makes multiple data assemblies accessible. E.g. if you have two (or more) vectors of data, the MatLab way of doing it would be to iterate through the number of indices and then accessing each vector by the respective index. Using `zip` you can "concatenate" those and access the same "row" of each vector at the same time.

[37]:
```
numbers = range(30, 40, 2)
entries = range(0, len(numbers))

for entry, number in zip(entries, numbers):
    print(str(number) + " is the " + str(entry) + " entry in the list")
```

```
30 is the 0 entry in the list
32 is the 1 entry in the list
```

```
34 is the 2 entry in the list
36 is the 3 entry in the list
38 is the 4 entry in the list
```

2.8.3 one line loop

While in MatLab data has to be collected in a separate variable, Python offers to collect the output of a for-loop directly as a list. In the example below each item within the range shall be squared and put in a new variable.

[38]:
```
numbers = range(0, 10)

squared_numbers = [number**2 for number in numbers]
print(squared_numbers)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

2.8.4 one line if

The one-line for-loop functionality can further be extended by including an if statement. Here only even numbers are squared, by adding an if statement, checking whether the modulo value equals zero.

[39]:
```
numbers = range(0, 10)

squared_even_numbers = [number**2 for number in numbers if number % 2 == 0]
print(squared_even_numbers)

print(["problem solved" if number == 42 else "problem still remains" for number␣
 ↪in (42, 13)])
```

```
[0, 4, 16, 36, 64]
['problem solved', 'problem still remains']
```

3 Packages

One of the ore features of Python is the use of packages or `modules`. Those are collections of functions that can be accessed after importing them. Importing in this context means telling Python to make use of those pre-installed functions and putting them into the respective name space.

3.1 installing packages

The easiest way to install packages is to use `pip`. Which works like so

`pip install <package_name>`

e.g.

`pip install numpy`

Sometimes a package is not contained within the package-management and must be installed manually. In that case it's important to link the respective package, so that you Python interpreter can find it.

3.2 Importing packages

While it is possible to import all functions of a package like so `from my_package import *` which would give you direct access to all the functions, it is not advised to do so. It is much better practice to import packages and refer to functions from this package from the packages name space.

E.g. it would be possible to `from math import *` and then use `sqrt(number)` it would be much better practice to `import math` and use `math.sqrt(number)` or even `from math import sqrt` and use `sqrt(number)`.

When using the asterisk all functions will be imported which can mess up you name space. Additionally you can keep much better track about what function you use when explicitly importing them.

Another handy feature is that the name space of a given set of functions can be modified. E.g. it is common practice to rename `numpy` to `np` (like below). This will ensure that all package specific functions are still in the name space of the package, but a shorter name can be chosen for convenance.

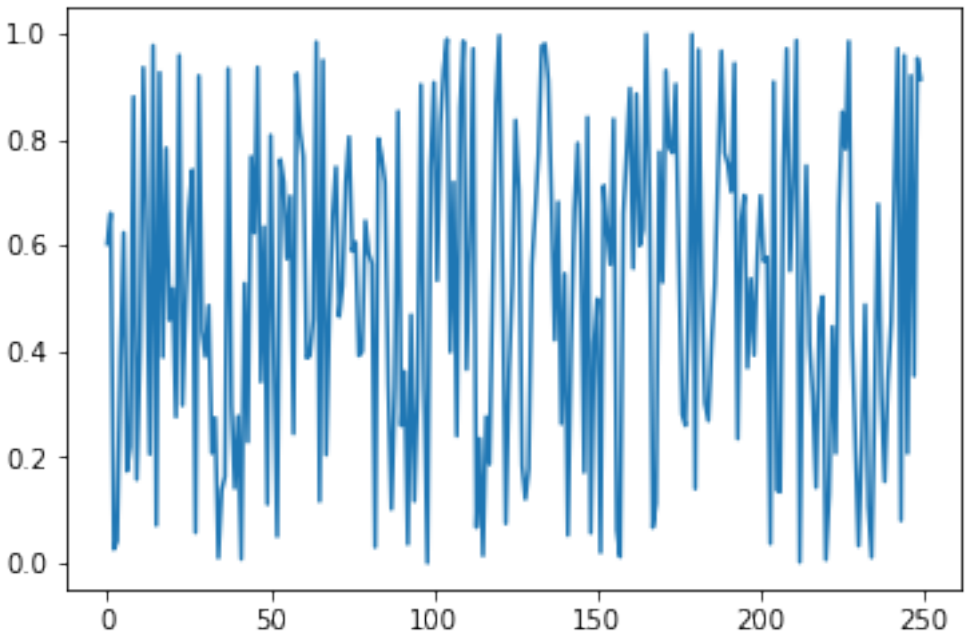
Every sub-function of one module can be used using the `.` operator. Large packages often use also sub-modules, so multiple `.` operators can be required. However it is also possible to import a sub-module only.

[40]:
```
import numpy as np # change namespace
from matplotlib import pyplot as plt # import only sub-module
```

[41]:
```
random_data = np.random.rand(250, 32) # access function of sub-module
```

[42]:
```
channel = 0

plt.figure
plt.plot(random_data[:,channel])
plt.show()
```



4 Functions

Functions are a set of executable operations, that can (but don't have to) take input variables and can (but don't have to) return one or multiple values.

[43]:
```
# function without return statement
def sub_numbers(val_one, val_two):
    sum_of_values = val_one + val_two

my_sum = sub_numbers(22, 20)
print(my_sum)

# function with return statement
def add_numbers(val_one, val_two):
    sum_of_values = val_one + val_two
    return sum_of_values

my_sum = add_numbers(22, 20)
print(my_sum)

# function with multiple return values
def add_and_sub_numbers(val_one, val_two):
    sum_of_values = val_one + val_two
    diff_of_values = val_one - val_two
    return sum_of_values, diff_of_values

my_sum, my_diff = add_and_sub_numbers(22, 20)
print(my_sum, my_diff)
```

```
None
42
42 2
```

5 Classes

Python not only encompasses functional aspects but also object oriented.

A simple example would be a String. In general a String would be an object to hold alpha-numeric data. It is possible to assign a certain value to a String (i.e. the characters to be hold), return those values, return only certain parts, etc.

While MatLab does not allow the creation of user defined objects, Python does by using `classes` to create objects.

A class can be seen as a definition of objects that not only are able to perform computations (like a function), but also to store data. The general principle thereby is, to indicate what would happen if an instance of this pre-defined class of objects is created and what this instance can do.

This is referred to as having `attributes` (to store data) and `methods` (i.e. functions). An attribute is a variable that belongs to the object (same as your eye-color is an attribute of yours), whereas methods are functions that the object can perform (like breathing).

When defining a new class all attributes that are defined right after the class definition will be attributes of the class. Those can than be accessed using `my_class.my_attribute` Within the class itself you would access the very same attribute using `self.my_attribute`

Not only can the class hold attributes but also methods, which follow the same logic as attributes, but are technically functions, not data. To call functions you would thus need to add parentheses to the call `my_class.my_function()`.

`__init__()` thereby is the method that is executed when first creating an instance of the class. All methods and attributes will be accessible from this instance.

[44]:
```
class pet:
    pet_type = None # attributes of every instance
    pet_name = None

    def __init__(self, pet_type, pet_name): # called when creating an instance
        self.pet_type = pet_type
        self.pet_name = pet_name

    def get_name(self): # methods of every instance
        return self.pet_name

    def get_type(self):
        return self.pet_type
```

[45]:
```
# create instance of the pet class
new_pet = pet('dog', 'Fiffy') # internally __init__() is called

print("My pet's name is " + new_pet.get_name()) # the instance has the methods␣
 ↪that we defined in the class definition
print("It's a " + new_pet.get_type())
```

```
My pet's name is Fiffy
It's a dog
```

Furthermore it is possible to define a class not from scratch, but instead "inheriting" attributes and methods from other classes.

E.g. If we would like to create a new "dog" class that has all the features the "pet" class had, we initialize the class definition by setting the base class to pet. This way all instances of "dog" will have the same attributes and methods as instances of "pet", but on top we can define more attributes that "dog"s have but not "pet"s.

[46]:
```
class dog(pet):
    pet_tricks = [] # additional attribute hold only by the dog (sub-) class

    def __init__(self, pet_name):
        self.pet_name = pet_name
        self.pet_type = 'dog'
```

```
    def add_trick(self, trick): # additional methods hold only by the dog␣
 ↪(sub-) class
        self.pet_tricks.append(trick)

    def get_tricks(self):
        return self.pet_tricks

class cat(pet):
    pet_toys = [] # additional attribute hold only by the cat (sub-) class

    def __init__(self, pet_name):
        self.pet_name = pet_name
        self.pet_type = 'cat'

    def add_favorite_toy(self, toy): # additional methods hold only by the cat␣
 ↪(sub-) class
        self.pet_toys.append(toy)

    def get_toys(self):
        return self.pet_toys
```

Below you can see that the "dog" class can make use of the methods that were initially defined only in the "pet" class

[47]:
```
new_dog = dog('Fiffy') # calls the dog classes __init__()

print("My pet's name is " + new_dog.get_name())
print("It's a " + new_dog.get_type())
```

```
My pet's name is Fiffy
It's a dog
```

Furthermore we have now as well access to the dog classes unique methods

[48]:
```
new_dog.add_trick('sit')
new_dog.add_trick('bark')
print("My dog " + new_dog.get_name() + " can perform the following tricks: " +␣
 ↪" and ".join(new_dog.get_tricks()))
```

```
My dog Fiffy can perform the following tricks: sit and bark
```

In general it is good practice to create methods that allow the retrieval of attribute values. However it is also possible to access attribute values directly.

[49]:
```
dogs_name = new_dog.get_name()
print("My dog's name is " + dogs_name)

cats_name = cat('Mimi').pet_name
print("My cat's name is " + cats_name)
```

```
My dog's name is Fiffy
My cat's name is Mimi
```

6 Stuff you probably wanna use daily

6.1 sets of values

[50]:
```
# create a range of values
print(list(range(0, 10))) # 10 values starting from 0
print(list(range(0, 10, 2))) # 10 values starting from zero, but only every␣
 ↪other
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[0, 2, 4, 6, 8]
```

6.2 indexing

[51]:
```
import numpy as np
np.random.seed(42) # fix random seed
my_array = np.random.rand(5,5)

print(my_array) # full array
```

```
[[0.37454012 0.95071431 0.73199394 0.59865848 0.15601864]
 [0.15599452 0.05808361 0.86617615 0.60111501 0.70807258]
 [0.02058449 0.96990985 0.83244264 0.21233911 0.18182497]
 [0.18340451 0.30424224 0.52475643 0.43194502 0.29122914]
 [0.61185289 0.13949386 0.29214465 0.36636184 0.45606998]]
```

[52]:
```
print(my_array[0,0]) # first value
```

```
0.3745401188473625
```

[53]:
```
print(my_array[1,:]) # second row
```

```
[0.15599452 0.05808361 0.86617615 0.60111501 0.70807258]
```

[54]:
```
print(my_array[:,[2,4]]) # column 3 and 5
```

```
[[0.73199394 0.15601864]
 [0.86617615 0.70807258]
 [0.83244264 0.18182497]
 [0.52475643 0.29122914]
 [0.29214465 0.45606998]]
```

[55]:
```
index_var = list(range(0,3))
index_var.append(4)
print(my_array[:,index_var]) # columns 1 to 3 and 5
```

```
[[0.37454012 0.95071431 0.73199394 0.15601864]
 [0.15599452 0.05808361 0.86617615 0.70807258]
 [0.02058449 0.96990985 0.83244264 0.18182497]
 [0.18340451 0.30424224 0.52475643 0.29122914]
 [0.61185289 0.13949386 0.29214465 0.45606998]]
```

[56]:
```
from_item = 0
to_item = 5
in_steps_of = 2
print(my_array[:,from_item:to_item:in_steps_of]) # odd numbered columns
```

```
[[0.37454012 0.73199394 0.15601864]
 [0.15599452 0.86617615 0.70807258]
 [0.02058449 0.83244264 0.18182497]
 [0.18340451 0.52475643 0.29122914]
 [0.61185289 0.29214465 0.45606998]]
```

[57]:
```
in_steps_of = 3
print(my_array[:,::in_steps_of]) # first to last in steps of 3 (1st and 4th␣
 ↪column)
```

```
[[0.37454012 0.59865848]
 [0.15599452 0.60111501]
 [0.02058449 0.21233911]
 [0.18340451 0.43194502]
 [0.61185289 0.36636184]]
```

[58]:
```
print(my_array[:,2::]) # 3rd to last column
```

```
[[0.73199394 0.59865848 0.15601864]
 [0.86617615 0.60111501 0.70807258]
 [0.83244264 0.21233911 0.18182497]
 [0.52475643 0.43194502 0.29122914]
 [0.29214465 0.36636184 0.45606998]]
```

[59]:
```
print(my_array[:,-1]) # only last column
```

```
[0.15601864 0.70807258 0.18182497 0.29122914 0.45606998]
```

[60]:
```
print(my_array[:,::-1]) # flip column order
```

```
[[0.15601864 0.59865848 0.73199394 0.95071431 0.37454012]
 [0.70807258 0.60111501 0.86617615 0.05808361 0.15599452]
 [0.18182497 0.21233911 0.83244264 0.96990985 0.02058449]
 [0.29122914 0.43194502 0.52475643 0.30424224 0.18340451]
 [0.45606998 0.36636184 0.29214465 0.13949386 0.61185289]]
```

[61]:
```
my_array = np.random.rand(5,5) * 2 -1
only_positive_indices = my_array > 0 # logical indexing
print(only_positive_indices)
```

```
[[ True False  True  True False]
 [ True False False  True  True]
 [ True False False  True False]
 [False False False  True False]
 [ True False  True  True False]]
```

[62]:
```
# equivalent to MatLab's 'find'
columnIDs, rowIDs = np.where(only_positive_indices != True) # find numeric␣
 ↪indices of 'False'

print(columnIDs)
print(rowIDs)
```

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

6.3 matrix (array) operations

[63]:
```
print(my_array.shape) #obtain shape
```

```
(5, 5)
```

[64]:
```
print(my_array.T) # transpose (Python 3 notation)
```

```
[[ 0.57035192  0.2150897   0.6167947  -0.75592353  0.32504457]
 [-0.60065244 -0.65895175 -0.39077246 -0.00964618 -0.37657785]
 [ 0.02846888 -0.86989681 -0.80465577 -0.93122296  0.04013604]
 [ 0.18482914  0.89777107  0.36846605  0.8186408   0.09342056]
 [-0.90709917  0.93126407 -0.11969501 -0.48244004 -0.63029109]]
```

[65]:
```
print(np.transpose(my_array)) # transpose using numpy
```

```
[[ 0.57035192  0.2150897   0.6167947  -0.75592353  0.32504457]
 [-0.60065244 -0.65895175 -0.39077246 -0.00964618 -0.37657785]
 [ 0.02846888 -0.86989681 -0.80465577 -0.93122296  0.04013604]
 [ 0.18482914  0.89777107  0.36846605  0.8186408   0.09342056]
 [-0.90709917  0.93126407 -0.11969501 -0.48244004 -0.63029109]]
```

[66]:
```
my_new_array = np.zeros((5,10)) # shape is given as a first argument as a tuple
my_concat_array = np.concatenate((my_array, my_new_array), axis=1) #␣
 ↪concatenate two arrays along columns
print(my_concat_array)
```

```
[[ 0.57035192 -0.60065244  0.02846888  0.18482914 -0.90709917  0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.        ]
 [ 0.2150897  -0.65895175 -0.86989681  0.89777107  0.93126407  0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.        ]
 [ 0.6167947  -0.39077246 -0.80465577  0.36846605 -0.11969501  0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.        ]
 [-0.75592353 -0.00964618 -0.93122296  0.8186408  -0.48244004  0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.        ]
 [ 0.32504457 -0.37657785  0.04013604  0.09342056 -0.63029109  0.
   0.          0.          0.          0.          0.          0.
   0.          0.          0.        ]]
```

[67]:
```
scale_mat = np.eye(4) * 2
print(np.ones((4,4)) * scale_mat) # element wise multiplication
```

```
[[2. 0. 0. 0.]
 [0. 2. 0. 0.]
 [0. 0. 2. 0.]
 [0. 0. 0. 2.]]
```

[68]:
```
print(np.matmul(np.ones((4,4)), scale_mat)) # matrix multiplication
```

```
[[2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]]
```

[69]:
```
print(np.ones((4,4)) @ scale_mat) # matrix multiplication (python 3.5+)
```

```
[[2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]
 [2. 2. 2. 2.]]
```

[ ]:
```

```