Python Basics

Index

INTRODUCTION, DATA TYPES,
COMMON IDE AND DEVELOPING TOOLS , CONDITIONAL AND LOOPS,
SAVE A SCRIPT,
JUPYTER NOTEBOOK,
FUNCTIONS, MODULES, FILE I/O

Introduction

What is and why Python?

1. Python is an **interpreted**, **multi-paradigm**, **high-level** programming language.

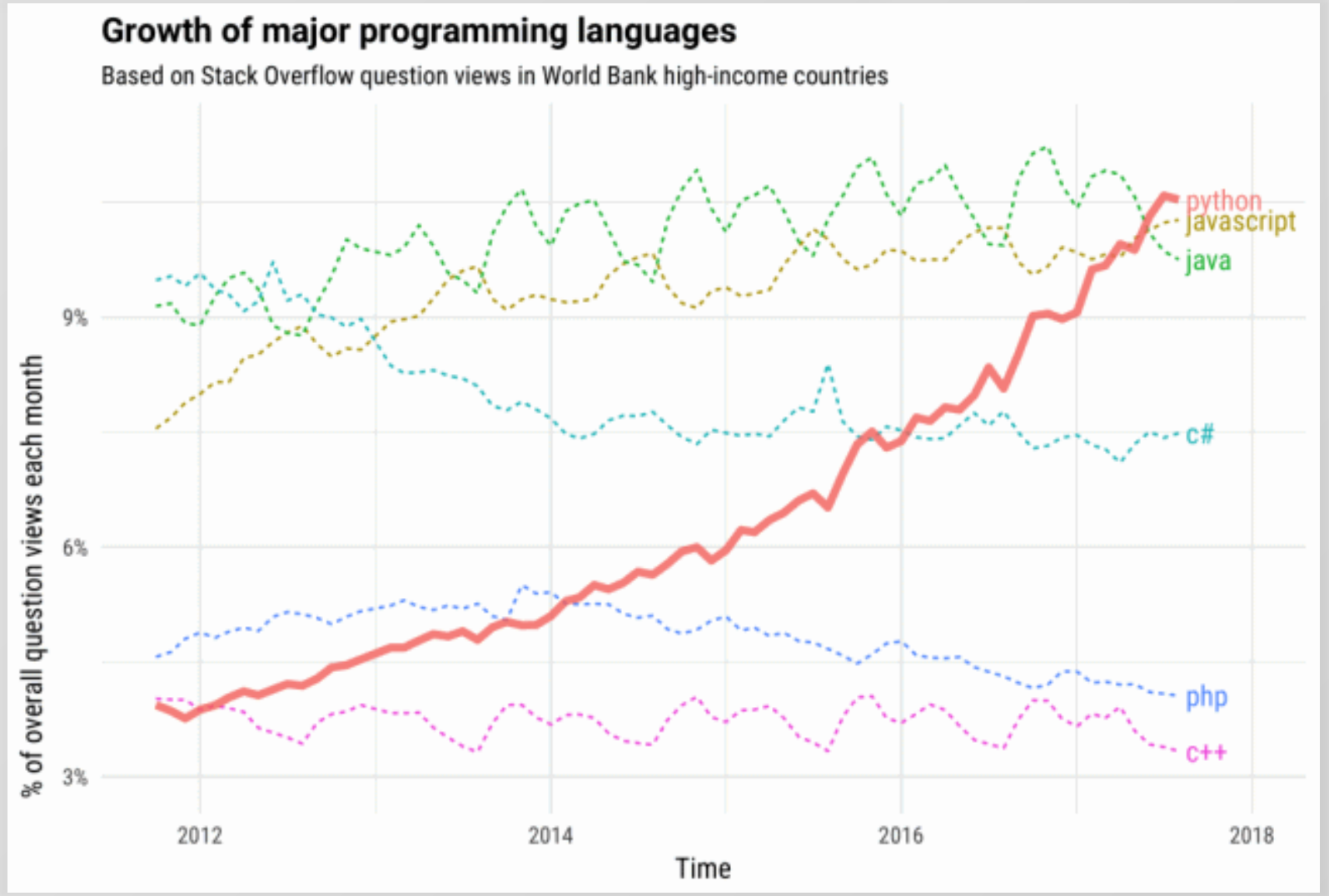
2. Python's simple, **easy to learn syntax** emphasizes readability and its attractiveness for Rapid Application Development.

3. Python supports **modules** and **packages**, which encourages program modularity and code reuse.

4. Since **there is no compilation step** (it is interpreted), the edit-test-debug cycle is incredibly fast.

5. When you have an error in the code the **interpreter prints a stack trace**.

What is and why Python?



Python vs Java

Java

- Statically Typed
- Compiled (low level)
- Complex to learn
- Complex to read
- Fast

Python

- Dinamically Typed
- Intepreted (high level)
- Easy to learn
- Easy to read
- Slow

Perfect for POCs and Data Science!

Python vs Java

Java

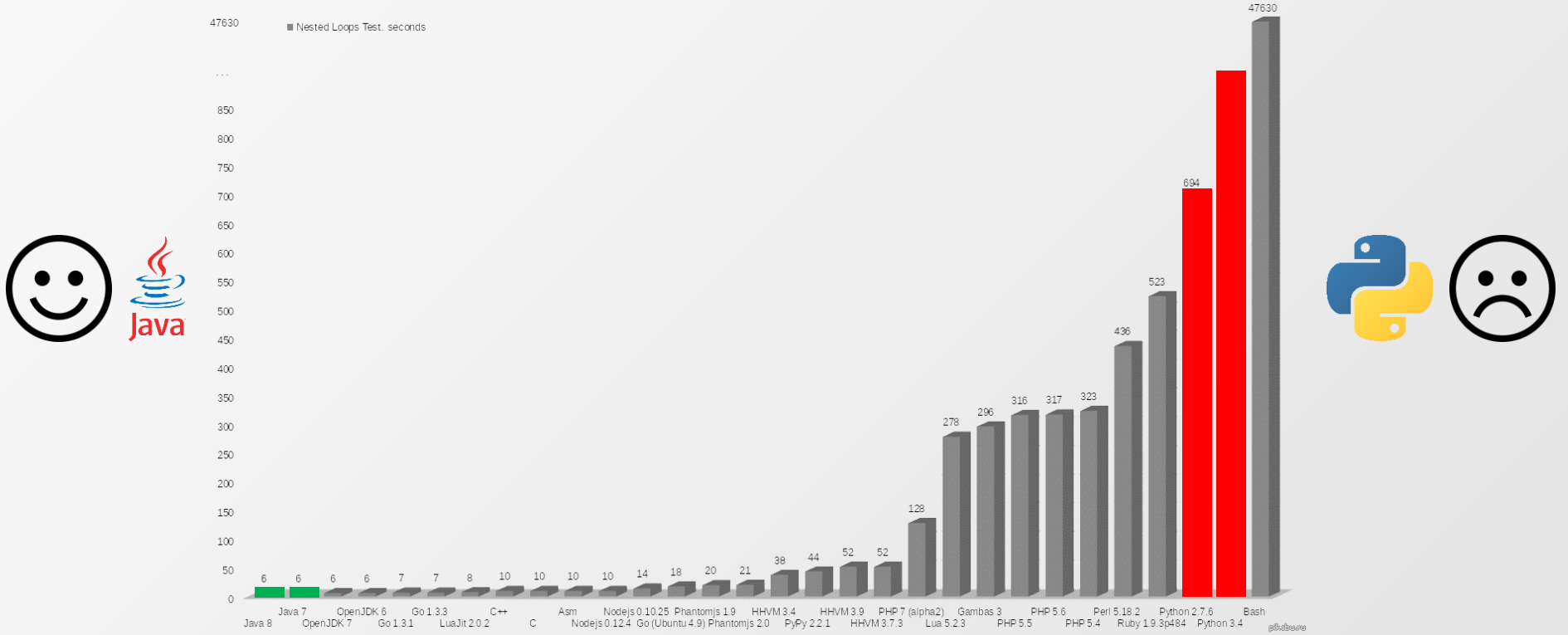
```java
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello World!");
    }

}
```

Python

```python
print("Hello World!")
```

Python vs Java

This easiness translates into:



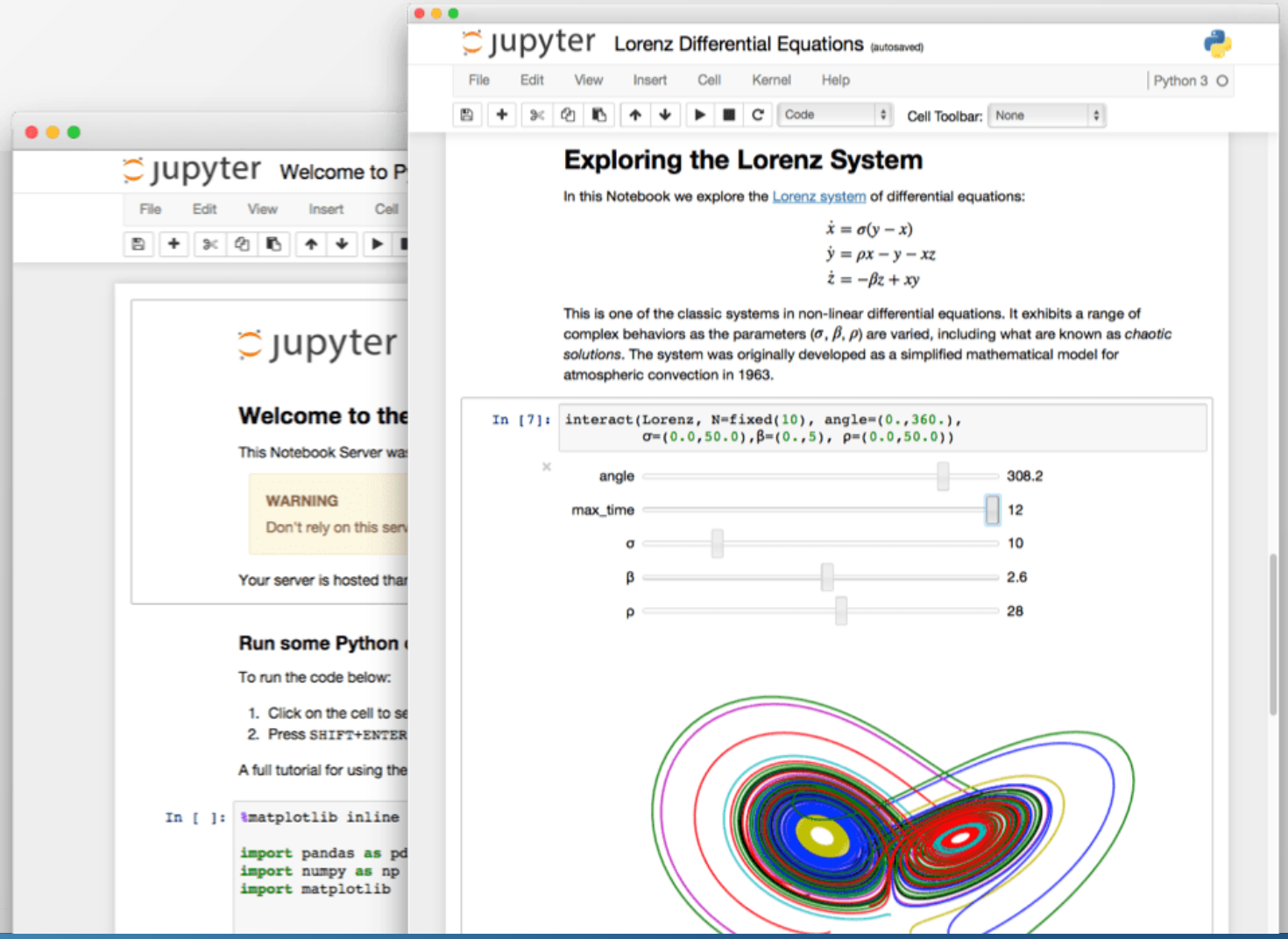
Nested loops

Common IDEs and Developing Tools

Common IDEs and DevTools

Jupyter Notebook

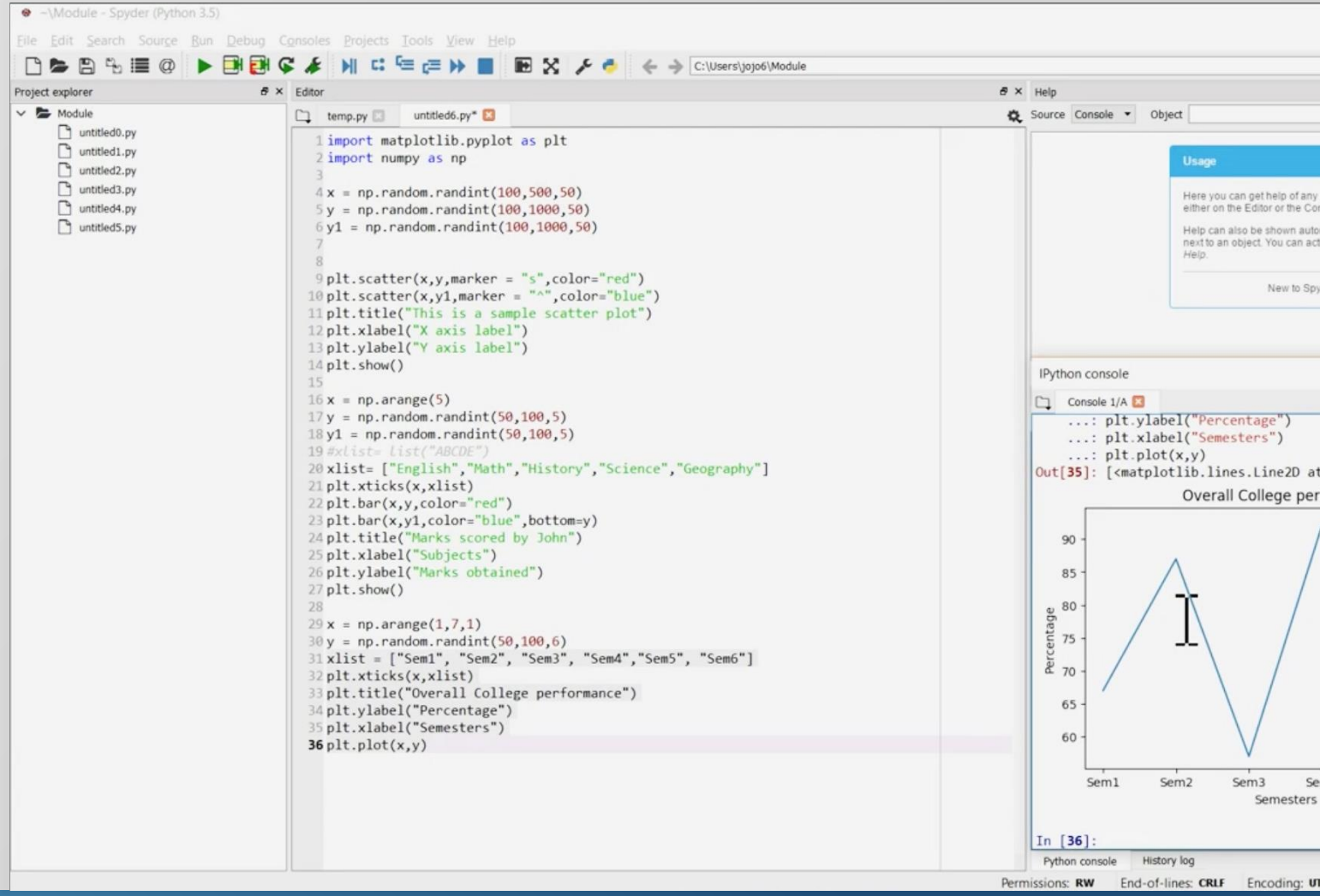
- Perfect for fast prototyping
- Not so good for structured applications
  (i.e. web services, data analytics projects, web apps)



Common IDEs and DevTools

Spyder

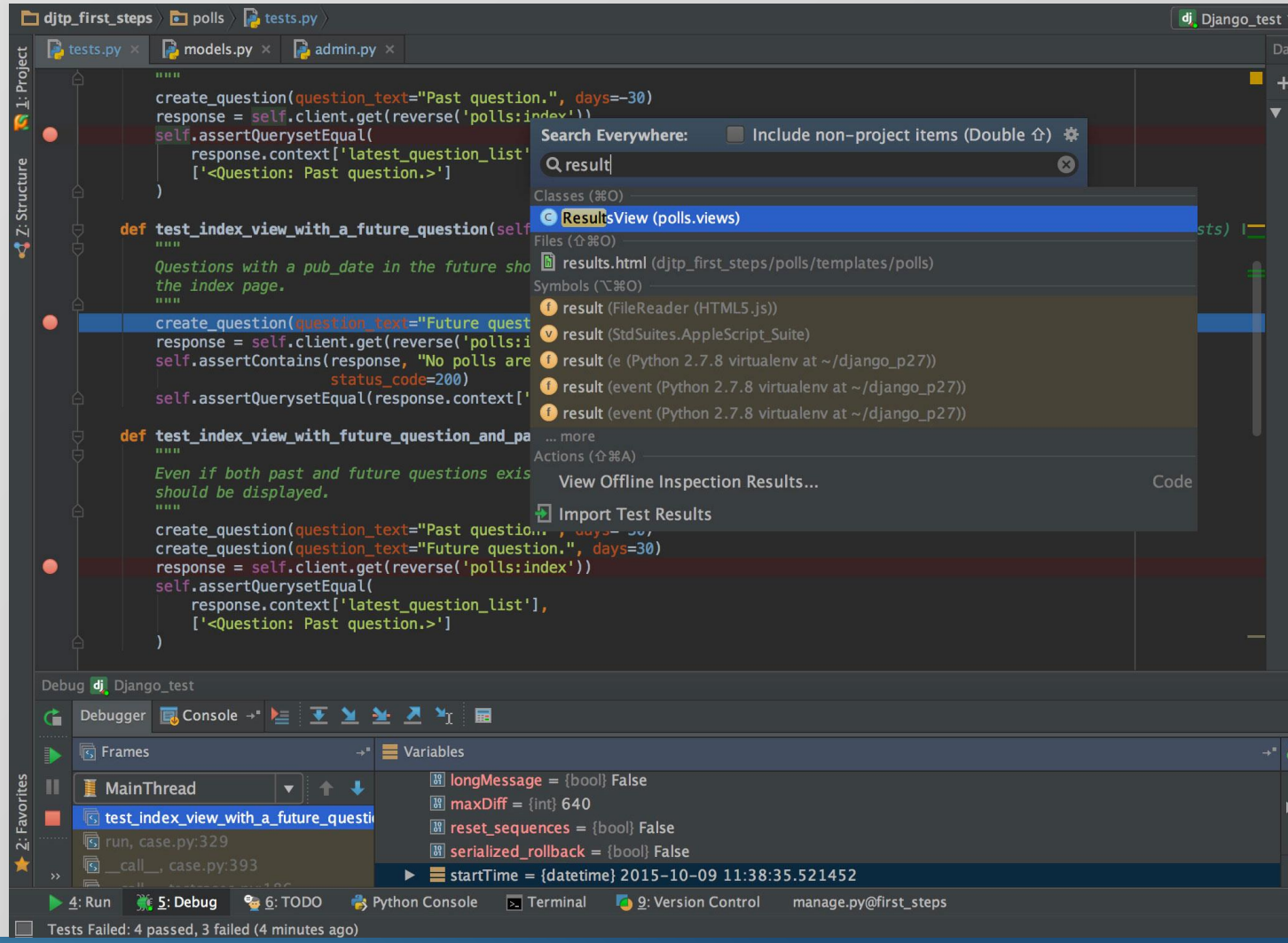
- Nice for fast prototyping
- Well for structured applications
  (i.e. web services, data analytics projects, web apps)



Common IDEs and DevTools

PyCharm

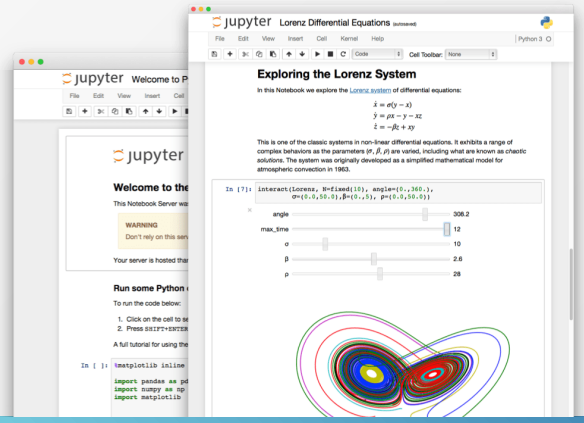
- Not so good for fast prototyping
- Perfect for structured applications
  (i.e. web services, data analytics projects, web apps)



Common IDEs and Tools

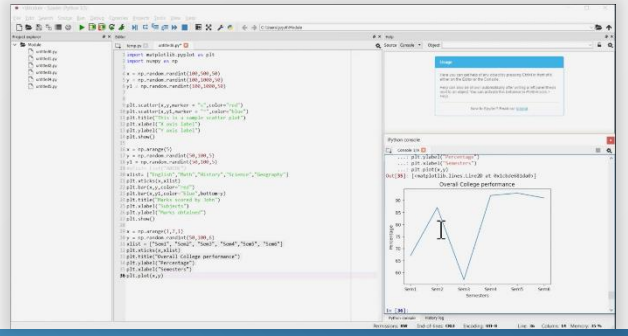
Jupyter Notebook

- Perfect for fast prototyping
- Not so good for structured applications (i.e. web services, data analytics projects, web apps)



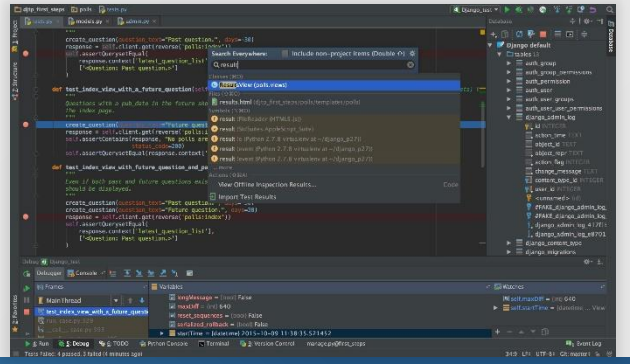
Spyder

- Nice for fast prototyping
- Well for structured applications (i.e. web services, data analytics projects, web apps)



PyCharm

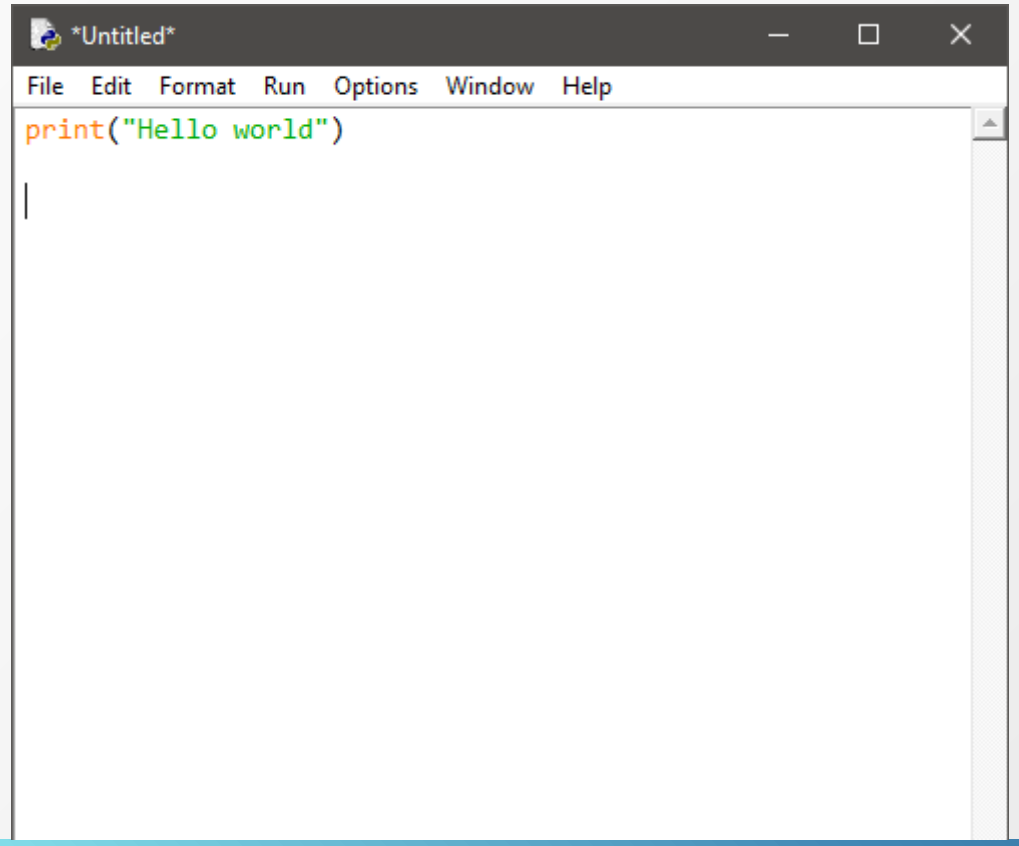
- Well for fast prototyping
- Perfect for structured applications (i.e. web services, data analytics projects, web apps)



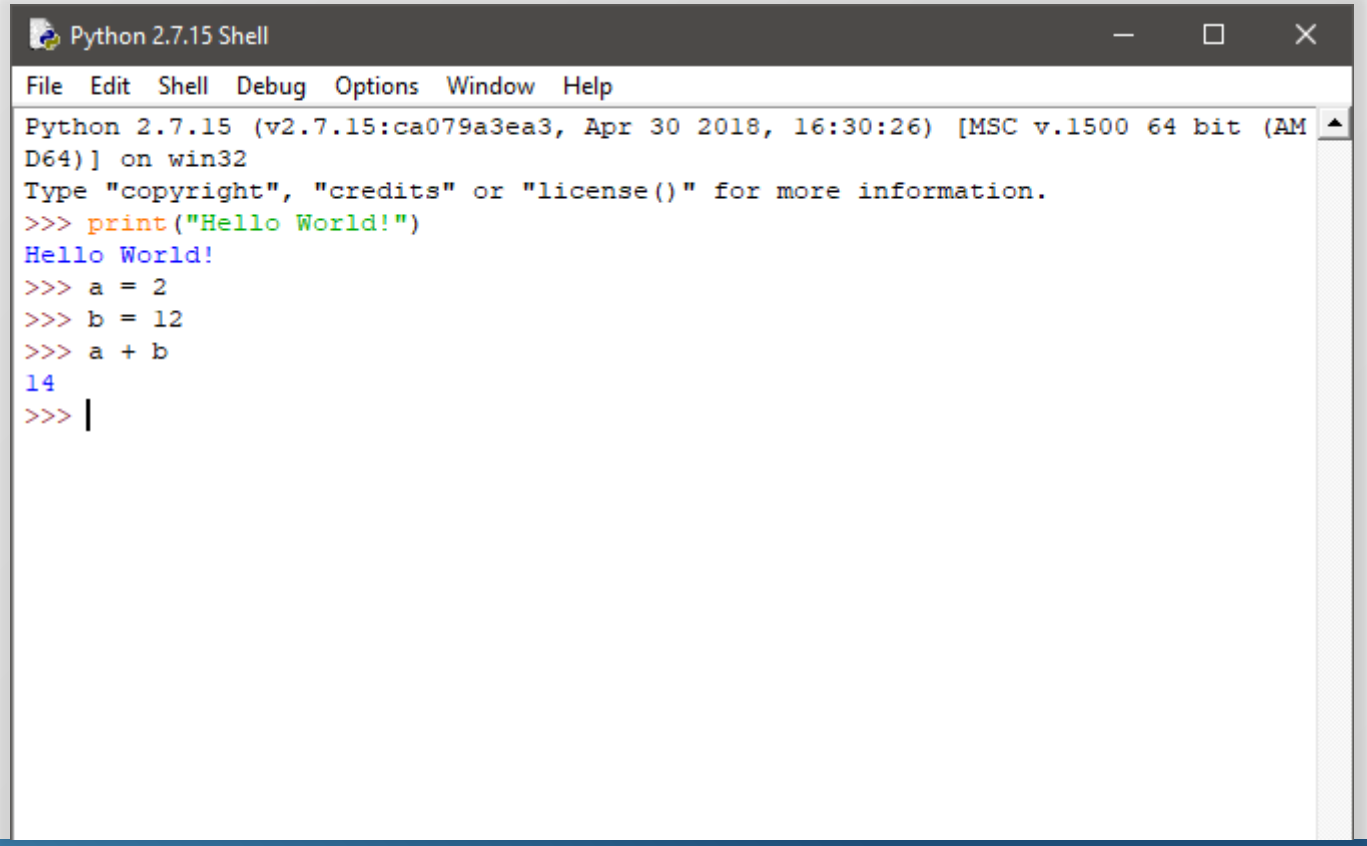
The IDLE and the Shell

The IDLE is the **Integrated Development and Learning Environment** automatically installed with the Python interpreter. Perfect for beginners and to learn the basics (*but also for quick correction to the code!*).

The IDLE



The Shell



Data Types

Variables and data types in Python

- Integer `1  2  3  100  13151  0`
- Float `1.5  2.048128  3.000001`
- String `"Hi! I'm Alessio Vaccaro"`
- List `[1, 2, 7, 10]`
- Tuple `(1, 2, 7, 10)`
- Dictionary `{ "age":"28", "hair":"none" }`
- Booleans `True  False`

IDLE and Integers

Try to open the IDLE Shell and to write this:

```
>>> a = 2
>>> b = 9
>>> print(a + b)
11
>>> c = a + b
>>> print(c)
11
>>> type(c)
<type 'int'>
```

You declared two variables **a** and **b** and assigned them two values (respectively 2 and 9).

You printed their sum (**11**) and then put it into a variable called **c**.

By using the **type()** function you notice that **a**, **b** and **c** are all **integer** variables.

IDLE and Strings

Write this:

```
>>> my_string = "MIDAS Course"
>>> print(my_string)
MIDAS Course
>>> print(my_string[0])
M
>>> print(my_string[5])

>>> print(my_string[4])
S
>>> print(my_string[:4])
MIDA
```

You declared a `string` called `my_string` that contains the words «MIDAS Course».

You can access to single characters (i.e. position x) of the `string` by using the syntax `[x]`.

You can access to a list of characters of the `string` by using the syntax `[:x]` or `[x:]`.

Use `type()` to inspect the type of the variable.

IDLE and Lists 1/4

Write this:

```
>>> my_list = [ 1, 2, 3 ]
>>> print(my_list)
[1, 2, 3]
>>> type(my_list)
<type 'list'>
>>> my_list[0]
1
>>> my_list[2]
3
>>> my_list[3]

Traceback (most recent call last):
  File "<pyshell#72>", line 1, in <module>
    my_list[3]
IndexError: list index out of range
```

You declared a `list` called `my_list` that contains the integers `1`, `2` and `3`.

Strings are lists of characters. Indexing works in the same way **`[x]`**.

IDLE and Lists 2/4

Write this:

```
>>> my_strange_list = [ 1, "bla", 1.4 ]
>>> type(my_strange_list[0])
<type 'int'>
>>> type(my_strange_list[1])
<type 'str'>
>>> type(my_strange_list[2])
<type 'float'>
>>> type(my_strange_list)
<type 'list'>
```

Python lists can contain **everything**. Even another `list` or a `dictionary`:

```
>>> crazy_list = [ 1, "foo" , 2.3, [ 0, 1 ], {"age":23} ]
>>> crazy_list
[1, 'foo', 2.3, [0, 1], {'age': 23}]
```

IDLE and Lists 3/4

Write this:

```
>>> my_list
[1, 2, 3]
>>> my_list.append(15)
>>> my_list
[1, 2, 3, 15]
```

append() is a method to add a new element at the end of a `list`.

```
>>> my_list
[1, 2, 3, 15]
>>> my_list.append([9,8])
>>> my_list
[1, 2, 3, 15, [9, 8]]
```

IDLE and Lists 4/4

Write this:

```
>>> my_list
[1, 2, 3]
>>> my_list[1] = 0
>>> my_list
[1, 0, 3]
```

You replaced that value in the list by selecting it:

```
list[x]
```

and assigning it a new value:

```
list[x] = value
```

IDLE and Tuple 1/2

Write this:

```
>>> my_list = [1,2,3]
>>> my_tuple = (1,2,3)
>>> my_list
[1, 2, 3]
>>> my_tuple
(1, 2, 3)
>>> my_list[0]
1
>>> my_tuple[0]
1
```

Lists and tuples are apparently the same.

IDLE and Tuple 2/2

But they aren't:

```
>>> my_list.append(99)
>>> my_list
[1, 2, 3, 99]
>>> my_tuple.append(99)

Traceback (most recent call last):
  File "<pyshell#143>", line 1, in <module>
    my_tuple.append(99)
AttributeError: 'tuple' object has no attribute 'append'
```

The difference between a `list` and a `tuple` lies in their mutability. A `list` is a mutable object, a `tuple` is an immutable object.

IDLE and Dictionaries 1/3

Write this:

```
>>> my_dict = {"name" : "Alessio", "age" : 28}
>>> my_dict
{'age': 28, 'name': 'Alessio'}
>>> my_dict["age"]
28
>>> my_dict["name"]
'Alessio'
```

You instantiated an object called **my_dict**. This object is a dictionary.

To access values inside the dictionary you use **keys**. Let's try to look at it in this way:

```
{
    "name" : "Alessio",
    "age"  : 28
}
```

This is the same thing we wrote in the code.

IDLE and Dictionaries 2/3

Dictionaries are not immutable object so we can add items when needed.

```
>>> my_dict
{'age': 28, 'name': 'Alessio'}
>>> my_dict["hairs"] = None
>>> my_dict
{'hairs': None, 'age': 28, 'name': 'Alessio'}
```

We added a new element (it is actually a **value**) to our **dictionary** by simply indicating its corresponding **key**.

So we *can not* do this. We can't access **keys** not in the **dictionary**:

```
>>> my_dict
{'hairs': None, 'age': 28, 'name': 'Alessio'}
>>> my_dict["language"]

Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    my_dict["language"]
KeyError: 'language'
```

IDLE and Dictionaries 3/3

To know the **keys** inside a `dictionary` we can:

```
>>> my_dict
{'hairs': None, 'age': 28, 'name': 'Alessio'}
>>> my_dict.keys()
['hairs', 'age', 'name']
>>> my_dict["age"]
28
>>> my_dict["name"]
'Alessio'
>>> my_dict["hairs"]
>>>
```

We used the method `keys()` to inspect all the available **keys** inside the `dictionary`.

Conditionals and Loops

Conditionals and loops

Conditionals
if
if else
if elif else

Conditionals exist to write useful programs, to check something and change the behavior of the program accordingly.

Loops
for
while

Loops exist to automate repeating tasks. Repeating identical or similar tasks without making errors is something that computers do well and people do poorly.

Conditional: **if**

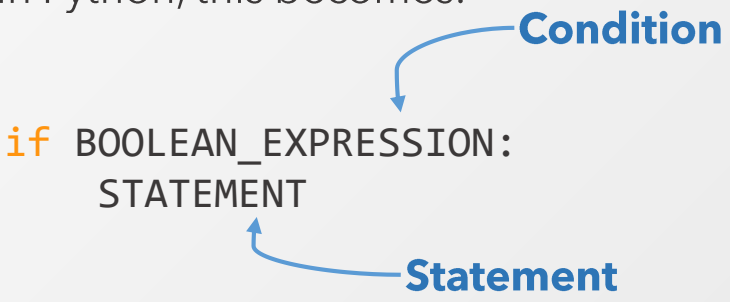
The statement `if` is used to check something. For example:

> «*if the temperature of my room is too high **open** the window*»

Conceptually this can be translated into:

```
if hot:
    open_the_window
```

Formally, in Python, this becomes:



Conditional: **if**

Write this:

```
>>> valueToCheck = 32
>>> if valueToCheck > 22:
        print("HOT")


HOT
```

We used the **if** statement to check if a value (**valueToCheck** in our case) is over a threshold (**22**).

The condition (**valueToCheck > 22**) returned True so the statements printed HOT.

If you try to write:

```
>>> valueToCheck
32
>>> valueToCheck > 22
True
```

Conditional: **if-else**

The `if` statement *do something* if the condition is `True`. But does not anything useful if the condition is `False`.

```
>>> valueToCheck = 21
>>> if valueToCheck > 22:
        print("HOT")
```

We can use the `if-else` statement to add more control to our program:

```
>>> valueToCheck = 21
>>> if valueToCheck > 22:
        print("HOT")
else:
        print("I'm ok :)")


I'm ok :)
```

Conditional: **if-elif-else**

We can also use the **`if-elif-else`** statement to add *more and more* controls.

```
>>> valueToCheck = 20
>>> if valueToCheck > 22:
        print("HOT")
elif valueToCheck < 18:
        print("Brr COLD")
else:
        print("I'm ok :)")


I'm ok :)
```

Loops: **for 1/3**

The `for` statement is born to repeat something a fixed number of times or to access elements of something iterable (i.e. a `list`). More generally for the `for` processes each item of a sequence.

```
>>> listOfNumbers = [1,9,3,5]
>>> for number in listOfNumbers:
        print(number)


1
9
3
5
```

We used the `for` statement to scan the elements of our `list` (`listOfNumbers`).

Loops: **for 2/3**

We can use the **`range()`** function to generate a temporary list to scan.

```
>>> for i in range(6):
        print i


0
1
2
3
4
5
```

We used the **`for`** statement to scan the elements of our **`list`** generated by the **`range()`** function. In fact:

```
>>> range(6)
[0, 1, 2, 3, 4, 5]
```

Loops: **for 3/3**

Other examples of the **`range()`** function.

```
>>> for i in range(0, 10, 2):
        print(i)

0
2
4
6
8
```

```
>>> for i in range(2, 16, 3):
        print(i/4)

0
1
2
2
3
```

```
>>> for i in range(2, 16, 3):
        print(float(i)/4)

0.5
1.25
2.0
2.75
3.5
```

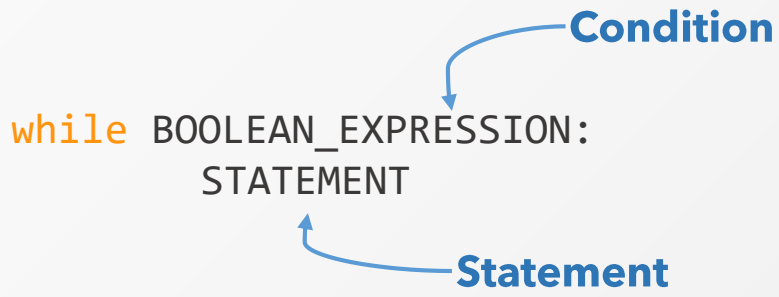
Example: **for** and **if**

Try this:

```
>>> my_values = [12,20,18,19,22,25]
>>> for value in my_values:

        if value > 22:
                print(value, "HOT")
        elif value < 19:
                print(value, "BRR Cold")
        else:
                print(value, "I'm ok! :)")


(12, 'BRR Cold')
(20, "I'm ok! :)")
(18, 'BRR Cold')
(19, "I'm ok! :)")
(22, "I'm ok! :)")
(25, 'HOT')
```

Loops: **While 1/2**

A `while` loop executes an unknown number of times, as long at the `BOOLEAN_EXPRESSION` is `True`:



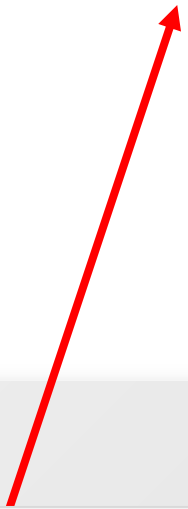

```
>>> while True:
        print("repeating...")


repeating...
repeating...
repeating...
repeating...
repeating
```

Loops: **While 2/2**

Other examples with the **`while`** loop:

```
>>> number = 5
>>> while number > 0:
        print(number)
        number = number - 1


5
4
3
2
1
```

```
>>> number = 3
>>> while number > 1:
        print(number)
        number -= 1


3
2
```



A compact way to write:

```
number = number - 1
```

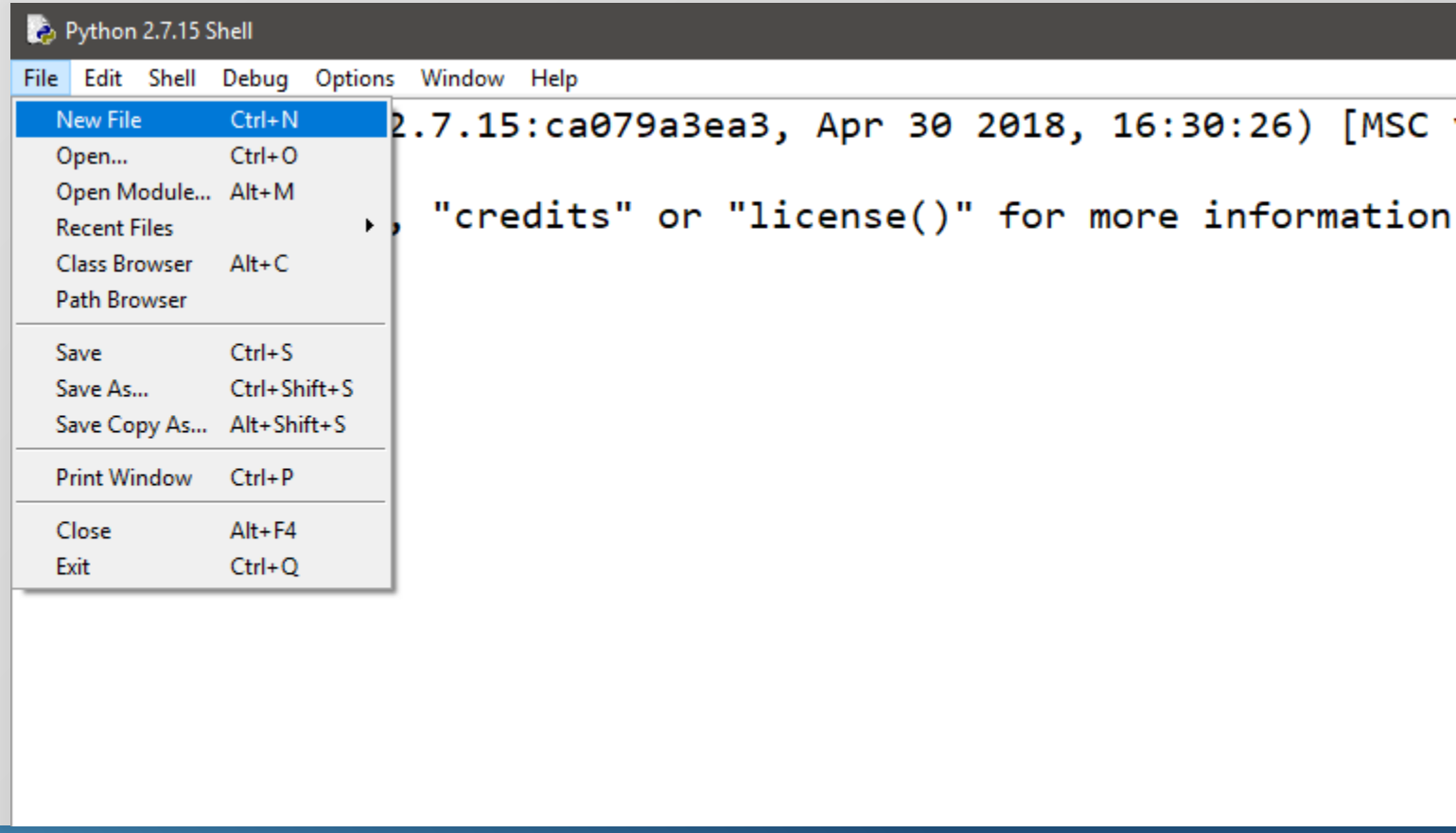
Saving a Script

Scripting

Until now we just played with the IDLE Shell. Now we want to create a script that runs independently.

Steps

1. Open the Shell
2. File → New File

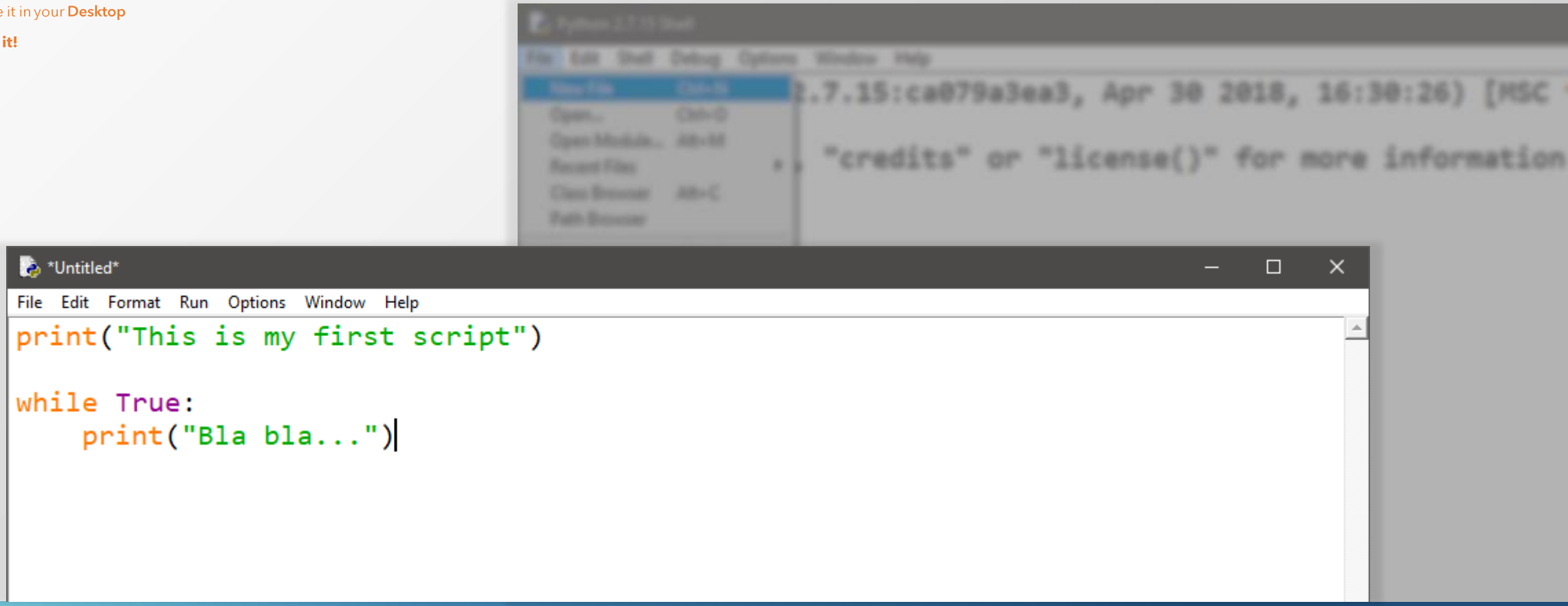


Scripting

Until now we just played with the IDLE Shell. Now we want to create a script that runs independently.

Steps

1. Open the Shell
2. File → New File
3. Save it in your **Desktop**
4. **Run it!**



Jupyter Notebook

Jupyter Notebook

The Jupyter Notebook is an **open-source web application** that allows you to create and share documents that contain live code, equations, visualizations and narrative text.

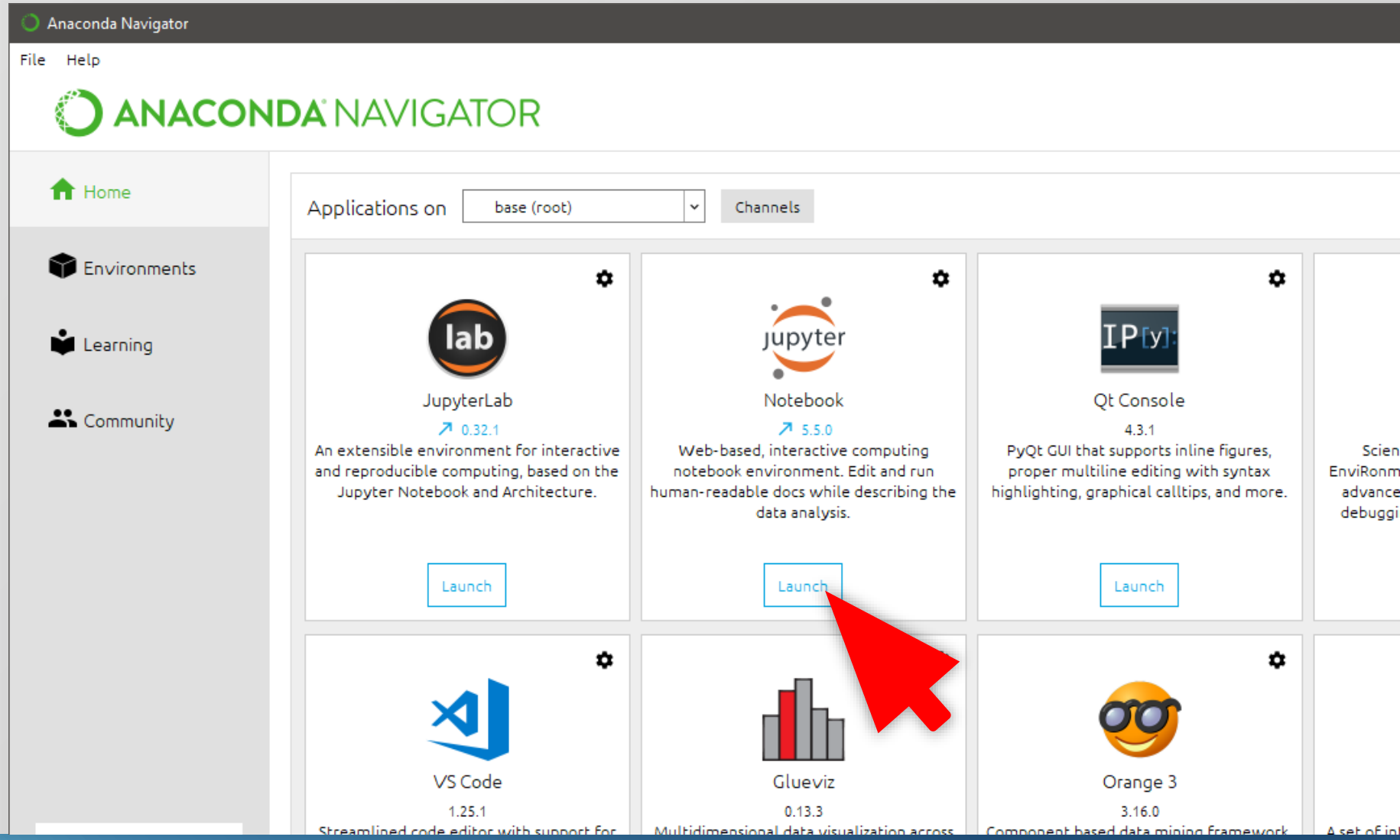
It can be used for: data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more.

It is convenient to install **Jupyter** by installing **Anaconda**, a very popular *Python Data Science Platform* that includes **Spyder** too.

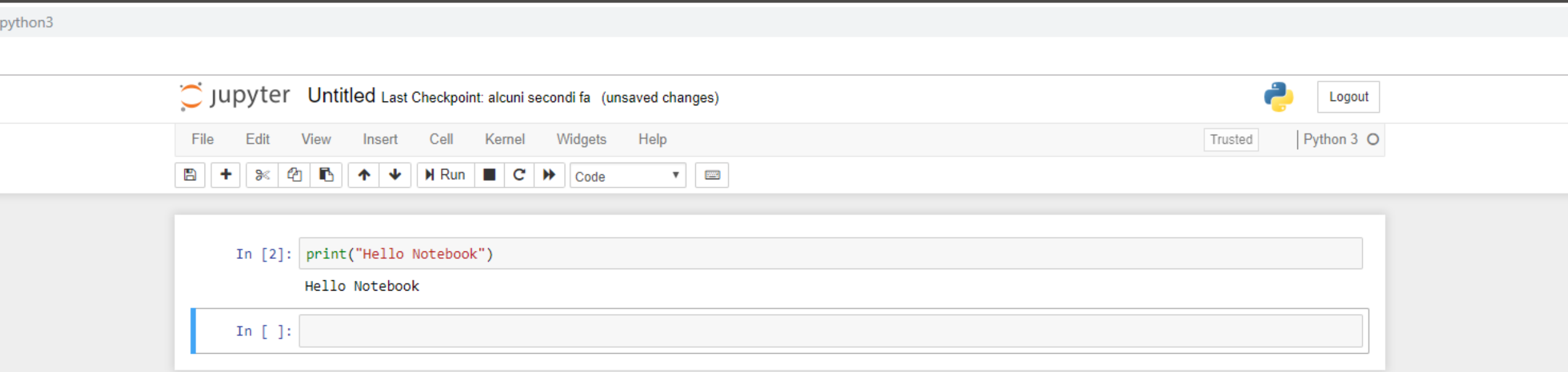
Jupyter Notebook

To open **Jupyter Notebook** open the **Anaconda Navigator** and then Launch the Notebook.

Anaconda Navigator is just a platform that contains a handful of useful tools.



First lines of code in a Notebook



First lines of code in a Notebook

You can write Titles

Subtitles

And text to annotate useful things. Let's define a variable called **a**.

In [3]:
```
a = 2
```

... a list ...

In [9]:
```
my_list = [1, 2, 3, 4]
```

... and let's try a for

In [10]:
```
for i in my_list:
    print(a + i)
```

```
3
4
5
6
```

First lines of code in a Notebook

My Super Expensive Clock

1. Importing modules

In [18]:
```
import time
```

2. Executing the code

In [22]:
```
while True:
    print(time.strftime("%H:%M:%S",time.localtime()))
    time.sleep(1)
```

```
15:12:06
15:12:07
15:12:08
15:12:09
15:12:10
15:12:11
15:12:12
```

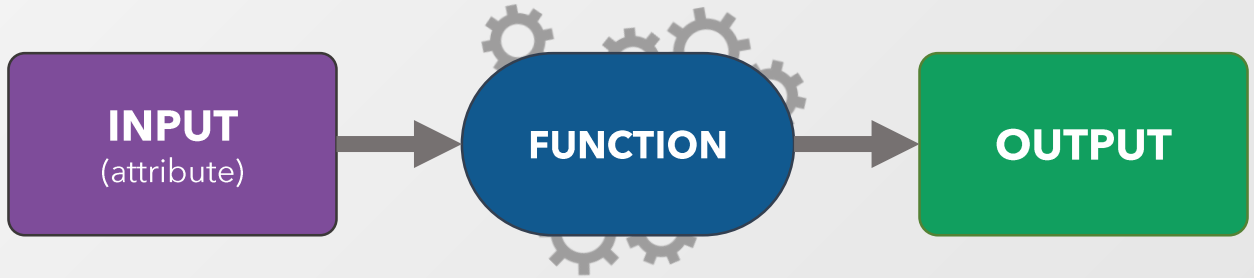
Functions

Functions 1/4

A `function` is piece of code you write once and use many times. The principal purpose of a function is to avoid rewriting code too many times.

This is the syntax of a generic `function` in Python:

```
def my_function(argument1):
        do_something_with_my_argument1
        return(result)
```



Functions 2/4

My first function

1. Define functions

In [30]:
```
def my_math_function(x):
    y = x*2 + 3
    return(y)
```

2. Testing the function

In [35]:
```
my_math_function(2)
```

Out[35]: 7

In [36]:
```
my_math_function(-2)
```

Out[36]: -1

In [37]:
```
my_math_function(1.2)
```

Out[37]: 5.4

Functions 3/4

My first function

1. Define functions

In [25]:
```
def valueChecker(valueToCheck):

    if valueToCheck > 22:
        response = "Hot"
    elif valueToCheck < 18:
        response = "Brr"
    else:
        response = "I'm fine! :)"

    return(response)
```

2. Testing the function

In [26]:
```
valueChecker(15)
```

Out[26]: 'Brr'

In [28]:
```
valueChecker(29)
```

Out[28]: 'Hot'

In [29]:
```
valueChecker(19)
```

Out[29]: "I'm fine! :)"

Functions 3/4

My first function

1. Define functions

```
In [51]: def my_math_function_2(x, y):

             z = x**2 + y*3 + 9

             result = "Your result is %s" % z

             return(result)
```

2. Testing the function

```
In [52]: my_math_function_2(2,5)
Out[52]: 'Your result is 28'

In [54]: my_math_function_2(-2, 1)
Out[54]: 'Your result is 16'
```

A `function` can have more than one argument!

Modules

Modules: time

A `module` is a collection of scripts and `functions` created to allow you to do things faster.

A Python `module` is what in other programming languages is called `library`.

For example we can use the `module` `"time"` that provides several time-related `functions`.

```
>>> import time
>>> time.localtime()
time.struct_time(tm_year=2018, tm_mon=10, tm_mday=19, tm_hour=12, tm_min=57, tm_
sec=58, tm_wday=4, tm_yday=292, tm_isdst=1)
```

```
>>> import time
>>> local_obj = time.localtime()
>>> local_obj
time.struct_time(tm_year=2018, tm_mon=10, tm_mday=19, tm_hour=13, tm_min=4, tm_s
ec=6, tm_wday=4, tm_yday=292, tm_isdst=1)
>>> time.strftime("%H:%M;%S", local_obj)
'13:04;06'
```

Modules: math

Or the `math module` that provides a lot of useful mathematical functions.

```
>>> import math
>>> math.sin(0)
0.0
>>> math.sin(3.14159265359)
-2.0682310711021444e-13
```

In this case the `sin function` of the `math module` wants radiants instead of degrees. Each module has its own documentation (Google is the way).

Modules: math

Or the `module` `"math"` that provides a lot of useful mathematical functions.

```
>>> import math
>>> math.pi
3.141592653589793
```

As you can see `math.pi` is without parenthesis instead of `math.sin()`.

This is why `pi` is an **attribute** of the `math` module, `sin()` is a **function** instead.

`Functions` ***do things***, `attributes` ***are constant (but changeable) values/characteristics***.

Modules: random

`module` `“random”` is useful to generate random values.

```
>>> import random
>>> random.randint(0, 10)
3
```

`rand.int()` is a `function` (*parenthesis!*) that wants two attributes to generate a random value between the two defined limits.

Modules and **documentation/dir()**

Trying to remember every `attribute/function` of every `module` is almost impossible.

That is why exist ***documentations*** or the `dir() function` (or Google).

```
>>> dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', 'Wichmann
Hill', '_BuiltinMethodType', '_MethodType', '__all__', '__builtins__', '__doc__', '__file__', '__name__', '
__package__', '_acos', '_ceil', '_cos', '_e', '_exp', '_hashlib', '_hexlify', '_inst', '_log', '_pi', '_ran
dom', '_sin', '_sqrt', '_test', '_test_generator', '_urandom', '_warn', 'betavariate', 'choice', 'division'
, 'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'jumpahead', 'lognormvariate', 'normal
variate', 'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'tria
ngular', 'uniform', 'vonmisesvariate', 'weibullvariate']
```

```
>>> dir(math)
['__doc__', '__name__', '__package__', 'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil',
'copysign', 'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial', 'floor', 'fm
od', 'frexp', 'fsum', 'gamma', 'hypot', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log', 'log10', 'log1p', 'modf
', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

File I/O

File I/O

What is file I/O?

I/O stands for **Input/Output** that is the ability to read a file or write inside a file.

With Python (*and the proper modules*) you can read **everything**.

Images

Png, Jpg, Tiff, Gif, Bmp, …

Plain Text files

Txt, Csv, …

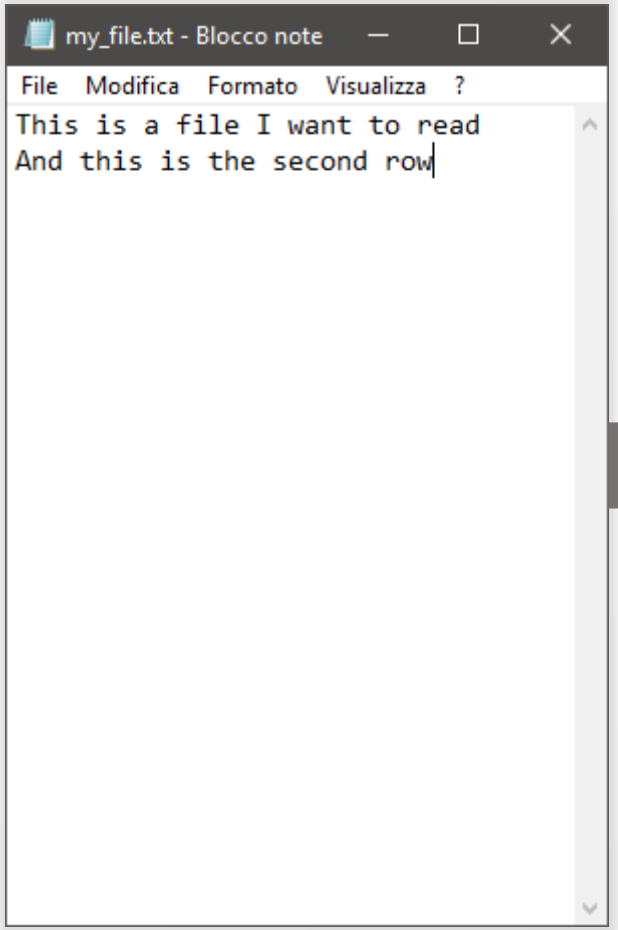
Audio

Mp3, Wav, Ogg, …

Other

Xls, Doc, Ppt, Pdf, …

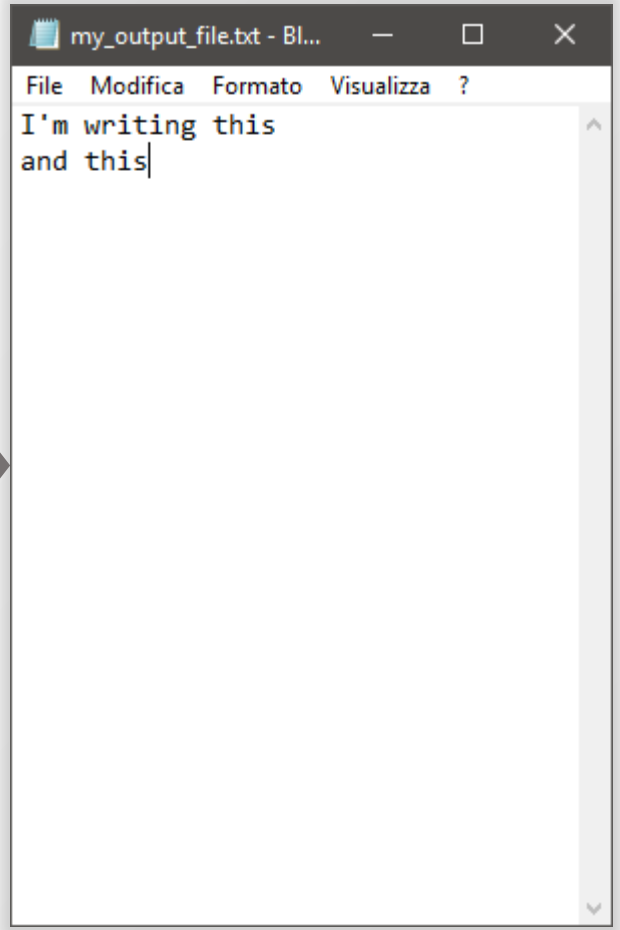
Read a TXT file




```
In [25]: file = open("my_file.txt", "r")

In [26]: file
Out[26]: <_io.TextIOWrapper name='my_file.txt' mode='r' encoding='cp1252'>

In [27]: text = file.read()

In [28]: text
Out[28]: 'This is a file I want to read\nAnd this is the second row'

In [29]: rows = text.split("\n")

In [30]: rows
Out[30]: ['This is a file I want to read', 'And this is the second row']

In [31]: rows[0]
Out[31]: 'This is a file I want to read'
```

Write into a TXT file

```
In [38]: string_to_write = "I'm writing this\nand this"

In [39]: file = open("my_output_file.txt", "w")

In [40]: file.write(string_to_write)
Out[40]: 25

In [42]: file.close()
```



END

Thank you for reading!

Alessio Vaccaro

You can find more here:
www.alessiovaccaro.com
2018