EPL448: Data Mining on the Web – Lab 3

University of Cyprus
Department of Computer Science

Παύλος Αντωνίου

Γραφείο: B109, ΘΕΕ01

Python

- Open source, general-purpose language
- Object Oriented, Procedural, Functional
- Easy to interface with C/ObjC/Java/Fortran
- Easy-ish to interface with C++ (via SWIG)
- Great interactive environment (python idle)

- Official Website: http://www.python.org
  - Documentation: http://www.python.org/doc/
  - Free book: Dive into Python
- **Download powerful enterprise-ready open data-science platform: Anaconda https://www.anaconda.com/**
- **OR use Google Collab: https://colab.research.google.com/**

Install Anaconda (Individual Edition) on Windows

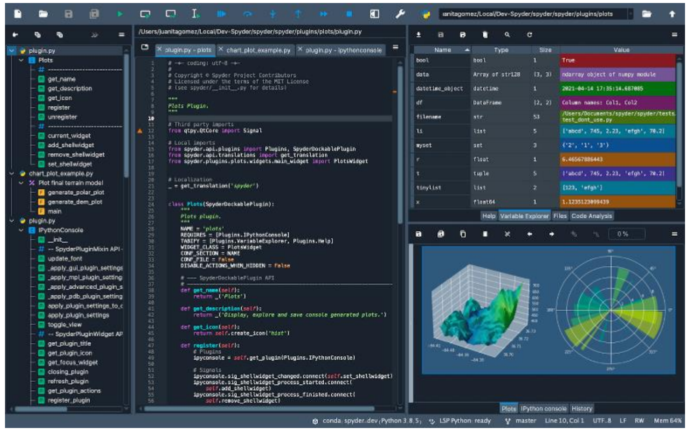
- Go to https://www.anaconda.com/products/individual#Downloads

- Download 64-bit or 32-bit installer depending on your machine architecture

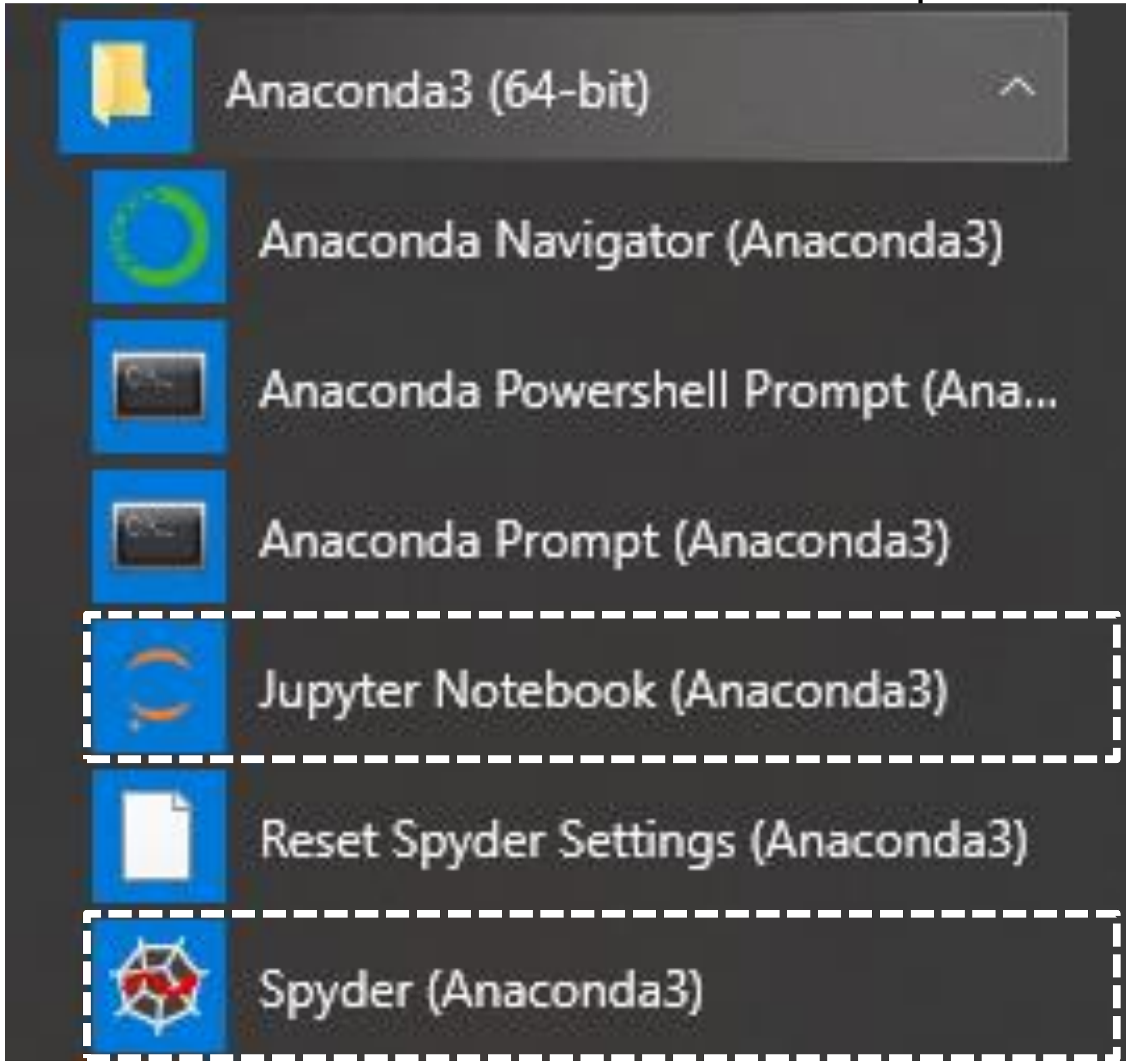
- Double-click the **.exe** file to install Anaconda and follow the instructions on the screen

Python development

- Python Spyder IDE



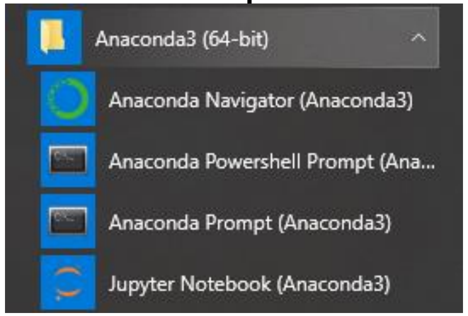
- Jupyter Notebook or Jupyter Lab
  - open-source web application that enables users to create and share documents that combine live code with narrative text, mathematical equations, visualizations, interactive controls, and other rich output.

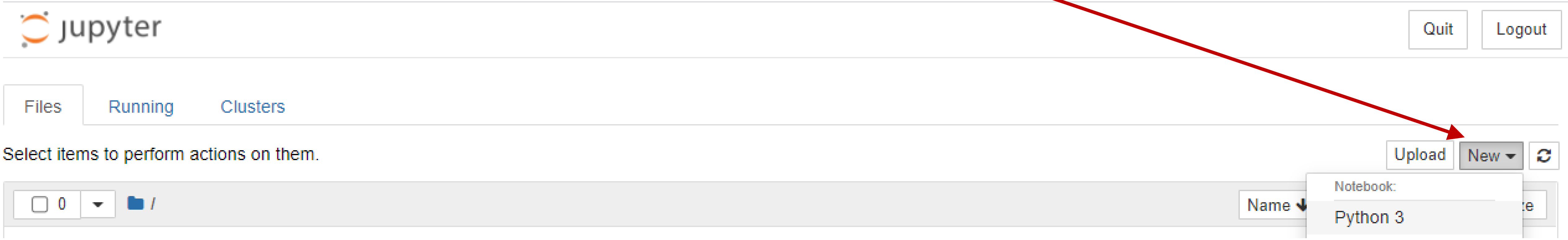


- Google Collab Notebook: https://colab.research.google.com/

Option 1: Jupyter Notebook

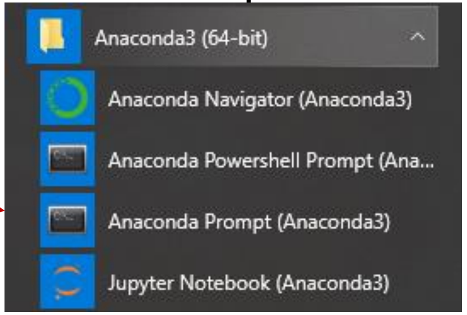
- Open Jupyter Notebook
- Create new Python 3 Notebook

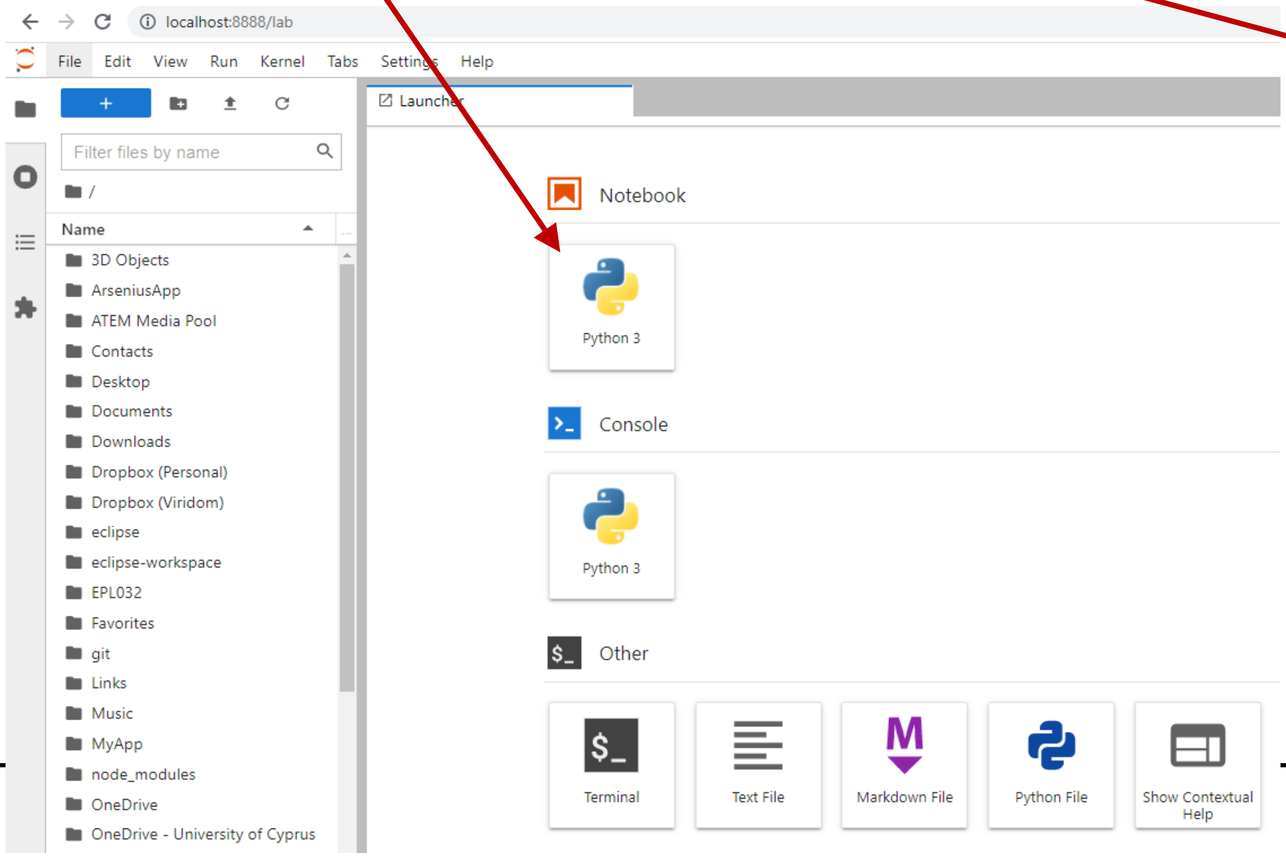


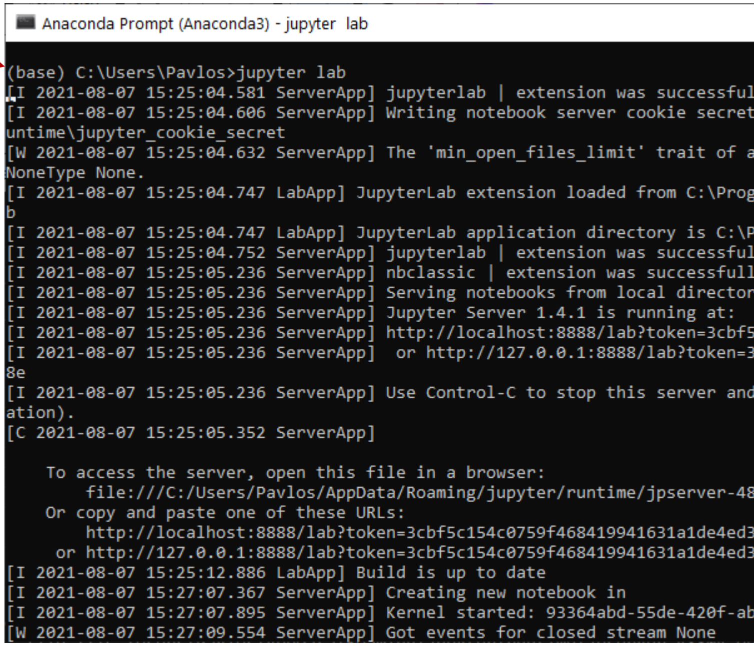


Option 2: Jupyter Lab

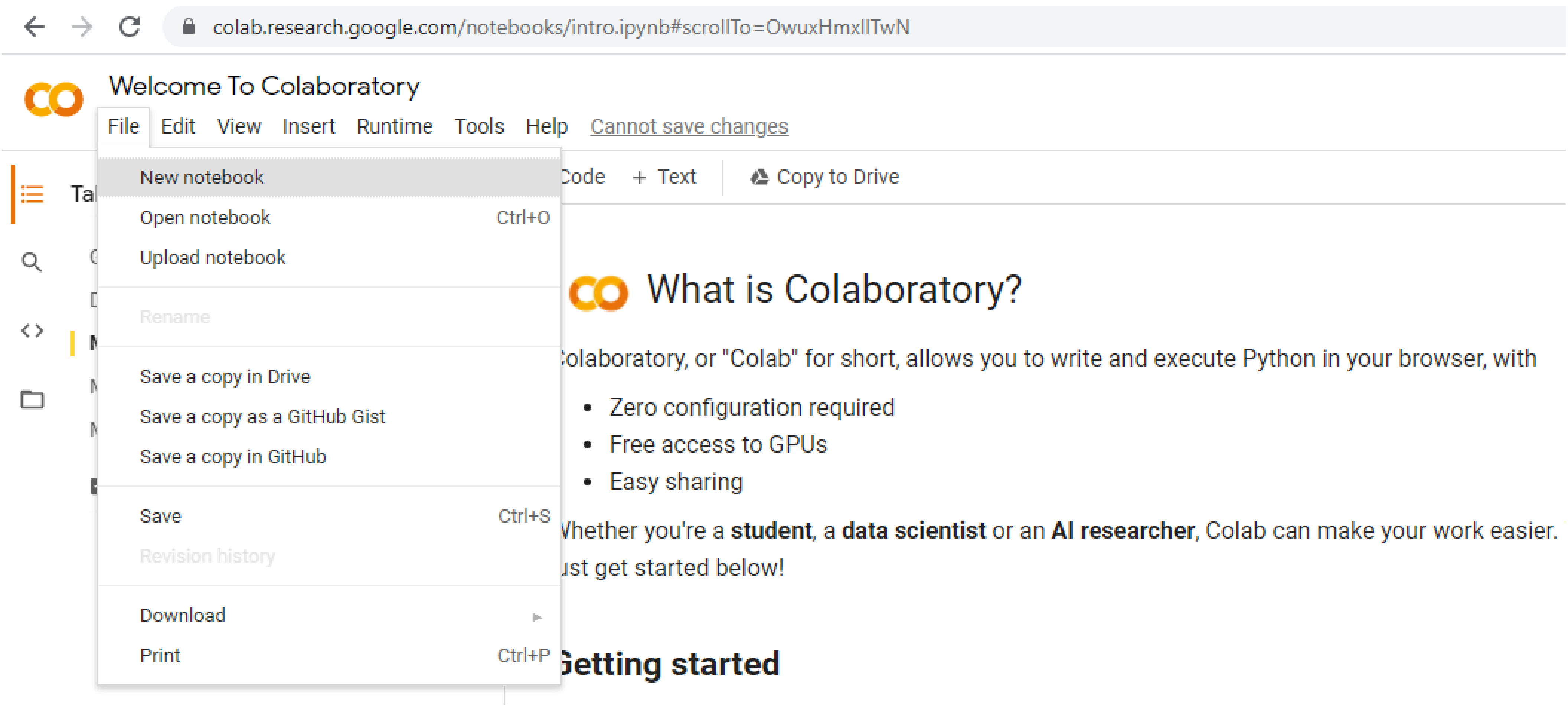
- Open Anaconda Prompt
- Type `jupyter lab`
- Open Python 3 Notebook



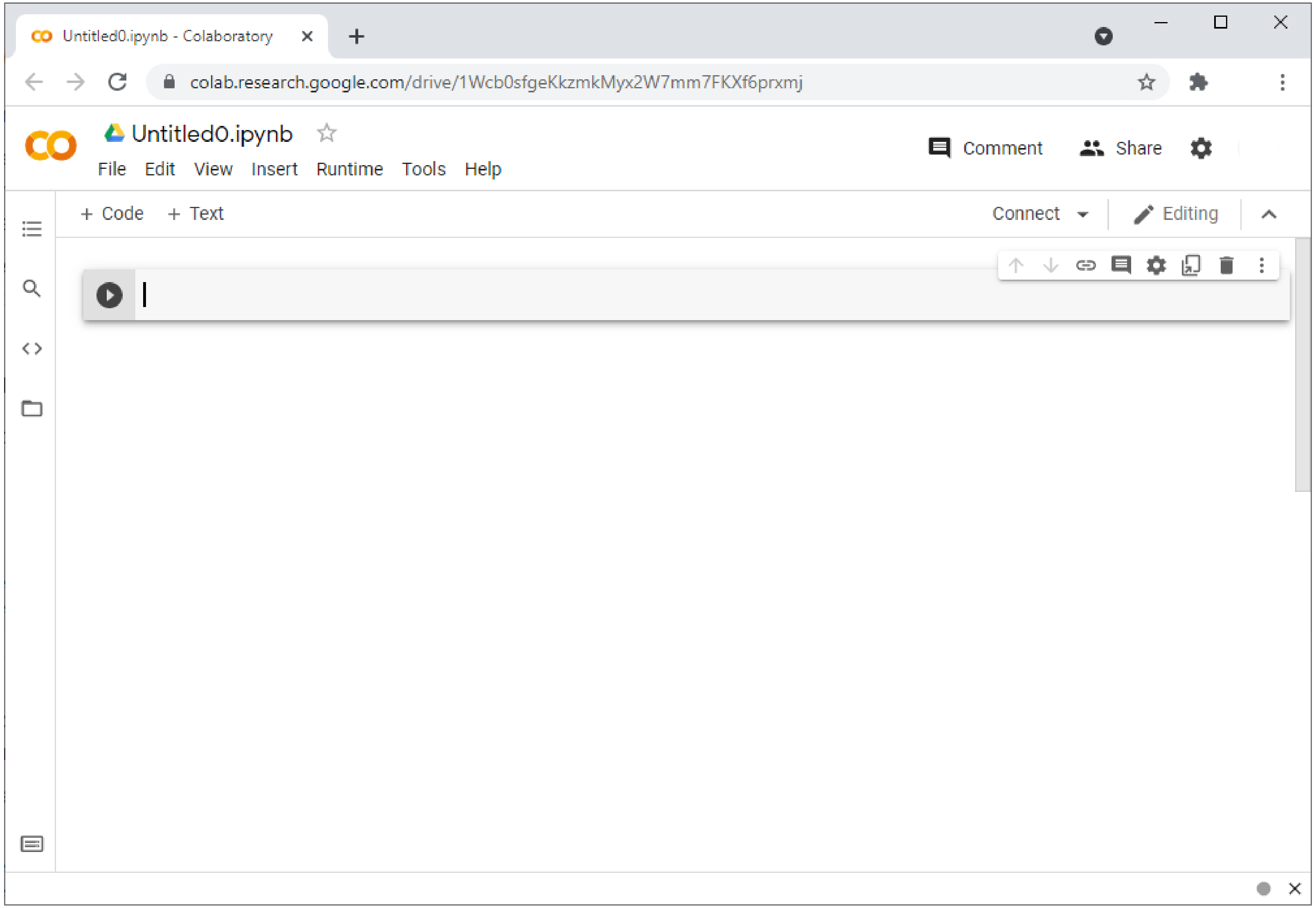




Option 3: Google Collab



Option 3: Google Collab Notebook



A Code Sample

```
x = 34 - 23                # A comment.
y = "Hello"                # Another one.
z = 3.45
if z == 3.45 or y == "Hello":       # colon needed
    x = x + 1              # similar to x += 1.
    y = y + " World"       # String concatenation.
print(x)                   # 12
print(y)                   # Hello World
x = y
print(x)                   # Hello World
```

Enough to Understand the Code

- Assignment uses `=` and comparison uses `==`
- For numbers `+ - * / %` are as expected
  - Special use of `+` for string concatenation
  - Special use of `%` for string formatting (as with printf in C)
    - `print("%d + %d = %d" %(x,y,x+y))`
- Logical operators are words (`and`, `or`, `not`) **not** symbols
- The basic printing command is `print()`
- The first assignment to a variable creates it
  - Variable types don’t need to be declared
  - Python figures out the variable types on its own

Multiple ways of printing

```python
a = 10
b = 20
c = a + b

# Normal string concatenation, space is automatically printed
# in the position of each comma
print("sum of", a , "and" , b , "is" , c)

# convert variables into str
print("sum of " + str(a) + " and " + str(b) + " is " + str(c))

# if you want to print in tuple way (C-like way)
print("sum of %d and %d is %d" %(a,b,c))

# New style string formatting
print("sum of {0} and {1} is {2}".format(a,b,c))
```

Numeric Datatypes

- Integer numbers (int)
  - `z1 = 23`
  - `z2 = 5 // 2        # Answer is 2, integer division.`
  - `z3 = int(6.7)      # Converts 6.7 to integer. Answer is 6.`
  - Booleans (bool) are a subtype of integers
- Floating (float) point numbers (implemented using double in C)
  - `x1 = 3.456`
  - `x2 = 5 / 2         # Answer is 2.5`
- Complex numbers
- Additional numeric types: fractions, decimal
- See more [here]

Newlines and Whitespaces

- Use a newline to end a line of code.
  - Use \ when must go to next line prematurely.
- Whitespace is meaningful in Python: especially indentation
- No braces { } to mark blocks of code in Python…
  Use consistent indentation – whitespace(s) or tab(s) – instead.
  - The first line with more indentation starts a nested block
  - The first line with less indentation is outside of the block
  - Indentation levels must be equal within the same block

```
if x%2 == 0:
   print("even")
   print("number")
else:
 print("odd")
```

- Often a colon appears at the start of a new block.
  - e.g. in the beginning of `if`, `else`, `for`, `while`, as well as function and class definitions

Comments

- Single line comments: Start comments with **#** – the rest of line is ignored by the python interpreter
- Multiple line comments: Start/end comments with `"""`
- Can include a "documentation string" as the first line of any new function or class that you define.
- The development environment, debugger, and other tools use it: it's good style to include one.

```
def my_function(x, y):
    """This is the docstring. This
    function does blah blah blah."""
    # The code would go here...
```

Naming Rules

- Names are case sensitive and cannot start with a number. They can contain letters, numbers, and underscores.

```
bob Bob _bob _2_bob_ bob_2 BoB
```

- There are some reserved words:

```
and, assert, break, class, continue, def, del,
elif, else, except, exec, finally, for, from,
global, if, import, in, is, lambda, not, or,
pass, print, raise, return, try, while
```

Some Python datatypes (objects)

- Some **immutable** objects
  - **int**
  - **float**
  - decimal
  - complex
  - bool
  - **string**
  - **tuple**
  - bytes
  - **range**
  - frozenset

Numeric datatypes

Sequences

Set type

- Some **mutable** objects
  - **list**
  - bytearray
  - **set**
  - **dict**
  - user-defined classes (unless specifically made immutable)

Sequences

Set type

Mapping

❖ When we change these data, this is done in place.
❖ They are not copied into a new memory address each time.

Immutable Sequences I

- **Strings**
  - Defined using double quotes `""` or single quotes `''`

    ```
    >>> st = "abc"
    >>> st = 'abc'
    ```
    (Same thing.)
  - Can occur within the string.

    ```
    >>> st = "matt's"
    ```
  - Use triple double-quotes for multi-line strings or strings than contain both ‘ and “ inside of them:

    ```
    >>> st = """This is a multi-line
    string that uses triple quotes."""
    >>> st = """a'b"c"""
    ```

Immutable Sequences II

- **Tuples (Πλειάδες)**
  - A simple **immutable** ordered sequence of items of mixed types
  - Defined using parentheses (and commas) or using `tuple()`.

```
>>> t = tuple()          # create empty tuple
>>> tu = (23, 'abc', 4.56, (2,3), 'def')   # another tuple
>>> tu[2] = 3.14
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

  - You can't change a tuple.
  - You can make a fresh tuple and assign its reference to a previously used name.

```
>>> tu = (23, 'abc', 3.14, (2,3), 'def')
```

Immutable Sequences III : data access

- We can access individual members of a **tuple** or **string** using square bracket “array” notation.
- Positive index: count from the left, starting with 0.
- Negative index: count from right, starting with –1.

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
>>> tu[1] # Second item in the tuple.
'abc'
>>> tu[-3]
4.56
>>> st = "Hello World"
>>> st[1] # Second character in string.
'e'
```

Mutable Sequences I

- **Lists**
  - **Mutable** ordered sequence of items of mixed types
  - Defined using square brackets (and commas) or using `list()`.

```
>>> li = ["abc", 34, 4.34, 23]
```

  - We can access individual members of a list using square bracket “array” notation as in tuples and strings.

```
>>> li[1] # Second item in the list.
34
```

  - We can change lists in place.
    - Name li still points to the same memory reference when we’re done.
    - The mutability of lists means that they aren’t as fast as tuples.

```
>>> li[1] = 45
>>> li
['abc', 45, 4.34, 23]
```

Tuples vs. Lists

- Lists slower but more powerful than tuples.
  - Lists can be modified, and they have lots of handy operations we can perform on them.
  - Tuples are immutable and have fewer features.
- To convert between tuples and lists use the list() and tuple() functions:

```
li = list(tu)
tu = tuple(li)
```

Slicing in Sequences: Return Copy of a Subset 1

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Return a **copy** of the container with a subset of the original members. Start copying at the first index, and stop copying before the second index.

```
>>> tu[1:4]
('abc', 4.56, (2,3))
```

- You can also use negative indices when slicing.

```
>>> tu[1:-1]
('abc', 4.56, (2,3))
```

Slicing in Sequences: Return Copy of a Subset 2

```
>>> tu = (23, 'abc', 4.56, (2,3), 'def')
```

- Omit the first index to make a copy starting from the beginning of the container.

```
>>> tu[:2]
(23, 'abc')
```

- Omit the second index to make a copy starting at the first index and going to the end of the container.

```
>>> tu[2:]
(4.56, (2,3), 'def')
```

Copying the Whole Sequence

- To make a copy of an entire sequence, you can use [:].

```
>>> tu[:]
(23, 'abc', 4.56, (2,3), 'def')
```

- Note the difference between these two lines for mutable sequences:

```
>>> list2 = list1        # 2 names refer to 1 reference
                         # Changing one affects both
>>> list2 = list1[:]     # Two independent copies, two refs
>>> list2 = list(list1)# Two independent copies, two refs
```

The ‘in’ Operator

- Boolean test whether a value is inside a container:

```
>>> li = [1, 2, 4, 5]
>>> 3 in li
False
>>> 4 in li
True
>>> 4 not in li
False
```

- For strings, tests for substrings

```
>>> a = 'abcde'
>>> 'c' in a
True
>>> 'cd' in a
True
>>> 'ac' in a
False
```

- Be careful: the `in` keyword is also used in the syntax of for loops and list comprehensions.

The + Operator

- The + operator produces **a <u>new</u> string, tuple, or list** whose value is the concatenation of its arguments.

```
>>> "Hello" + " " + "World"
'Hello World'
>>> (1, 2, 3) + (4, 5, 6)
(1, 2, 3, 4, 5, 6)
>>> [1, 2, 3] + [4, 5, 6]
[1, 2, 3, 4, 5, 6]
```

The * Operator

- The * operator produces **a <u>new</u> string, tuple, or list** that “repeats” the original content.

```
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> "Hello" * 3
'HelloHelloHello'
```

Operations on Lists Only 1

```
>>> li = [1, 11, 3, 4, 5]
>>> li.append('a') # Our first exposure to method syntax
>>> li
[1, 11, 3, 4, 5, 'a']
>>> li.insert(2, 'i')
>>> li
[1, 11, 'i', 3, 4, 5, 'a']
```

Operations on Lists Only 2

- extend operates on list li in place.

```
>>> li.extend([9, 8, 7])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7]
```

- Confusing:
  - Extend takes a list as an argument.
  - Append takes a singleton as an argument.

```
>>> li.append([10, 11, 12])
>>> li
[1, 2, 'i', 3, 4, 5, 'a', 9, 8, 7, [10, 11, 12]]
```

Operations on Lists Only 3

```
>>> li = ['a', 'b', 'c', 'b']
>>> li.index('b')       # index of first occurrence
1
>>> li.count('b')       # number of occurrences
2
>>> li.remove('b')      # remove first occurrence
>>> li
['a', 'c', 'b']
```

Operations on Lists Only 4

```
>>> li = [5, 2, 6, 8]
>>> li.reverse()  # reverse the list *in place*
>>> li
[8, 6, 2, 5]
>>> li.sort()     # sort the list *in place*
>>> li
[2, 5, 6, 8]
>>> li.sort(some_function) # sort in place using user-
defined comparison
```

Sets: A set type

- Sets store unordered, finite sets of unique, immutable objects
  - **Sets are mutable**, cannot be indexed.
  - Defined using { } (and commas) or `set()`
  - Common uses:
    - fast membership testing
    - removing duplicates from a sequence
    - computing mathematical operations such as intersection, union

```
>>> se1 = set()                              # create an empty set
>>> se2 = {"arrow", 1, 5.6}                  # create another set
>>> se2.add("hello")
>>> print(se2)
{1, 'hello', 5.6, 'arrow'}
>>> se2.remove("hello")
>>> print(se2)
{1, 5.6, 'arrow'}
```

Dictionaries: A Mapping type

- Dictionaries store a mapping between a set of keys and a set of values.
  - Dictionaries are **mutable**
  - Keys can be any immutable type.
  - Values can be any type
  - A single dictionary can store values of different types
  - Defined using { } : (and commas).
- You can define, modify, view, lookup, and delete the ***key-value*** pairs in the dictionary.

Using dictionaries

```
d = {'user': 'john', 'pswd': 1234}
print(d['user'])       # john
print(d['pswd'])       # 1234
print(d['john'])       # KeyError: 'john'
d['user']='bill'       # modify user value
print(d)               # {'user': 'bill', 'pswd': 1234}
d['id']=45             # add another key
print(d)               # {'user': 'bill', 'pswd': 1234, 'id': 45}
del d['user']          # remove one key/value pair
print(d)               # {'pswd': 1234, 'id': 45}
                       # d.clear() to remove all key/value pairs
print(d.keys())        # dict_keys(['pswd', 'id'])
print(d.values())      # dict_values([1234, 45])
```

Control of flow 1

- The `if`/`elif`/`else` statement

```
if x == 3:
    print("x equals 3.")
elif x == 2:
    print("x equals 2.")
else:
    print("x equals something else.")
print("This is outside the if statement.")
```

Control of flow 2

- The `while` statement

```
x = 0
while x < 5:
    print(x)
    x = x + 1
print("Outside of the loop.")
```

Output:

0
1
2
3
4
Outside of the loop.

Control of flow 3

- The `for` statement

```
                          # the same as
for i in range(5):        # for i in [0,1,2,3,4]:
    print(i)
print("Outside of the loop.")
```

Output:

0
1
2
3
4
Outside of the loop.

range()

- The range() function has two sets of parameters, as follows:
  - range(stop)
    - stop: Number of integers (whole numbers) to generate, starting from zero. E.g. range(3) == [0, 1, 2].
  - range([start], stop[, step])
    - start: Starting number of the sequence.
    - stop: Generate numbers up to, but not including this number.
    - step: Difference between each number in the sequence.
- Note that:
  - All parameters must be integers.
  - All parameters can be positive or negative.

Control of flow 4

- The `for` statement

```
for i in [3, 6, 9]:
    print(i)
```

Output:

3
6
9

```
for c in "Hello":
    print(c)
```

Output:

H
e
l
l
o

List comprehension

List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = []
for x in fruits:
  if "a" in x:
    newlist.append(x)
print(newlist)
```

With list comprehension you can do all that with only one line of code:

```
fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
newlist = [x for x in fruits if "a" in x]
print(newlist)
```

User-defined functions

- `def` creates a function and assigns it a name
- `return` sends a result back to the caller
- Arguments are passed by assignment
- Arguments and return types are not declared

```
def <name>(arg1, arg2, ..., argN):
    <statements>
    return <values>
```

```
def times(x,y):
    return x*y
```

Function call:

```
x = times(4,5) # returns 20
```

Passing Arguments to Functions

- Arguments are passed by assignment
- Passed arguments are assigned to local names
- There is no call-by-reference per se since:
  - changes to immutable objects within a function only change what object the name points to (and **do not affect the caller**, unless it's a global variable)

- For immutable objects (e.g., integers, strings, tuples), Python acts like C's pass by value
- For mutable objects (e.g., lists), Python acts like C's pass by pointer; in-place changes to mutable objects can affect the caller

Example.py

```
def f1(x,y):
    x = x + 1
    y = y * 2
    print(x, y)       # 1 [1, 2, 1, 2]
def f2(x,y):
    x = x + 1
    y[0] = y[0] * 2
    print(x, y)       # 1 [2, 2]
a = 0                 # immutable
b = [1,2]             # mutable
f1(a,b)
print(a, b)           # 0 [1, 2]
f2(a,b)
print(a, b)           # 0 [2, 2]
```

Optional Arguments

- Can define defaults for arguments that need not be passed

```
def func(a, b, c=10, d=100):
    print(a,b,c,d)

>>> func(1,2)
1 2 10 100
>>> func(1,2,3,4)
1 2 3 4
```

Important notes

- All functions in Python have a return value
  - even if no return line inside the code.
- Functions without a return, return the special value `None`.
- There is <u>no function overloading</u> in Python.
  - Two different functions can’t have the same name, even if they have different arguments.
- Functions can be used as any other data type. They can be:
  - Arguments to function
  - Return values of functions
  - Assigned to variables
  - Parts of tuples, lists, etc

Built-in functions

- https://docs.python.org/3/library/functions.html

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

`len()` :

- Return the length (the number of items) of an object. The argument may be a sequence (such as a string, bytes, tuple, list, or range) or a collection (such as a dictionary, set, or frozen set).

`min()` / `max()` :

- Return the smallest / largest item in an iterable or the smallest of two or more arguments.

Built-in functions: len(), max(), min()

```
>>> my_list = ['one', 'two', 3]
>>> my_list_len = len(my_list)
>>> for i in range(0, my_list_len):
...     print(my_list[i])
...
one
two
3
>>> max("hello","world")
'world'
>>> max(3,13)
13
>>> min([11,5,19,66])
5
```

Modules

- Modules are functions and variables defined in separate files
- Items are imported using from or import

```
from module import function
function()
```

A’ Τρόπος

```
import module
module.function()
```

B’ Τρόπος

Mathematical functions

- https://docs.python.org/3.9/library/math.html

```
>>> import math
>>> print(math.sqrt(3))
1.7320508075688772

>>> from math import sqrt
>>> print(sqrt(3))
1.7320508075688772
```

Lambda function

- Shorthand version of def statement; Useful for “inlining” functions
- A lambda function can take any number of arguments, but can only have one expression (e.g., no if statements, etc)
- A `lambda` returns a function; the programmer can decide whether or not to assign this function to a name
- Simple example:

```
>>> def sum(x,y): return x+y
>>> sum(1,2)
3
>>> sum2 = lambda x, y: x+y
>>> sum2(1,2)
3
```

Built-in functions: map()

- map(func, seq) calls a given function on every element of a sequence and returns an iterator (not a list as in Python 2)
- map.py:

```
def double(x):
    return x*2
a = [1, 2, 3]
print(map(double, a)) # <map object at 0x000001B0512EDD30>
print(list(map(double, a)))                # [2, 4, 6]
```

- Alternatively (without def):

```
a = [1, 2, 3]
print(list(map((lambda x: x*2), a)))  # [2, 4, 6]
```

Built-in functions: map()

- map() can be applied to more than one sequence
- sequences have to have the same length
- map() will apply its lambda function to the elements of the argument sequences, i.e. it first applies to the elements with the 0th index, then to the elements with the 1st index until the n-th index is reached:

```
>>> a = [1,2,3,4]
>>> b = [17,12,11,10]
>>> c = [-1,-4,5,9]
>>> list(map(lambda x,y:x+y, a,b))
[18, 14, 14, 14]
>>> list(map(lambda x,y,z:x+y+z, a,b,c))
[17, 10, 19, 23]
>>> list(map(lambda x,y,z : 2.5*x + 2*y - z, a,b,c))
[37.5, 33.0, 24.5, 21.0]
```

Built-in functions: filter()

- filter(func, seq) filters out all the elements of a sequence for which the function returns True
  - Function has to return a Boolean value
- Example: filter out first the odd and then the even elements of the sequence of the first 11 Fibonacci numbers:

```
>>> fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
>>> odd_numbers = list(filter(lambda x: x % 2, fibonacci))
>>> print(odd_numbers)
[1, 1, 3, 5, 13, 21, 55]
>>> even_numbers = list(filter(lambda x: x % 2 == 0,
fibonacci))
>>> print(even_numbers)
[0, 2, 8, 34]
```

Useful python libraries for data science

- Pandas
  - high-performance, easy-to-use data structures and data analysis tools
  - allows for fast analysis and data cleaning and preparation
  - suited for many different kinds of data: tabular data, time-series data, arbitrary matrix data with row and column labels, and any other form of observational/statistical data sets
- Matplotlib, Seaborn
  - comprehensive library for creating static, animated, and interactive visualizations