PYTHON!

"Everything is an Object"

"Everything in Python is an object". If you haven't heard this phrase already, you would probably hear it eventually anyway. It can make a little sense when you don't know much about programming. You can think of it in a way that **everything** you create in Python is going to be an object of a specific type. Integer `5` is an object, function `sum()` is an object, even the imported Excel sheet is also an object.

Say goodbye to the calculator

You can use Python only as a calculator to perform math calculations of any type (and no-one will judge you for that):

```
2+2*2
```

```
6
```

Notes: Guess what is the type of `2`? Correct, that's an object of type `int` (integer) with some unique id (which you don't need to worry about). Guess what is the resulting `6` in the example on the left? Yes, that's another object of `int` type!

You will see later that each object type has its own properties and functions.

Another big note here is that math operations order in Python is the same as we think of it "in a real-life". Thus, the result is `6` and not `8`.

Saving your objects

Save the result of a calculation:

```
x = 2 + 2*2
print(x)
print(type(x))
```

```
6
int
```

Use it later:

```
y = x - 10
print(y)
```

```
-4
```

Built-in types

You have seen `int` type already. Here are some more:

```
var = 5.5 # floating point number
print(type(var))
```

```
float
```

```
var = "Hello, Neuroscience" # string
print(type(var))
```

```
str
```

```
var = True # boolean
print(type(var))
```

```
bool
```

Operators

Operation	Note
`x + y`	sum of `x` and `y`
`x - y`	difference of `x` and `y`
`x * y`	product of `x` and `y`
`x / y`	quotient of `x` and `y`
`x // y`	floored quotient of `x` and `y`
`x % y`	remainder of `x` / `y`
`-x`	`x` negated

Comparing objects

Notes: Now that you know how to create new variables, you can compare them. This opens a lot of possibilities for further data manipulation!

Python comparison operations

Operation	Meaning
`<`	strictly less than
`<=`	less than or equal
`>`	strictly greater than
`>=`	greater than or equal
`==`	equal

Notes: Comparison always return a Boolean object which can be either `True` or `False`.
You can combine multiple comparisons together using `&` (AND) or `|` (OR) statements.
Note, that `True` equals `1` and `False` equals 0.

Boolean operations

Operation	Note
x | y	if x is False, then y, else x
x & y	if x is False, then x, else y
not x	if x is False, then True, else False

```
(sex == "Male") & (age > 45)
```

A female who is 50 years old would have `False & True` result of two comparisons, resulting in the final `False` result. And that's exactly what you wanted since you want two conditions to be `True` at the same time.

Examples

```
# example 1
x = 45
x <= 100
```

```
True
```

```
# example 2
y = 20
(x > 10) | (y < 5)
```

```
True
```

```
# example 3
result = (x > 10) & (y < 5)
print(result)
print(type(result))
```

```
False
bool
```

Making use of strings

Notes: Strings are another awesome built-in data type. It's very likely that some of the variables you will be working with in a future will be in a `str` format, so it's important to understand how to deal with them.

What are strings?

We can say that strings are sequences of (Unicode) characters.

```
x = "Neuroscience Rocks!"
# print the length of the string (how many characters are in the string)
print(len(x))
```

```
19
```

```
print("You " + "can " + "also " + "add " + "strings!")
print("Or even multiply!"*3)
```

```
You can also add strings!
Or even multiply!Or even multiply!Or even multiply!
```

```
x = "Hello" # single quotes
```

```
x = 'Hello' # double quotes
```

What are the differences? There is absolutely no difference which type of quotes you are going to use, as long you are consistent within one string (for example, `"hello'` would raise an Error).

Slicing

Python allows getting slices from the string by calling

```
string[start:end:step]
```

- `start` - index where you want to start the slicing (included)
- `end` - index where you want to end the slicing (not included)
- `step` - step of the slice (for example, every second value); default is 1

```
s = "Neuroscience Rocks!"
print(s[5:12]) # example 1
print(s[5])    # example 2
print(s[:4])   # example 3
print(s[::2])  # example 4
print(s[::-1]) # example 5
```

```
science
s
Neur
Nuocec ok!
!skcoR ecneicsorueN
```

It's important to remember, that indexes in Python start with 0!

Check occurrence

```
mRNA = "GUAUGCACGUGACUUUCCUCAUGAGCUGAU"
arginine_codon = "CGC"
arginine_codon not in mRNA
```

```
False
```

```
leucine_codon = "CUC"
leucine_codon in mRNA
```

```
True
```

BUT:

```
leucine_codon = "cuc"
leucine_codon in mRNA
```

```
False
```

An useful trick is to check whether a string hold some other string using the `in` operator. Note, that strings (and Python in general) are case sensitive, so `"CUC"` and `"cuc"` are two different strings.

Strings' methods

```
s = "Neuroscience Rocks!"

# basic string methods (does not modify the original string)
s.lower()                   # returns 'neuroscience rocks!'
s.upper()                   # returns 'NEUROSCIENCE ROCKS!'
s.startswith('brain')       # returns False
s.endswith('!')             # returns True
s.isdigit()                 # returns False
s.find('science')           # returns index of first occurrence, which is 5
s.find('Psychology')        # returns -1 since not found
s.replace('Neuro','Brain') # replaces all instances of 'Neuro' with 'Brain'

print(s)
```

```
Neuroscience Rocks!
```

Note that some (or even most) of the methods (aka functions), don't change the original object. In our example we applied multiple methods on the string `s`, but in the end it was still the same. If we want to save the modification after the method application we need to assign it to a variable.

```
s = "Neuroscience Rocks!"
# rewrite the variable
s = s.upper()
print(s)
```

```
'NEUROSCIENCE ROCKS!'
```

Collecting objects together

Notes: So far you have seen how to work with variables that hold a single object (like `age = 20` or `DNA = "ATGC"`). But wouldn't it be great if you could store multiple objects in one variable (for example, variable with all participants' age)?

There are 4 commonly used collection types in Python that you will see in the next slides:

- Lists
- Tuples
- Sets
- Dictionaries

Lists

List is a collection which is ordered and changeable. Allows duplicate members.

```
age = [15, 25, 45, 16, 18, 20, 25]
print(type(age))
```

```
list
```

```
age[:5])   # returns [15, 25, 45, 16, 18]
age[3:7])  # returns [16, 18, 20, 25]
age[::-1]  # returns [25, 20, 18, 16, 45, 25, 15]
age[-2]    # returns 20
print(age) # object stayed unchanged
```

```
[15, 25, 45, 16, 18, 20, 25]
```

Notes: Remember the slicing we did for the strings? Exactly the same idea applies to the lists. And no matter what type of slicing you do, as long as you don't rewrite your variable, it will stay unchanged.

Basic operations with lists

```
age = [15, 25, 45, 16, 18, 20, 25]

len(age) # returns 7
min(age) # returns 15
max(age) # returns 45
sum(age) # returns 164

avg_age = sum(age) / len(age)
print(round(avg_age,2))
```

```
23.43
```

Lists can hold objects of different types:

```
i_am_valid_list = [1, "Hello", [1,2,False], True-0, 42<3.14]
print(i_am_valid_list)
```

```
[1, 'Hello', [1, 2, False], 1, False]
```

Notes: There are some trivial built-in functions like `sum()`, `max()`, `min()` that could be applied to lists. There is no built-in `avg()` or `mean()` function, but you could easily calculate it yourself.

Keep in mind, that list can hold objects of different types, even another lists. Some functions like `sum()` wouldn't work in that case since you cannot take the sum of string and number for obvious reasons.

```python
participants = ['Bob', 'Bill', 'Sarah', 'Max', 'Jill']
# methods that modify the initial list
participants.append('Jack') # append one element to the end
# ['Bob', 'Bill', 'Sarah', 'Max', 'Jill', 'Jack']
participants.extend(['Anna', 'Bill']) # append multiple elements to end
# ['Bob', 'Bill', 'Sarah', 'Max', 'Jill', 'Jack', 'Anna', 'Bill']
participants.insert(0, 'Louis') # insert element at index 0 (shifts everything to the right)
# ['Louis', 'Bob', 'Bill', 'Sarah', 'Max', 'Jill', 'Jack', 'Anna', 'Bill']
participants.remove('Jill') # searches for first instance and removes it
# ['Louis', 'Bob', 'Bill', 'Sarah', 'Max', Jack', 'Anna', 'Bill']
participants.pop(1) # removes the element at index 0 and returns it
# ['Louis', 'Bill', 'Sarah', 'Max', Jack', 'Anna', 'Bill']

# not a method, but in this way you can change the value(s) of the list
participants[2] = 'Ben' # replace element at the index 2
# ['Louis', Bill', 'Ben', 'Max', Jack', 'Anna', 'Bill']
```

```python
# methods that don't change a sting and return a new object
print(participants.count('Bill')) # returns the number of instances;
print(participants.index('Max'))  # returns index of first instance;
```

```
2
3
```

It's important to note that some of these methods change the initial string!

The best way to know whether the method changes the object or returns new objects is the documentation.

Tuples

Tuple is a collection which is ordered and unchangeable. Allows duplicate members.

```
brain_lobes = ('frontal', 'parietal', 'temporal', 'occipital')
# or:
# brain_lobes_list = ['frontal', 'parietal', 'temporal', 'occipital']
# brain_lobes = tuple(brain_lobes_list)
print(type(brain_lobes))
```

```
tuple
```

```
brain_lobes[0] = 'anterior'
```

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

We defined lists using the square brackets (`[1,2,3]`), but for tuples we use parentheses (`(1,2,3)`).

Tuples are quite "boring" since they don't have so many methods that can be applied to them. But there is a reason for that. Tuples are **unchangeable**. This means that no function or method can change objects in the tuple.

Sets

Set is a collection which is unsorted and un-indexed. No duplicate members.

```
languages = {'python', 'r', 'java'} # create a set directly
snakes = set(['cobra', 'viper', 'python']) # create a set from a list
```

Set operations:

```
languages & snakes # intersection, AND
```

```
{'python'}
```

```
languages | snakes # union, OR
```

```
{'cobra', 'java', 'python', 'r', 'viper'}
```

```
languages - snakes # set difference
```

```
{'java', 'r'}
```

Figure brackets are the indicator for sets, another collection type. You cannot access items in a set by referring to an index since sets are unordered and have **no index**. But you can loop through the set items using a `for` loop, or ask if a specified value is present in a set, by using the `in` keyword.

You can apply basic sets commands (like union or intersection). Note that we didn't get `'python'` twice for the union since sets consist only of unique values. This fact can become handy used when looking for the unique values in a list.

Dictionaries

Dictionaries are: unordered, iterable, mutable.

```
participant = {'name': 'Jon Doe', 'group': 'Control', 'age': 42}
print(participant['name'])
```

```
Jon Doe
```

```
# add new key-value pair to the dictionary
participant['ID'] = 'CJD'
print(participant)
```

```
{'name': 'Jon Doe', 'group': 'Control', 'age': 42, 'ID': 'CJD'}
```

```
participant.keys()
```

```
['name', 'group', 'age', 'ID']
```

```
my_dict.values()
```

```
['Jon Doe', 'Control', 42, 'CJD']
```

Dictionaries are structures which can contain multiple data types, and is ordered with key-value pairs: for each unique key, the dictionary has one value. Keys can be strings, numbers, or tuples, while the corresponding values can be any Python object.

```
dict_obj = {
    'key1': value1,
    'key2': value2,
    ...}
```

As you can see from the first example, you cannot access values of the dictionary by the indexes (like you did in lists). But you can access them by the key. Due to this feature dictionaries don't allow duplicated keys.

You can also access just the keys or just the indexes by `.keys()` and `.values()` methods.

`if` statement

General form:

```
if condition is True:
    do something
elif another condition is True:
    do something
else:
    do something else
```

```
x = 100
y = 500

if x > y:
    print('X is greater than Y')
elif x == y:
    print('X equals Y')
else:
    print('X is smaller than Y')
```

```
X is smaller that Y
```

We have looked at the comparisons in a previous chapter (like `==`, `!=` or `<`). Now we want to take some actions which will depend on the outcome of the comparison.

The form is pretty straightforward. `if` something is `True`, take one action, if something else (`elif`) is true, take another action. If none of the statements were true, do something else (`else`). You can see the trivial example by comparing `x` and `y` variables.

A couple of comments:

1. 4 spaces at the beginning of the line are there for a reason. They tell Python that this line corresponds to the `if` statement body. There can be more than one line;
2. You can skip `else` statement in case you don't want to take any actions when the above conditions were false;
3. You can add as many `elif` statements as you want to make a sophisticated pattern or you can skip it at all;
4. You can combine multiple comparisons, like `if (x>5) & (y<10)`.

`while` loop

Using the `while` loop we execute the set of statements as long as condition is `True`.

General form:

```
while condition is True:
    do something
```

```
x = 0

while x < 4:
    x += 1 # which is the equivalent of `x = x + 1`
    print(x)
```

```
1
2
3
4
```

You can also say that you want to keep on taking some actions as long as condition is `True` using `while` loop.

`for` loops

A `for` loop is used for iterating over a sequence (like lists, tuples, dictionaries, sets, strings).

```
my_list = ['data', 'science', 'rocks']

for word in my_list:      # iterate over all values in the list
    print(word.upper())  # change to upper register and print out the value
```

```
DATA
SCIENCE
ROCKS
```

```
stdev = [2, 4, 1.5, 2, 4]      # list of standard deviations

variances = []                 # initialize the empty list that will hold variances
for val in stdev:              # iterate over all values in the list
    variances.append(val**2)   # append the variances list with squared value

print(variances)
```

```
[4, 16, 2.25, 4, 16]
```

Notes: Unlike `while` loops, `for` loops don't require any condition. But they require an ***iterable object*** (like a list).

In the first example, we were iterating over the list `my_list`, which consists of three strings. At each step of the loop, we defined a temporary variable `word` (name `word` is arbitrary, you can call it as you wish) as a value from the lists.

- *Step 1*. `word` = `'data'`
- *Step 2*. `word` = `'science'`
- *Step 3*. `word` = `'rocks'`
- Since no more objects left in the list, exit the loop.

The second example shows how you can get take every number to the power of two using `for` loop.

List comprehensions

List comprehensions provide a concise way to create lists.

```
stdev = [2, 4, 0, 1.5, 2, 4]
variances = [val**2 for val in stdev]
print(variances)
```

```
[4, 16, 0, 2.25, 4, 16]
```

```
variances = [val**2 for val in stdev if val != 0]
print(variances)
```

```
[4, 16, 2.25, 4, 16]
```

```
variances = [val**2 if val != 0 else "wrong value" for val in stdev]
print(variances)
```

```
[4, 16, "wrong value", 2.25, 4, 16]
```

Notes: List comprehensions allow creating a new list in a shorter way using `for` loops directly inside the list. However, sometimes such lines of codes become "harder" to read.

You can even add `if` / `else` statement. Note the difference in the order of statements with and without `else`.

Functions

A function is a block of code that only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

General form:

```
def my_new_function(some_argument1, some_argument2):
    do something
    return something
```

Example:

```
def mean(input_list):
    nominator = sum(input_list)
    denominator = len(input_list)
    return nominator/denominator


l = [1,2,3,4,5]
print(mean(input_list=l))
```

```
3.0
```

You have seen the examples of several built-in functions like `len()`, `max()`, `sorted()`!

In the `mean()` function we specified that there is going to be just one argument (`input_list`). That will hold the value of an object we pass to the function when we call it.

In the function's body, we create two new variables that hold the values of sum and length. And at the end, we return the value of a division. The last step is very important since now `mean()` function returns an object, that could be passed to a variable and used later.

Arguments of the function

```
def get_pi():
    pi = 22/7
    return pi

print(get_pi())
```

```
3.142857142857143
```

```
def get_pi(circumference, diameter, digits_to_round=2):
    pi = circumference/diameter
    return round(pi, digits_to_round)

print(get_pi(circumference=22, diameter=7))
print(get_pi(circumference=22, diameter=7, digits_to_round=1))
```

```
3.14
3.1
```

Notes: Functions can take no input arguments, like in the `get_pi()` example. In such a case, the result of the function will be always the same.

You can specify as many input arguments as you wish. Also, arguments can have default values (like `digits_to_round=2`). This means that if you don't specify its value in the function run, it will be taken from the default value.

`digits_to_round` was used inside the `round()` function and specified the numbers we want to keep after the coma. In the first run, we didn't specify `digits_to_round`, so it was set to `2`. In the first run, we set it to `1` in the function call.

Numerical operations with NumPy

Welcome `numpy`

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

Website || Documentation

Importing NumPy:

```
import numpy as np

# square root function
print(np.sqrt(4))
```

```
2.0
```

Arrays

NumPy is great for doing vector arithmetic. A NumPy array is a grid of values, all of the same type (i.e. it is a matrix), and is indexed by a tuple of non-negative integers.

```
a = np.array([1, 3, 5])
print(type(a))
```

```
numpy.ndarray
```

```
b = np.array([1, 2, "hello"])
print(b)
```

```
['1' '2' 'hello']
```

```
c = np.array(
    [[1,2],
     [3,4]]
    )
print(c.shape)
```

```
(2,2)
```

Vector operations

Convert values in an array from Celsius to Fahrenheit:

```
# numpy array
temperature_array = np.array([18.0, 21.5, 21.0, 21.0, 18.8, 17.6, 20.9, 20.0])
temperature_array = temperature_array * 9/5 + 32
print(temperature_array)
```

```
[64.4, 70.7, 69.8, 69.8, 65.84, 63.68, 69.62, 68.0]
```

NumPy makes life easier through vectorization. We can apply operations directly on an array without calling any mapping functions, loops, or so on.

Linear algebra (1)

```
A = np.array(
    [[6, 1, 1],
     [4, -2, 5],
     [2, 8, 7]]
    )

print("Rank of A:", np.linalg.matrix_rank(A))
print("\nTrace of A:", np.trace(A))
print("\nDeterminant of A:", np.linalg.det(A))
print("\nInverse of A:\n", np.linalg.inv(A))
```

```
Rank of A: 3

Trace of A: 11

Determinant of A: -306.0

Inverse of A:
 [[ 0.17647059 -0.00326797 -0.02287582]
  [ 0.05882353 -0.13071895  0.08496732]
  [-0.11764706  0.1503268   0.05228758]]
```

Notes: The Linear Algebra module of NumPy offers various methods to apply linear algebra on any numpy array. One can find:

- rank, determinant, trace, etc. of an array;
- eigenvalues/vectors of matrices;
- matrix and vector products (dot, inner, outer product), matrix exponentiation;
- solve linear or tensor equations, and much more!

Linear algebra (2)

Solving the equation:

```
A = np.array(
    [[1,2],
     [3,4]])

B = np.array([10, 20])

x = np.linalg.solve(A,B)
print(x)
```

```
[0. 5.]
```

Notes: Example of solving the linear equations. Here we distinguished that *x1* = 0, *x2* = 0.5.

Packages for data visualization

Matplotlib

Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.

Website || Documentation || Gallery

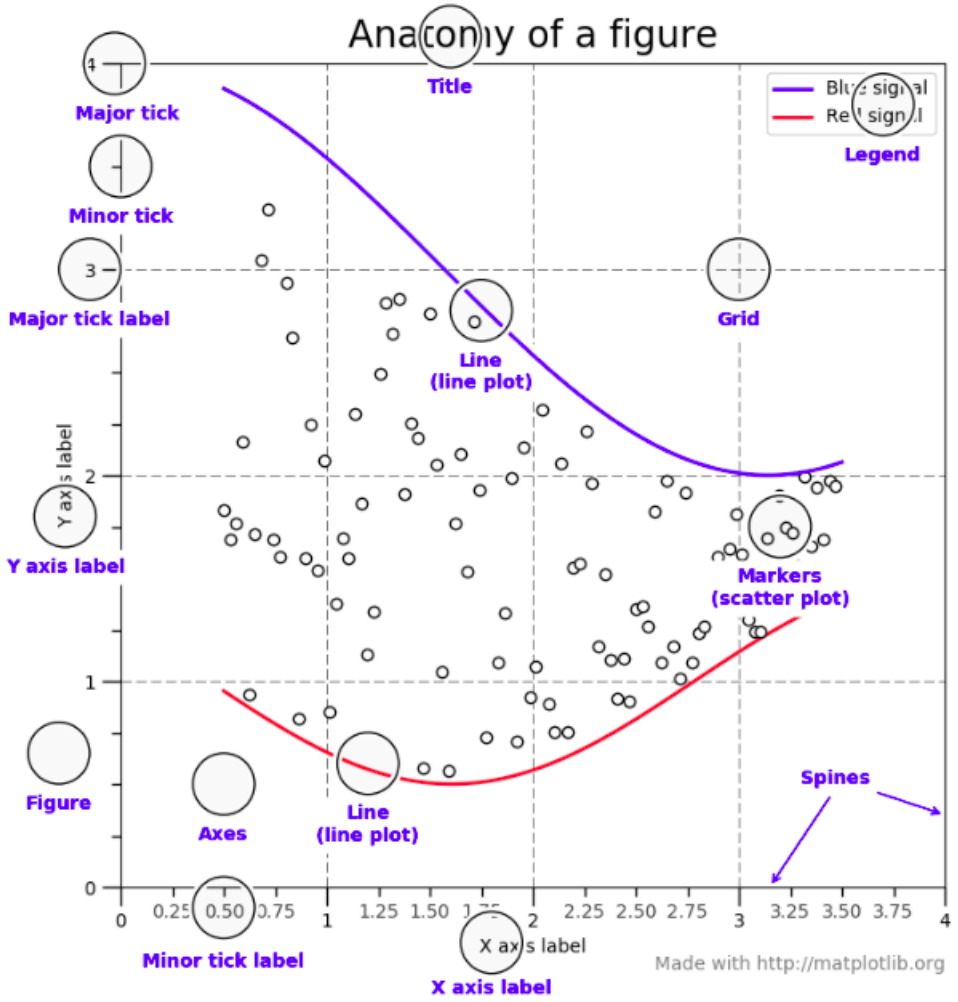
Importing `matplotlib`:

```
import matplotlib.pyplot as plt
```

Notes: Matplotlib must be the most commonly used package for data visualization in Python. If you have worked in MATLAB, you might see a lot of similarities in the plotting syntax (hence the name). Matplotlib has a gallery with the possible graphs you can make, which makes it easy to adapt the code for your own problem.

Most of the time we don't need the whole package, but just a module `pyplot`, with the alias name `plt`.

Anatomy of a figure

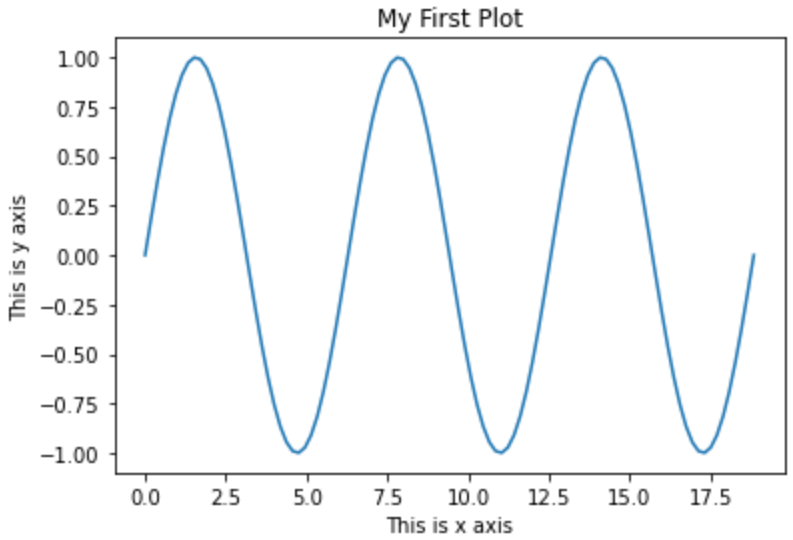


Notes: Here are some possible parameters you can change in the plot, such as title, legend, grid, axis labels, grid and so on.

Syntax basics

```
x = np.linspace(start=0, stop=6*np.pi, num=100)
y_sin = np.sin(x)

plt.figure() # start
plt.plot(x, y_sin)
plt.title("My First Plot")
plt.xlabel("This is x axis")
plt.ylabel("This is y axis")
plt.show()   # end
```



Notes: One of the way to think about the plotting syntax is in this way.

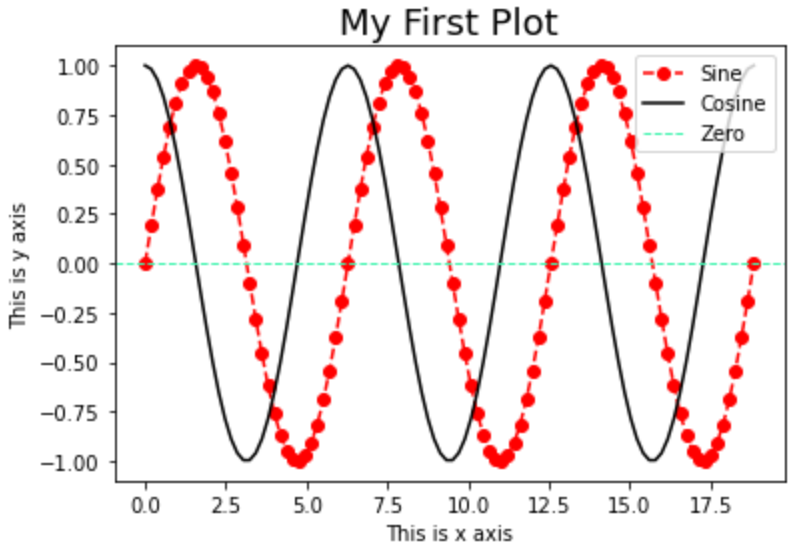
1. Start new figure with `plt.figure()`
2. Add parameters you need, such as line, bars, labels, legend, etc.
3. Stop figure creation by `plt.show()`

`plt.plot()` can be used to create a line plot (be default) or scatter plot (by adding `"o"`), so it is not a "universal" plotting function, as it could seen.

Adding more objects to the figure

```
y_cos = np.cos(x)

plt.figure() # start
plt.plot(x, y_sin, 'o--', color='r', label='Sine')
plt.plot(x, y_cos, color='black', label='Cosine')
plt.axhline(y=0, linewidth=1, color='#42f5b0', linestyle='dashed', label='Zero')
plt.title("My First Plot", fontsize=18)
plt.xlabel("This is x axis")
plt.ylabel("This is y axis")
plt.legend()
plt.show()   # end
```



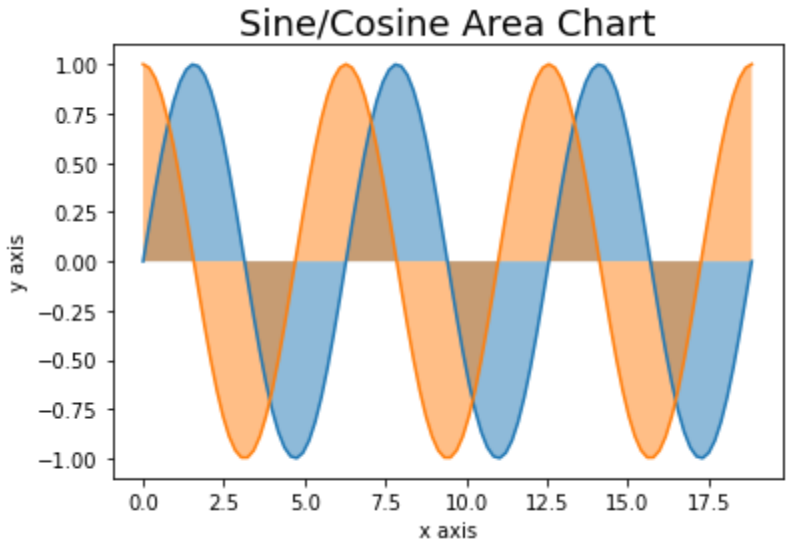
Notes: Here is an example of how you can add more objects to the figure. Note that order is not important, as long as you keep all the new plotting objects between `plt.figure()` and `plt.plot()`.

Some explanations:

- `'o--'` in `plt.plot()` is a shortcut to make a scatter plot connected with a dashed line;
- `color` can be in a string format with a full word (`"green"`), with one letter (`"g"` , meaning green), HEX color (`"#ffffff"`) or RGB color as a list (`(0.1, 0.2, 0.5)`) (and maybe even more other options);
- `label` is responsible for assigning a name to an object on a legend. In order to show the legend we need to add `plt.legend()` ;
- `plt.axhline()` creates a `h` orizontal `line` through the `ax` is. There is also `plt.axvline()` .

Area chart

```
plt.figure()
plt.plot(x, y_sin, x, y_cos)
plt.fill_between(x, y_sin, alpha=0.5)
plt.fill_between(x, y_cos, alpha=0.5)
plt.xlabel('x axis')
plt.ylabel('y axis')
plt.title("Sine/Cosine Area Chart", fontsize=18)
plt.show()
```

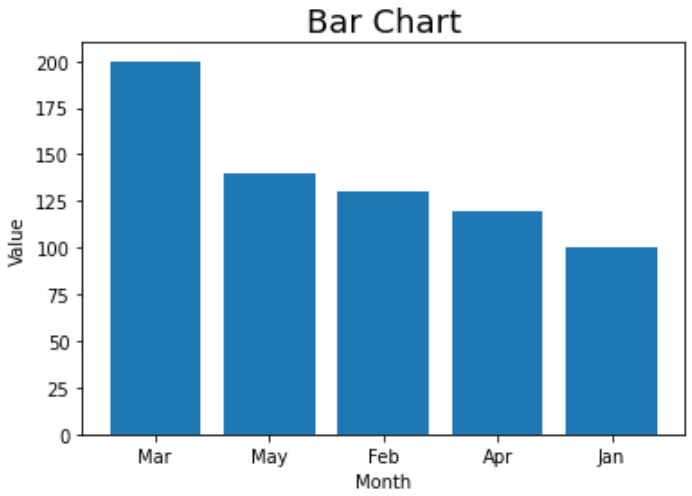


Notes: One of the ways to create an area chart is to use `plt.fill_between()` function. `alpha` is responsible for opacity (`0` : object is transparent, `1` : full color).

Bar plot

```
df = pd.DataFrame(
    {"month": ["Jan", "Feb", "Mar", "Apr", "May"],
     "value": [100, 130, 200, 120, 140]})
df.sort_values(by="value", ascending=False, inplace=True)

plt.figure()
plt.bar(df["month"], df["value"])
plt.title("Bar Chart", fontsize=18)
plt.xlabel("Month")
plt.ylabel("Value")
plt.show()
```



Notes: `plt.bar()` displays the bar in a given order from the DataFrame/list. So if you want to represent bars in a specific order (for example, descending as on the example), you have to perform some manipulations on the DataFrame before calling the plot function.