Introduction to Python programming

July 1, 2022

1 What is Python ?

What is Python?

In 1989 the dutch computer scientist, Guido van Rossum, created PYTHON .

- The name of this language comes from the TV-series "Monty Python's Flying Circus" of which G. van Rossum is a big fan.
- The first version was published in 1991.
- The most recent version is 3.8 that was released in 14/10/2019.
- *Python Software Foundation* is the association that organizes the developement of this language and manages the community of developers and users.

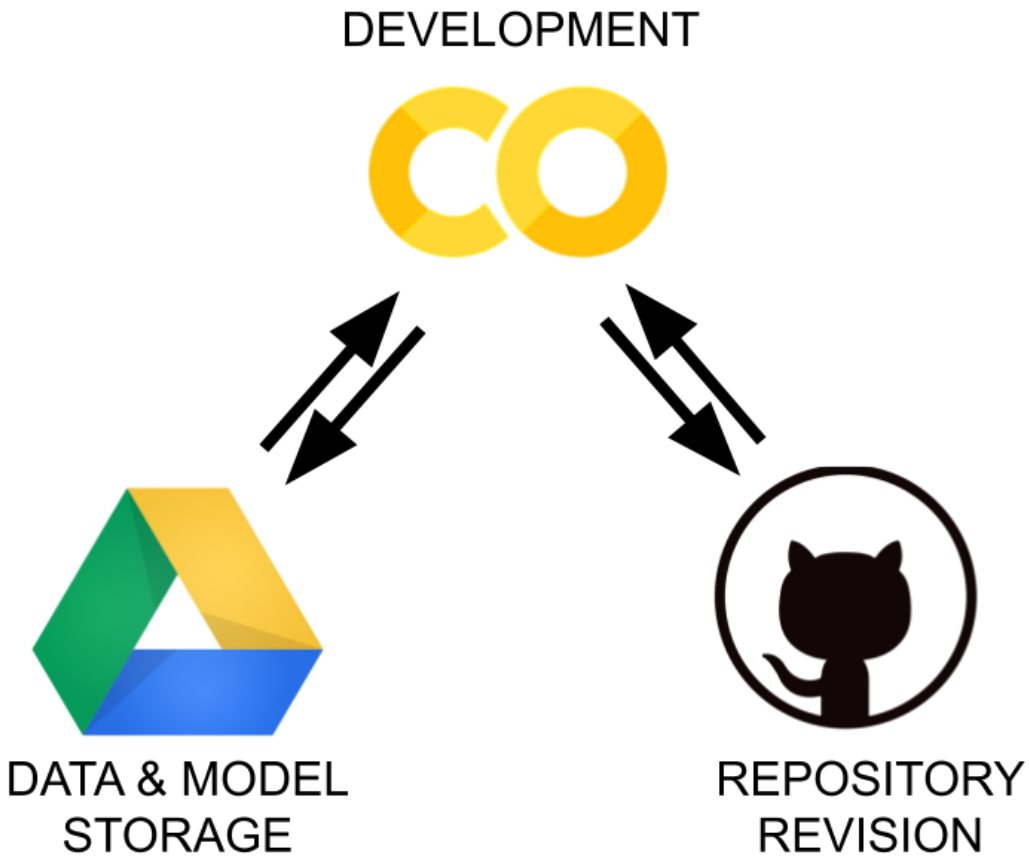
This programming language presents several interesting caracteristics:

1. It is multiplateform. This means that it can be used on several operating systems: Windows, Mac OS X, Linux, Android, iOS, starting from the Mini-computers Raspberry Pi to the latest super calculators.
2. It is free. It can be installed over any PC that you want and even on your phones.
3. It is a high level language. It does NOT require the user to have a big knowledge in the functionning of the PC.
4. It is an interpretated language. This means that a Python script doesn't need to be compiled to be executed. Contrarily to other languages like C and C++.
5. It is object oriented. This means that it is possible to see in Python some entities that mime the real world (a cell, a protein, an atom, etc.) with a certain number of functionning rules and interactions.
6. It is relatively simple to work with it.
7. It is used in several domains, like bioinformatics, data analysis etc..

All theses caracteristics make of PYTHON a very useful language.
That's why, nowadays it is used in high schools and higher education levels.

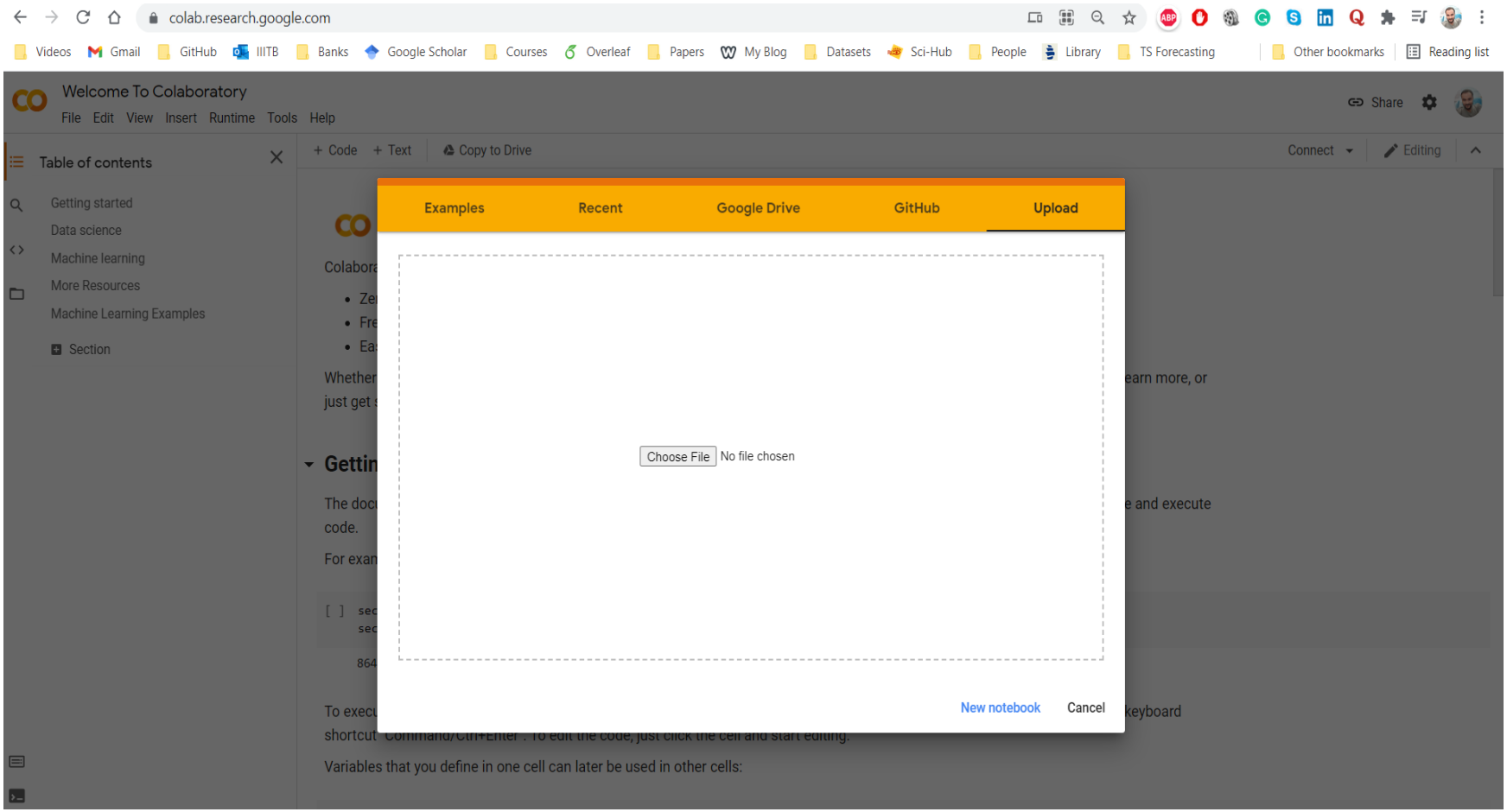
2 What is Google Colab

GOOGLE COLAB



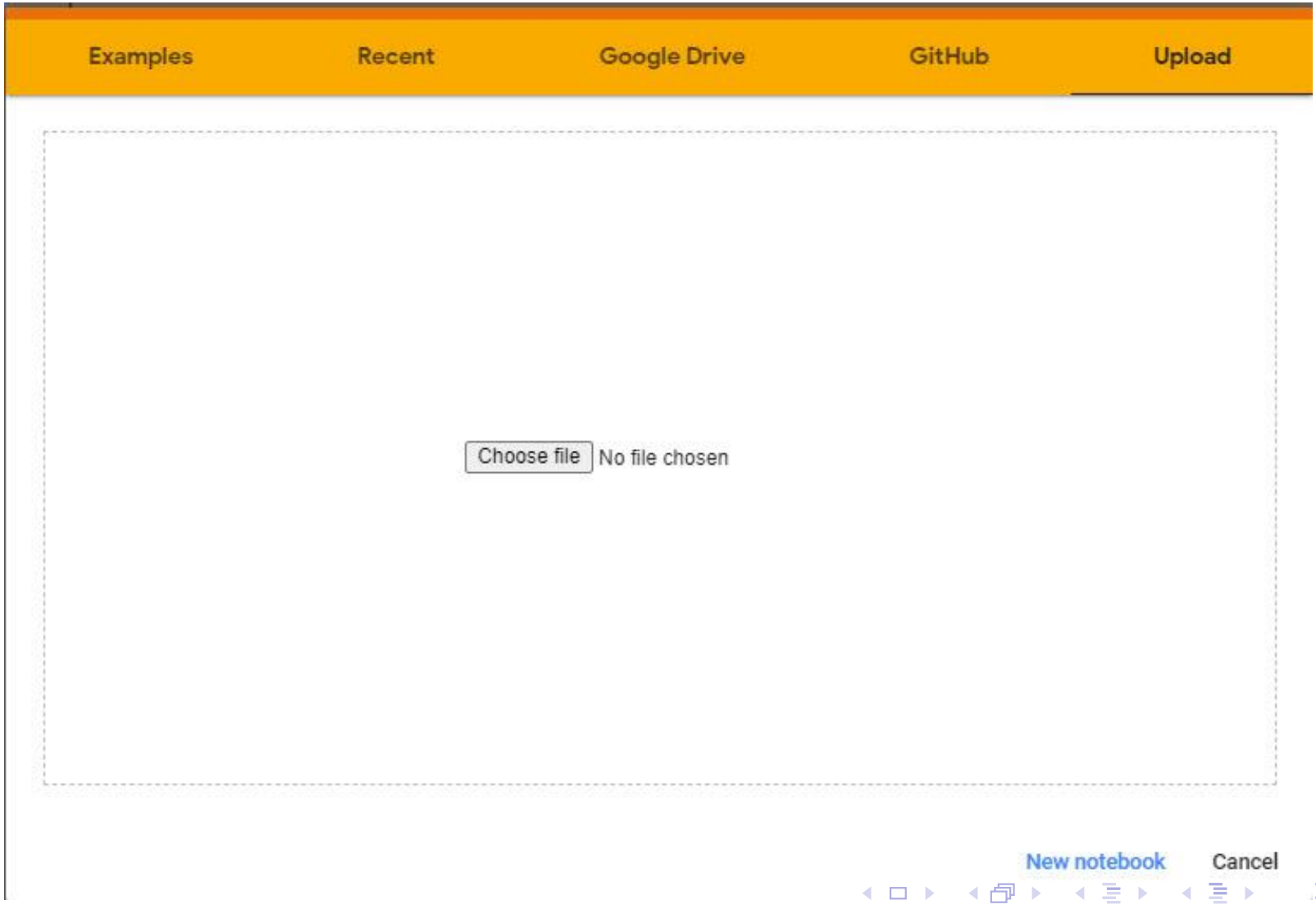
GOOGLE COLAB

Open Google Colab: https://colab.research.google.com/



GOOGLE COLAB

Upload the ipynb / py file or start with a new notebook



3 Basic Arithmetic Operations

Numerical operations

Using Python shell as a calculator : introducing in Python basic mathematical operations.

1. **A-S** Addiction $a + b$ and Subtraction $a - b$;
2. **M-D** Multiplication $a * b$ and Division a/b;
3. **E** Exponential $a * *b$;
4. **P** Parenthesis(...), are used to force the order of operations;

Left-to-right order: remember the acronymous "PEMDAS":

$$P > E > M \sim D > A \sim S.$$

5- **F** Floor division $a//b$ $(= \lfloor a/b \rfloor, integerpart)$;

5- **R** Rest $a\%b$ $(= a - b * a//b)$.

Left-to-right order: $F \sim R \sim M \sim D$.

Example

Predict the answer in Python: $2+3$; $2*(3-1)$; $2*3-1$; $(1+1)**(5-2)$; $3*1**3$; $2/3$; $2/0$; $(3*1)**3$; $4//3$; $2//3$; $17\%3$ and $15\%4$.

```
>>> 2+3
5
```

...

```
>>> 2*3-1
5
```

```
>>> 2*(3-1)
4
```

...

```
>>> (1+1)**(5-2)
8
```

```
>>> 2/3
0.6666666666666666
```

..

```
>>> 3*1**3
3
```

...

```
>>> (3*1)**3
27
```

```
>>> 4//3
1
```

...

```
>>> 2//3
0
```

```
>>> 2/0
Traceback (most recent call last):
File "<pyshell#3>", line 1, in <module>
2/0
ZeroDivisionError:  int division or modu
```

```
>>> 17%3
2
```

...

```
>>> 15%4
3
```

Operations on Strings

1. $*$ is used for the **repetition**;
2. $+$ is used for the **concatenation**;

```
>>> 'hello'*3
'hellohellohello'
```

...

```
>>> 3*'hello'
'hellohellohello'
```

...

```
>>> 'hello' + 'world'
'helloworld'
```

3. # is used to **insert comments** in the text.

```
>>> 'hello' + 'world' # concatenation of strings
'helloworld'
```

4 Data Structures

4.1 Strings and the `print()` built-in function

Strings

1. Strings are sequences of character data. In Python, it is called str.
2. String literals are delimited using either single quote ' ' or double quotes " ". All the characters between the opening and closing delimiter are part of the string:

Strings

1. Strings are sequences of character data. In Python, it is called str.
2. String literals are delimited using either single quote ' ' or double quotes " ". All the characters between the opening and closing delimiter are part of the string:

Python

```
>>> print("I am a string.")
I am a string.
>>> type("I am a string.")
<class 'str'>

>>> print('I am too.')
I am too.
>>> type('I am too.')
<class 'str'>
```

Strings

1. Strings are sequences of character data. In Python, it is called str.
2. String literals are delimited using either single quote ' ' or double quotes " ". All the characters between the opening and closing delimiter are part of the string:
3. A string in Python can contain as many characters as you wish. It can also be empty:

Strings

1. Strings are sequences of character data. In Python, it is called str.
2. String literals are delimited using either single quote ' ' or double quotes " ". All the characters between the opening and closing delimiter are part of the string:
3. A string in Python can contain as many characters as you wish. It can also be empty:

```
Python
>>> ''
''
```

Strings

1. Strings are sequences of character data. In Python, it is called str.
2. String literals are delimited using either single quote ' ' or double quotes " ". All the characters between the opening and closing delimiter are part of the string:
3. A string in Python can contain as many characters as you wish. It can also be empty:
4. If you want to include a quote character as part of the string itself, you get:

Strings

1. Strings are sequences of character data. In Python, it is called str.
2. String literals are delimited using either single quote ' ' or double quotes " ". All the characters between the opening and closing delimiter are part of the string:
3. A string in Python can contain as many characters as you wish. It can also be empty:
4. If you want to include a quote character as part of the string itself, you get:

Python

```
>>> print('This string contains a single quote (') character.')
SyntaxError: invalid syntax
```

Strings

1. Strings are sequences of character data. In Python, it is called str.
2. String literals are delimited using either single quote ' ' or double quotes " ". All the characters between the opening and closing delimiter are part of the string:
3. A string in Python can contain as many characters as you wish. It can also be empty:
4. If you want to include a quote character as part of the string itself, you get:
   To Solve:

Python

```
>>> print("This string contains a single quote (') character.")
This string contains a single quote (') character.

>>> print('This string contains a double quote (") character.')
This string contains a double quote (") character.
```

Strings

1. Strings are sequences of character data. In Python, it is called str.
2. String literals are delimited using either single quote ' ' or double quotes " ". All the characters between the opening and closing delimiter are part of the string:
3. A string in Python can contain as many characters as you wish. It can also be empty:
4. If you want to include a quote character as part of the string itself, you get:
   To Solve:
   OR

Python

```
>>> print('This string contains a single quote (\') character.')
This string contains a single quote (') character.
```

Strings

1. Strings are sequences of character data. In Python, it is called str.
2. String literals are delimited using either single quote ' ' or double quotes " ". All the characters between the opening and closing delimiter are part of the string:
3. A string in Python can contain as many characters as you wish. It can also be empty:
4. If you want to include a quote character as part of the string itself, you get:
   To Solve:
   OR

Python

```
>>> print("This string contains a double quote (\") character.")
This string contains a double quote (") character.
```

Escape sequences

Escape Sequence	Usual Interpretation of Character(s) After Backslash	"Escaped" Interpretation
`\'`	Terminates string with single quote opening delimiter	Literal single quote (') character
`\"`	Terminates string with double quote opening delimiter	Literal double quote (") character
`\newline`	Terminates input line	Newline is ignored
`\\`	Introduces escape sequence	Literal backslash (\) character

Print()-function

- Pressing **Enter** in the middle of a string will cause Python to think it is incomplete:

```
Python
>>> print('a

SyntaxError: EOL while scanning string literal
```

Print()-function

- Pressing **Enter** in the middle of a string will cause Python to think it is incomplete:
- To break up a string over more than one line:

Python

```
>>> print('a\
... b\
... c')
abc
```

Print()-function

- Pressing **Enter** in the middle of a string will cause Python to think it is incomplete:
- To break up a string over more than one line:
- To include a literal backslash in a string:

Python

```
>>> print('foo\\bar')
foo\bar
```

Print()-function

- Pressing **Enter** in the middle of a string will cause Python to think it is incomplete:
- To break up a string over more than one line:
- To include a literal backslash in a string:
- A tab character can be specified by the escape sequence "$\backslash t$"

Python

```
>>> print('foo\tbar')
foo     bar
```

Print()-function

- Pressing **Enter** in the middle of a string will cause Python to think it is incomplete:
- To break up a string over more than one line:
- To include a literal backslash in a string:
- A tab character can be specified by the escape sequence "\t"
- Some examples

Python

```
>>> print("a\tb")
a    b
>>> print("a\141\x61")
aaa
>>> print("a\nb")
a
b
>>> print('\u2192 \N{rightwards arrow}')
→ →
```

Print()-function

- Pressing **Enter** in the middle of a string will cause Python to think it is incomplete:
- To break up a string over more than one line:
- To include a literal backslash in a string:
- A tab character can be specified by the escape sequence "$\backslash t$"
- Some examples
- A **raw string** literal is preceded by r or R , which specifies that escape sequences in the associated string are not translated.

Python

```
>>> print('foo\nbar')
foo
bar
>>> print(r'foo\nbar')
foo\nbar

>>> print('foo\\bar')
foo\bar
>>> print(R'foo\\bar')
foo\\bar
```

Print()-function

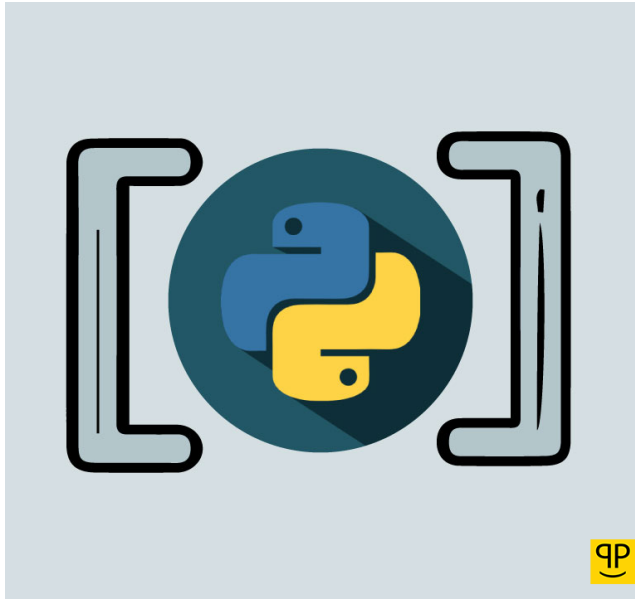
- Pressing **Enter** in the middle of a string will cause Python to think it is incomplete:
- To break up a string over more than one line:
- To include a literal backslash in a string:
- A tab character can be specified by the escape sequence "$\backslash t$"
- Some examples
- A **raw string** literal is preceded by r or R , which specifies that escape sequences in the associated string are not translated.
- Triple-Quoted Strings provides a convenient way to create a string with both single and double quotes in it.

Python

```
>>> print('''This string has a single (') and a double (") quote.''')
This string has a single (') and a double (") quote.
```

4.2 Lists

Lists in Python



1. A list is a **data structure** in Python that is a **mutable**, or **changeable, ordered sequence** of elements.
2. Each element or value that is inside of a list is called an **item**.
3. Just as strings are defined as characters between quotes, lists are defined by having values between square brackets [ ].

4. Lists are great to use when you want to work with many **related values**.

5. They enable you to keep data together that belongs together, condense your code, and perform the same methods and operations on multiple values at once.

6. When thinking about Python lists and other data structures that are types of collections, it is useful to consider all the different collections you have on your computer: your assortment of files, your song playlists, your browser bookmarks, your emails, the collection of videos you can access on a streaming service, and more.

To get started, let's create a list that contains items of the string data type:

```
sea_creatures = ['shark', 'cuttlefish', 'squid', 'mantis shrimp', 'anemone']
```

When we print out the list, the output looks exactly like the list we created:

```
print(sea_creatures)
```

```
Output
['shark', 'cuttlefish', 'squid', 'mantis shrimp', 'anemone']
```

As an ordered sequence of elements, each item in a list can be called individually, through **indexing**.

Indexing lists

Each item in a list corresponds to an index number, which is an integer value, starting with the index number 0.

'shark'	'cuttlefish'	'squid'	'mantis shrimp'	'anemone'
0	1	2	3	4

Because each item in a Python list has a corresponding index number, we're able to access and manipulate lists in the same ways we can with other sequential data types.

```
sea_creatures[0] = 'shark'
sea_creatures[1] = 'cuttlefish'
sea_creatures[2] = 'squid'
sea_creatures[3] = 'mantis shrimp'
sea_creatures[4] = 'anemone'
```

Negative indexing

NB: If we call the list with an index numberthat is greater than 4, it will be out of range as it will not be valid:

```
print(sea_creatures[18])
```

```
Output

IndexError: list index out of range
```

In addition to positive index numbers, we can also access items from the list with a negative index number, by counting backwards from the end of the list, starting at -1.

'shark'	'cuttlefish'	'squid'	'mantis shrimp'	'anemone'
-5	-4	-3	-2	-1

Negative indexing

NB: If we call the list with an index numberthat is greater than 4, it will be out of range as it will not be valid:

```
print(sea_creatures[18])
```

```
Output

IndexError: list index out of range
```

In addition to positive index numbers, we can also access items from the list with a negative index number, by counting backwards from the end of the list, starting at -1.

```
print(sea_creatures[-3])
```

```
Output

squid
```

Modifying Lists

If we want to change the string value of the item at index 1 from 'cuttlefish' to 'octopus', we can do so like this:

```
sea_creatures[1] = 'octopus'
```

Now when we print `sea_creatures`, the list will be different:

```
print(sea_creatures)
```

```
Output
['shark', 'octopus', 'squid', 'mantis shrimp', 'anemone']
```

Modifying Lists

If we want to change the string value of the item at index 1 from 'cuttlefish' to 'octopus', we can do so like this:
We can also change the value of an item by using a **negative index number** instead:

4.2.1 Operations on Lists

Operations on Lists

1) **Adding two lists using + to concatenate them:**
$l_1 + l_2 = l_3$

```
sea_creatures = ['shark', 'octopus', 'blobfish', 'mantis shrimp', 'anemone']
oceans = ['Pacific', 'Atlantic', 'Indian', 'Southern', 'Arctic']

print(sea_creatures + oceans)
```

```
['shark', 'octopus', 'blobfish', 'mantis shrimp', 'anemone', 'Pacific', 'Atlantic', 'Indian', 'Southern', 'Arctic']
```

Operations on Lists

1) **Adding two lists using $+$ to concatenate them:**
2) **Extending a list with another list using** *extend*():
$l_1.extend(l_2) = l_1;\ l_1 = l_1 + l_2$

```
In [9]: l1=[1,2,3,4,5,6]
        l2=[11,12]

In [10]: l1.extend(l2)

In [11]: l1
Out[11]: [1, 2, 3, 4, 5, 6, 11, 12]
```

Operations on Lists

1) **Adding two lists using $+$ to concatenate them:**
2) **Extending a list with another list using** *extend*():
3) **Adding an item to the list using** *append*():

$l_1.append(a) = l_1$;

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
```

Deletion

We must distinguish between two ideas, revoming and deleting :

Deletion

We must distinguish between two ideas, revoming and deleting :
A specific item *a* can be removed from a list "*l*" as follows:

Deletion

We must distinguish between two ideas, revoming and deleting :
A specific item *a* can be removed from a list "*l*" as follows:

```
In [12]: l1

Out[12]: [1, 2, 3, 4, 5, 6, 11, 12]

In [13]: l1.remove(5)

In [14]: l1

Out[14]: [1, 2, 3, 4, 6, 11, 12]
```

Deletion

We must distinguish between two ideas, revoming and deleting :
A specific item *a* can be removed from a list "*l*" as follows:
An item at position *n* can be deleted from a list "*l*" as follows:

Deletion

We must distinguish between two ideas, revoming and deleting :
A specific item *a* can be removed from a list "*l*" as follows:
An item at position *n* can be deleted from a list "*l*" as follows:

```
# list of numbers
n_list = [1, 2, 3, 4, 5, 6]

# Deleting 2nd element
del n_list[1]
```

Deletion

We must distinguish between two ideas, revoming and deleting :
A specific item *a* can be removed from a list "*l*" as follows:
An item at position *n* can be deleted from a list "*l*" as follows:
We can remove all the items from the list by using:

Deletion

We must distinguish between two ideas, revoming and deleting :
A specific item *a* can be removed from a list "*l*" as follows:
An item at position *n* can be deleted from a list "*l*" as follows:
We can remove all the items from the list by using:

```
In [16]: l1
Out[16]: [1, 2, 3, 6, 11, 12]

In [17]: l1.clear()

In [18]: l1
Out[18]: []
```

Deletion

We must distinguish between two ideas, revoming and deleting :
A specific item *a* can be removed from a list "*l*" as follows:
An item at position *n* can be deleted from a list "*l*" as follows:
We can remove all the items from the list by using:We can POP the last element of the list as follows:

Deletion

We must distinguish between two ideas, revoming and deleting :
A specific item *a* can be removed from a list "*l*" as follows:
An item at position *n* can be deleted from a list "*l*" as follows:
We can remove all the items from the list by using:We can POP the last element of the list as follows:

```
In [19]: l1=[1, 2, 3, 4, 5, 6, 11, 12]

In [20]: l1.pop()
Out[20]: 12
```

More operations

We can ask for the length of a list *l* by using:

More operations

We can ask for the length of a list *l* by using:

```
In [24]: l1

Out[24]: [1, 2, 3, 4, 5, 6, 11]

In [25]: len(l1)

Out[25]: 7
```

More operations

We can ask for the length of a list *l* by using:
We can count the occurences of a certain element *a* in *l* by using:

More operations

We can ask for the length of a list *l* by using:
We can count the occurences of a certain element *a* in *l* by using:

```
In [29]: l2

Out[29]: [3, 2, 4, 3, 3, 2, 4, 5]

In [30]: l2.count(3)

Out[30]: 3
```

More operations

We can ask for the length of a list *l* by using:
We can count the occurences of a certain element *a* in *l* by using:
We can ask for the index of an item in a list *l* by using:

More operations

We can ask for the length of a list *l* by using:
We can count the occurences of a certain element *a* in *l* by using:
We can ask for the index of an item in a list *l* by using:

```
In [32]: l1

Out[32]: [1, 2, 3, 4, 5, 6, 11]

In [33]: l1.index(3)

Out[33]: 2
```

More operations

We can ask for the length of a list *l* by using:
We can count the occurences of a certain element *a* in *l* by using:
We can ask for the index of an item in a list *l* by using:
Note that: If a list contains the same element several times, we can ask for the index starting a certain position :

More operations

We can ask for the length of a list *l* by using:
We can count the occurences of a certain element *a* in *l* by using:
We can ask for the index of an item in a list *l* by using:
Note that: If a list contains the same element several times, we can ask for the index starting a certain position :

```
In [50]: l2=[3,2,4,3,3,2,4,5]

In [51]: l2.index(3,2)
Out[51]: 3
```

4.3 Dictionaries

Mutable VS. Immutable

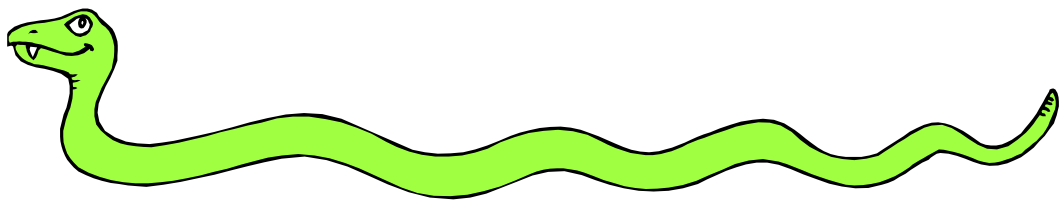
[plain]

- a mutable class/type is a class/type whose instances (objects/variables) can be modified once created eg. `list`, `dict`, `set`
- a immutable class/type is a class/type whose instances cannot be modified once created eg. `tuple`, `int`, `float`, `bool`

```
In [1]: L = [0,1]
        L[1] = 2
        print(L)

[0, 2]
```

```
In [2]: t = (0,1)
        t[1] = 2

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [2], in <cell line: 2>()
      1 t = (0,1)
----> 2 t[1] = 2

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

Dictionaries: a *mapping* collection type



Dictionaries: Like *maps* in Java

- **Dictionaries store a *mapping* between a set of keys and a set of values.**
  - ***Keys can be any immutable type.***
  - Values can be any type
  - Values and keys can be of different types in a single dictionary
- **You can**
  - define
  - modify
  - view
  - lookup
  - delete

  the key-value pairs in the dictionary.

Creating and accessing dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user']
'bozo'

>>> d['pswd']
1234

>>> d['bozo']

Traceback (innermost last):
  File '<interactive input>' line 1, in ?
KeyError: bozo
```

Updating Dictionaries

```
>>> d = {'user':'bozo', 'pswd':1234}

>>> d['user'] = 'clown'
>>> d
{'user':'clown', 'pswd':1234}
```

- Keys must be unique.
- Assigning to an existing key replaces its value.

```
>>> d['id'] = 45
>>> d
{'user':'clown', 'id':45, 'pswd':1234}
```

- Dictionaries are unordered
  - New entry might appear anywhere in the output.
- (Dictionaries work by *hashing*)

Removing dictionary entries

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> del d['user']                  # Remove one. Note that del is
                                   # a function.
>>> d
{'p':1234, 'i':34}

>>> d.clear()                      # Remove all.
>>> d
{}

>>> a=[1,2]
>>> del a[1]                     # (del also works on lists)
>>> a
[1]
```

Useful Accessor Methods

```
>>> d = {'user':'bozo', 'p':1234, 'i':34}

>>> d.keys()                  # List of current keys
['user', 'p', 'i']

>>> d.values()                # List of current values.
['bozo', 1234, 34]

>>> d.items()          # List of item tuples.
[('user','bozo'), ('p',1234), ('i',34)]
```

5 Loops and Comparison

5.1 for loop

Loop FOR

In programming, we mostly need to repeat an instruction several times. That's why we need LOOPS.

```
In [40]: M=['lion','tigre', 'monkey', 'zebra', 'bear', 'snake']

In [43]: print(M[0])
lion

In [44]: print(M[1])
tigre

In [45]: print(M[2])
monkey

In [46]: print(M[3])
zebra

In [47]: for i in M:
             print(i)
lion
tigre
monkey
zebra
bear
snake
```

- The letter *i* in the loop for is called ITERATION variable that changes its value at each iteration of the loop.
- It is a dummy variable , which means can be replaced by anything else.
- The iteration variable can be of any type depending on the list.
- We caracterize it by the <:> at the end of the line. This means that the loop for is waiting for a BLOC of instructions.
- This bloc is considered as the body of the loop.
- In order to know where the bloc starts or ends , we use the identation which is made of the 4 spaces (1 TAB) with respect to the position of the word FOR.,

Example

In [49]:
```
for animal in M :
    print ( animal )
    print ( animal *2)
print (" it is over ")
```

```
..

lion
lionlion
tigre
tigretigre
monkey
monkeymonkey
zebra
zebrazebra
bear
bearbear
snake
snakesnake
 it is over
```

The iteration can be over different types of lists and sub-lists:

```
In [50]: for animal in M[1:3]:
             print(animal)

tigre
monkey
```

```
In [51]: for i in [1,2,3]:
             print(i)

1
2
3
```

```
In [52]: for i in range(4):
             print(i)

0
1
2
3
```

```
In [53]: for i in range(3):
             print(M[i])

lion
tigre
monkey
```

```
In [56]: for i in range ( len ( M )):
             print (" The animal {} is a {}". format (i , M [ i ]))

 The animal 0 is a lion
 The animal 1 is a tigre
 The animal 2 is a monkey
 The animal 3 is a zebra
 The animal 4 is a bear
 The animal 5 is a snake
```

5.2 while loop and comparison

Loop WHILE

While-loops are **loops** that repeat while/until a specific condition is met.
They can replace for-loops for quicker computation in a lot of cases.
Additionally, they do not require a **parameter** for the amount of repetitions.
While-loops simply check the condition statement at each repetition of the loop.
If the conditional statement is not met, the loop breaks.
The syntax is: while condition :

```
>>> l = []
>>> x = 0
>>> while x < 100:
        l.append(x)
        x += 1
>>> print(l)

[1,2,3,4,5, ...  51,52,53, ...  97, 98, 99]
```

```
>>> l = []
>>> x = 0
>>> while x < 100:
        l.append(x)
        x += 1
>>> print(l)

[1,2,3,4,5, ...  51,52,53, ...  97, 98, 99]
```

Note

If the fifth line of $x+=1$ hadn't been added, the loop would continue for ever, as the loop would check $0 < 100$ and then afterwards the loop would append(x) to l.

Therefore, it is mandatory to think ahead and ensure that the condition that is being checked is **continuously changing**, otherwise, the loop will never break.

6 Three essential packages for scientific computing in Python

6.1 NumPy

What is NumPy

- NumPy stands for Numerical Python
- It is the most widely used library for working vectors, matrices and more generally n-dimensional arrays.
- *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. At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance.*[1]

[1] https://numpy.org/devdocs/user/whatisnumpy.html

Today we will use NumPy to:

- create NumPy arrays of zeros, ones and from lists of values.
- reshape and concatenate those arrays
- perform mathematical operations on those arrays

Instantiate NumPy arrays

```
In [7]: np.ones((3,3))

Out[7]: array([[1., 1., 1.],
               [1., 1., 1.],
               [1., 1., 1.]])

In [8]: np.zeros((3,3))

Out[8]: array([[0., 0., 0.],
               [0., 0., 0.],
               [0., 0., 0.]])

In [19]: np.identity(3)

Out[19]: array([[1., 0., 0.],
                [0., 1., 0.],
                [0., 0., 1.]])
```

Instantiate NumPy arrays and reshape

Create a 3x3 array with integer entries ranging from 0 to 8.

- from correctly arranged list of lists of values (each entry of the first list is a list containing a row of the output matrix):

```
In [15]: np.array([[0, 1, 2],[3, 4, 5], [6, 7, 8]])

Out[15]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])
```

- using `np.arange` and `np.reshape`

```
In [17]: np.arange(9)

Out[17]: array([0, 1, 2, 3, 4, 5, 6, 7, 8])

In [18]: np.arange(9).reshape((3,3))

Out[18]: array([[0, 1, 2],
                [3, 4, 5],
                [6, 7, 8]])
```

the shape of NumPy array

The shape of numpy array is a tuple containing the length of the array along each of its axes.
It can be used at the instantiation step to specify the required shape of the array (as was done in previous slides).
It can be accessed for an existing array by accessing the shape attribute of the array:

```
In [10]: A = np.ones((3,3))
         A.shape

Out[10]: (3, 3)
```

Concatenate numpy arrays

```
In [22]: np.vstack([np.zeros((3,3)), np.ones((3,3))])
```

```
Out[22]: array([[0., 0., 0.],
                [0., 0., 0.],
                [0., 0., 0.],
                [1., 1., 1.],
                [1., 1., 1.],
                [1., 1., 1.]])
```

```
In [24]: np.hstack([np.zeros((3,3)), np.ones((3,3))])
```

```
Out[24]: array([[0., 0., 0., 1., 1., 1.],
                [0., 0., 0., 1., 1., 1.],
                [0., 0., 0., 1., 1., 1.]])
```

```
In [42]: np.concatenate([np.zeros((3,3)), np.ones((3,3))], axis=1)
```

```
Out[42]: array([[0., 0., 0., 1., 1., 1.],
                [0., 0., 0., 1., 1., 1.],
                [0., 0., 0., 1., 1., 1.]])
```

pointwise operations on NumPy arrays

Pointwise operations are operations that are applied to each entry of the array separately as opposed to matrix operations that can be applied to 2-dimensional arrays and will be tackled later this week.
examples:`np.power`, `np.exp`, `np.sqrt`

```
In [34]: np.sqrt(4*np.ones((3,3)))

Out[34]: array([[2., 2., 2.],
                [2., 2., 2.],
                [2., 2., 2.]])
```

matrix multiplication

Matrix multiplication, the simplest matrix operation can be computed in two different ways:

```
In [40]: A = 2*np.ones((3,3))
         A @ A

Out[40]: array([[12., 12., 12.],
                [12., 12., 12.],
                [12., 12., 12.]])

In [41]: np.matmul(A,A)

Out[41]: array([[12., 12., 12.],
                [12., 12., 12.],
                [12., 12., 12.]])
```

np.linspace

This function allows to dicretize a given interval by outputting an array of evenly spaced points covering the interval. This is useful for plotting a function.

```
In [45]: np.linspace(0,1,11)
Out[45]: array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ])
```

6.2 Matplotlib

Matplotlib.pyplot

- Matplotlib is an enormous module that has functions for plotting data in $2D$ and $3D$.
- Some of the functions include plot(input, output), ylabel(''), title('').
- The syntax and arguments in matplotlib can seem daunting, however, they are in fact very simple with a slight amount of practice.

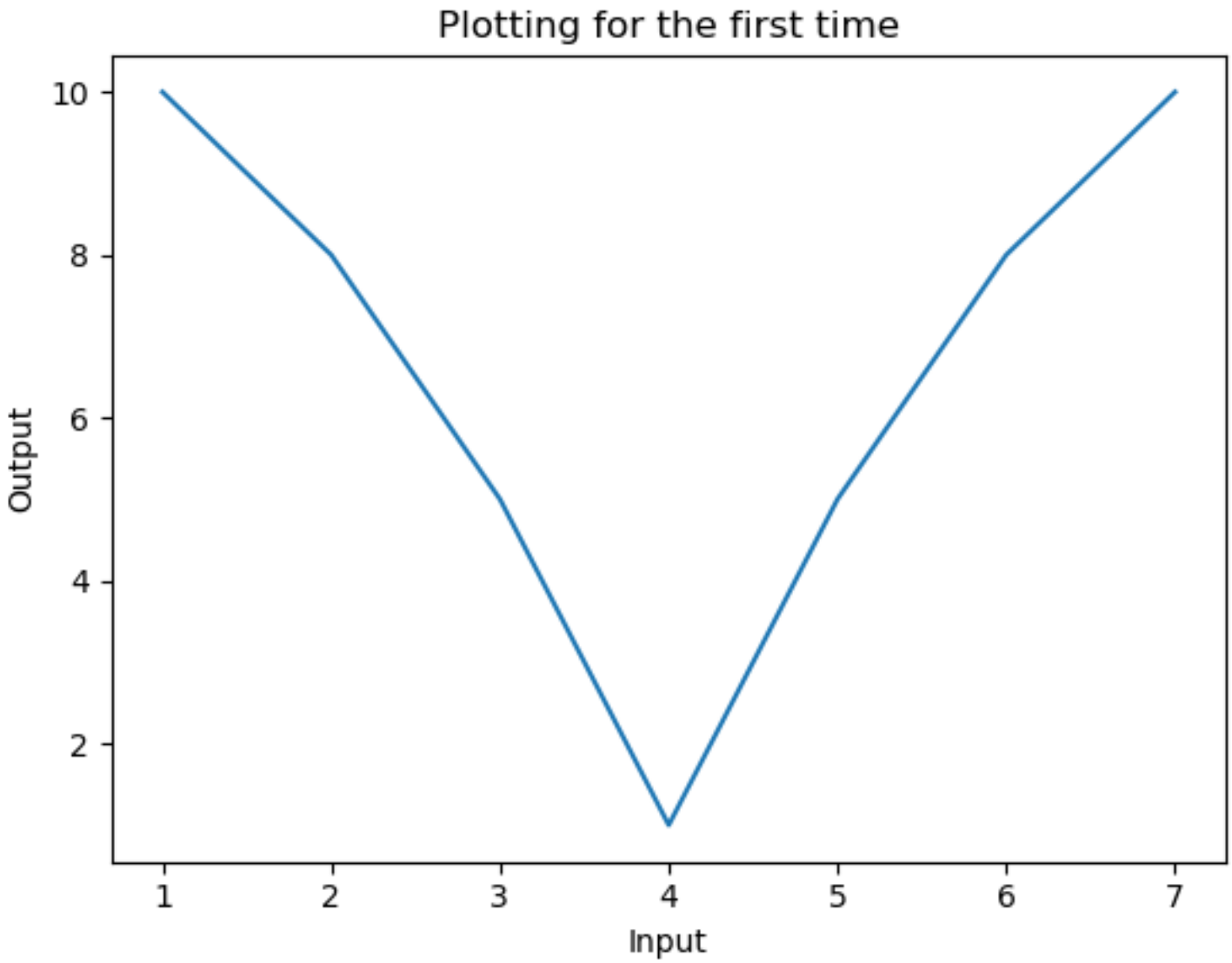
Example

1. import matplotlib.pyplot as plt
   plt.plot([1, 2, 3, 4])
   plt.ylabel('some numbers')
   plt.show()

2. plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
3. plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
   plt.axis([0, 6, 0, 20])
   plt.show()
4.

```
import numpy as np

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3, 'g^')
plt.show()
```

```
In [2]: import matplotlib.pyplot as plt
   ...: x=[1,2,3,4,5,6,7]
   ...: y=[10,8,5,1,5,8,10]
   ...: plt.title('Plotting for the first time')
   ...: plt.ylabel('Output')
   ...: plt.xlabel('Input')
   ...: plt.plot(x,y)
   ...:
Out[2]: [<matplotlib.lines.Line2D at 0x7f96086233d0>]
```


Plotting for the first time
Output
Input
10
8
6
4
2
1
2
3
4
5
6
7

More details

The full syntax for this module command is as follows:

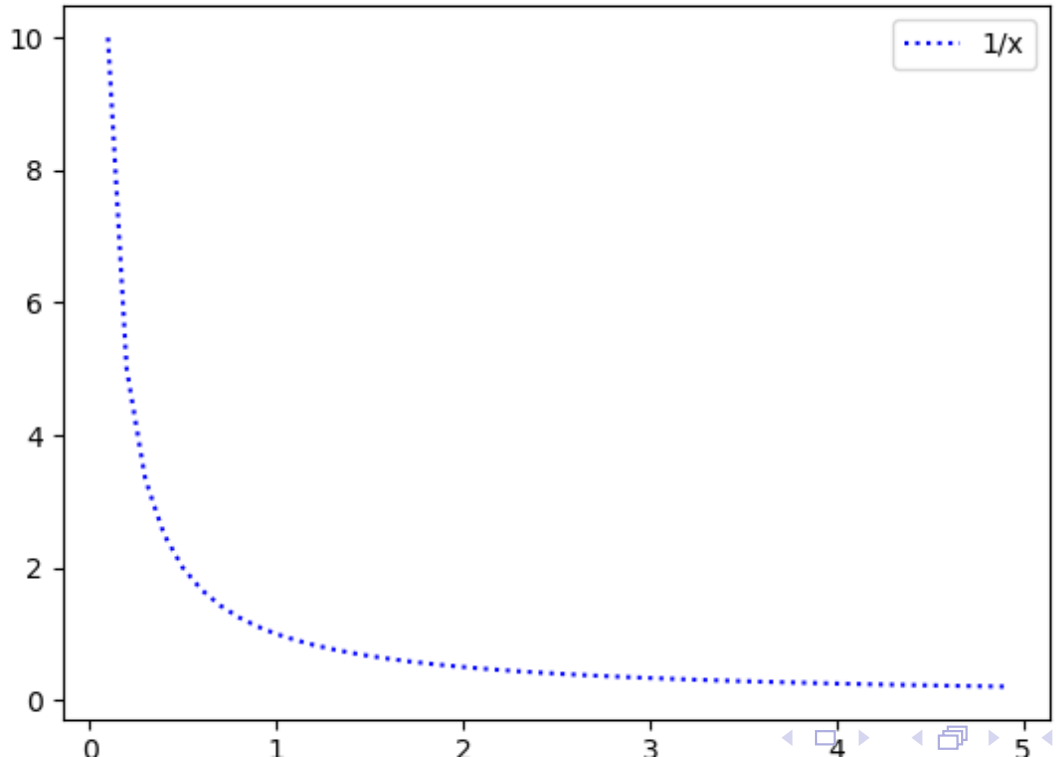
$$plt.plot(input, output, type, label = label).$$

The syntax is **trivial**, except for 'type'.
The list below shows the possible inputs and the function of the input.

1. 'r' makes the color red.
2. 'b' makes the color blue .
3. 'k' makes the graph **black**.
4. ':' makes the line dotted.
5. 'o' makes the graph a scatter plot with circles.
6. '+' makes the graph a scatter plot with '+' as points.
7. 'k:' makes the graph a black dotted line.
8. '.r' makes the graph a scatter plot with small red points.

Note that:This list is not a full list of all of the possible commands.
The arguments inside the string can be concatenated to yield a graph with all of the parameters requested as in 6 and 7.

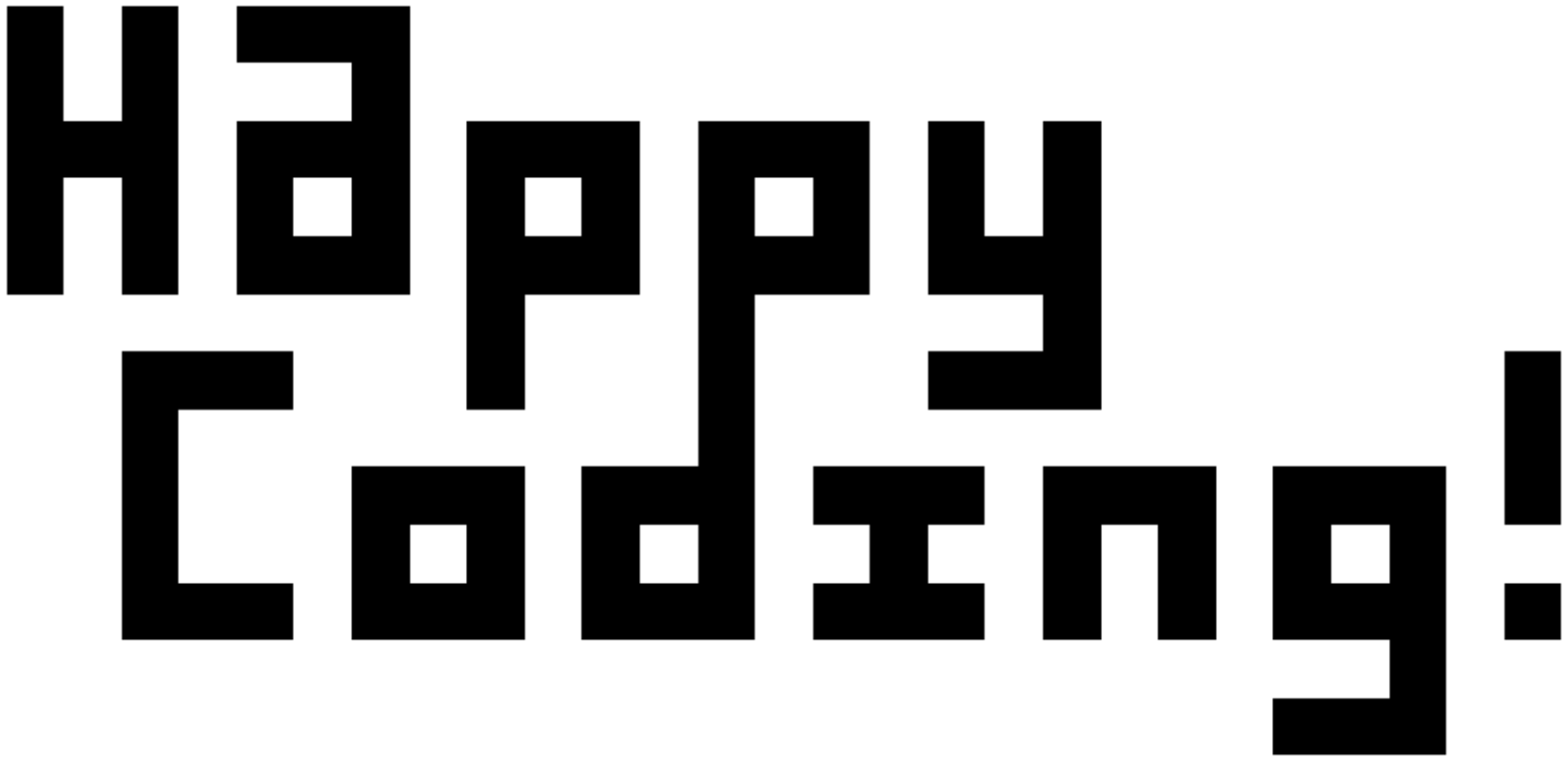
In [7]:
```
import matplotlib.pyplot as plt
import numpy as np
def f(x):
    return 1/x
x= np.arange(0.1, 5, 0.1)
plt.plot(x, f(x), 'b:',label = '1/x $')
plt.legend()
```



Some modifications

In [15]:
```
import matplotlib.pyplot as plt
import numpy as np
def f(x):
    return 1/x
x= np.arange(0.1, 5, 0.1)
plt.title('This is a new graph')
plt.ylabel('Branch of Hyperbola')
plt.xlabel('X')
plt.plot(x, f(x), 'r+',label = '1/x')
plt.legend()
```

6.3 Pandas


Happy
Coding!