Python Intermediate

Alessio Vaccaro

You can find more here:
www.alessiovaccaro.com

Summary

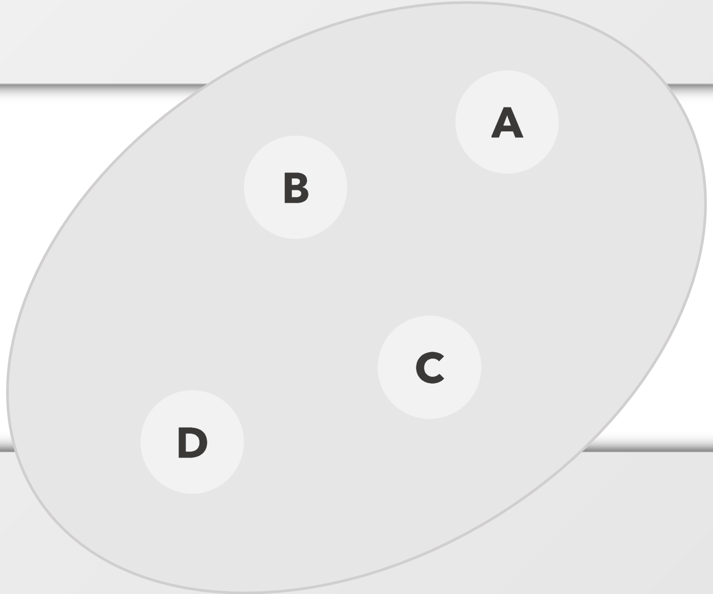
SETS, LIST COMPREHENSION, GENERATORS, CLASSES, EXCEPTIONS, LAMBDA FUNCTIONS, MAP & FILTER FUNCTIONS, PIP & ENVIRONMENTS, GIT

Sets, List Comprehensions and Generators

What is a **SET**?

A **set** is an unordered collection with no duplicate elements.

```
>>> my_set = {"A","B","C","D"}
>>> my_set
set(['A', 'C', 'B', 'D'])
```



SETs 1/3

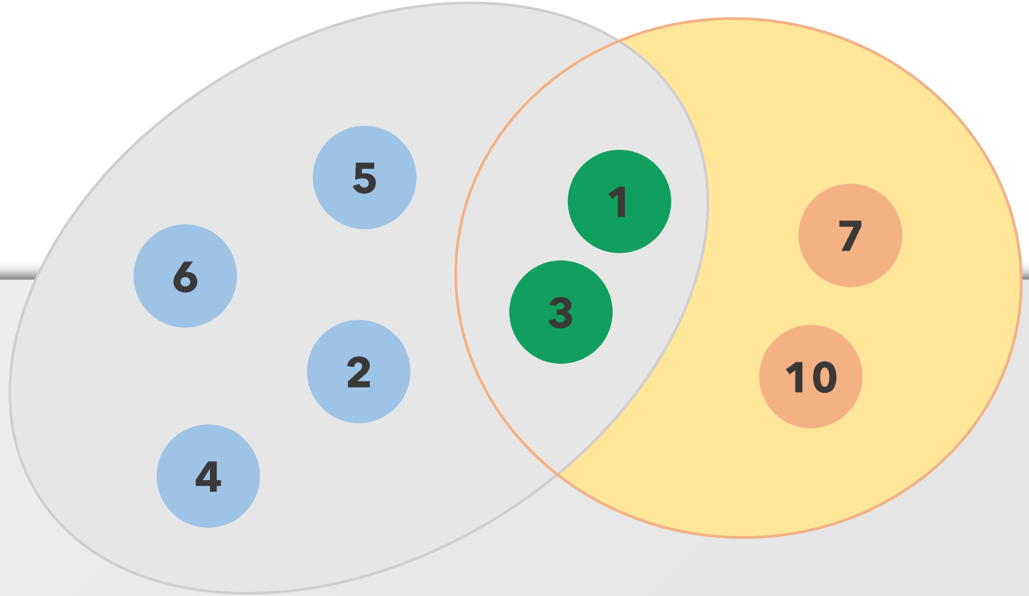
You can transform a **list** (or a **tuple**) into a **set**:

```
>>> my_list = [1,2,3,4,5,5,2,3,6]
>>> my_list
[1, 2, 3, 4, 5, 5, 2, 3, 6]
>>> my_set = set(my_list)
>>> my_set
set([1, 2, 3, 4, 5, 6])
```

SETs 2/3

Sets can be used to perform mathematical set operations like union, intersection, symmetric difference, ecc.…

```
>>> my_set_2 = {1,3,7,10}
>>> my_set_2
set([1, 10, 3, 7])
>>> my_set
set([1, 2, 3, 4, 5, 6])

>>> my_set.intersection(my_set_2)
set([1, 3])
>>> my_set_2.intersection(my_set)
set([1, 3])
```



Some useful operations (`methods`):

```
.intersection()
.union()
.difference()
```

SETs 3/3

You can modify a **set** by adding elements or other **sets**.

```
>>> my_set
set([1, 2, 3, 4, 5, 6])
>>> my_set_2
set([1, 10, 3, 7])
```

```
>>> my_set.add(8)
>>> my_set
set([1, 2, 3, 4, 5, 6, 8])
>>> my_set.update(my_set_2)
>>> my_set
set([1, 2, 3, 4, 5, 6, 7, 8, 10])
```

Common methods: `.add()` `.update()` `.remove()` `.discard()`

What is a **List Comprehension**?

You can see a `list comprehension` as a compact way to create `lists`...

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> [ i*2 for i in range(5) ]
[0, 2, 4, 6, 8]
>>> a = [ i+3 for i in range(3) ]
>>> a
[3, 4, 5]
```

Generators

…or `generators`. A `generator` is an object like a `list` but it returns its elements by calling the `next()` function.

```
>>> my_iter = ( i*2 for i in range(4) )
>>> my_iter
<generator object <genexpr> at 0x03F97F90>
>>> next(my_iter)
0
>>> next(my_iter)
2
>>> next(my_iter)
4
>>> next(my_iter)
6
>>> next(my_iter)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    next(my_iter)
StopIteration
```

List comprehension 1/3

Suppose we have a `list` of words and we want to create another `list` containing the length of each word.

```
my_list = ["Car", "House", "Cat", "Telephone"]
```

```
lengths = [3, 5, 3, 9]
```

Traditional way

With list comprehension

List comprehension 1/3

Suppose we have a **list** of words and we want to create another **list** containing the length of each word.

```
my_list = ["Car", "House", "Cat", "Telephone"]  →  lengths = [3, 5, 3, 9]
```

```
>>> lengths = []
>>> for word in my_list:
        lengths.append(len(word))


>>> lengths
[3, 5, 3, 9]
```

Traditional way

```
>>> lengths = [len(word) for word in my_list]
>>> lengths
[3, 5, 3, 9]
```

With list comprehension

List comprehension 2/3

Suppose we want to sum all the rows of a matrix.

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

→

```
sums = [6, 15, 24]
```

Traditional way

With list comprehension

List comprehension 2/3

Suppose we want to sum all the rows of a matrix.

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

→

```
sums = [6, 15, 24]
```

```
>>> sums = []
>>> for row in my_matrix:
        sums.append(sum(row))


>>> sums
[6, 15, 24]
```

Traditional way

```
>>> sums = [sum(row) for row in my_matrix]
>>> sums
[6, 15, 24]
```

With list comprehension

List comprehension 3/3

Suppose we want to sum **1** to all the elements of the rows of the matrix.

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

```
[[2, 3, 4], [5, 6, 7], [8, 9, 10]]
```

Traditional way

With list comprehension

List comprehension 3/3

Suppose we want to sum **1** to all the elements of the rows of the matrix.

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

→

```
[[2, 3, 4], [5, 6, 7], [8, 9, 10]]
```

-

```
>>> new_matrix = [ [i+1 for i in row] for row in my_matrix ]
>>> new_matrix
[[2, 3, 4], [5, 6, 7], [8, 9, 10]]
```

Traditional way | **With list comprehension**

Classes

Classes 1/10

Creating a new `class` creates a new *type* of object, allowing new *instances* of that type to be made.

Each `class` instance can have `attributes` attached to it for maintaining its state. `Class` instances can also have `methods` (defined by its `class`) for modifying its state.

```
class Car:

    def __init__(self, name, manuf):
        self.name = name
        self.manufacturer = manuf
```

SCRIPT

1. Created a `class` in a saved script;
2. Launched the script in the shell;

```
>>> myCar = Car("Viper GTS", "Dodge")
>>> myCar
<__main__.Car object at 0x03F87930>
```

SHELL

Classes 2/10

Creating a new `class` creates a new *type* of object, allowing new *instances* of that type to be made.

Each `class` instance can have `attributes` attached to it for maintaining its state. `Class` instances can also have `methods` (defined by its `class`) for modifying its state.

```
class Car:

    def __init__(self, name, manuf):
        self.name = name
        self.manufacturer = manuf
```

SCRIPT

```
>>> myCar.name
'Viper GTS'
>>> myCar.manufacturer
'Dodge'
```

SHELL

I can access **`attributes`** of my **`class`** by using the proper syntax.

Classes 3/10

If we try access `attributes` that doesn't exists:

```python
class Car:

    def __init__(self, name, manuf):
        self.name = name
        self.manufacturer = manuf
```

SCRIPT

```
>>> myCar.year
Traceback (most recent call last):
  File "<pyshell#30>", line 1, in <module>
    myCar.year

AttributeError: 'Car' object has no attribute 'year'
```

SHELL

Classes 4/10

But we can add a new `function` to our `class` to add manufacturing year.

```
class Car:

    def __init__(self, name, manuf):
        self.name = name
        self.manufacturer = manuf

    def setYear(self, year):
        self.year = year
```

SCRIPT

```
>>> myCar = Car("Viper GTS", "Dodge")
>>> myCar
<__main__.Car object at 0x036F2B70>
>>> myCar.name
'Viper GTS'
>>> myCar.manufacturer
'Dodge'
>>> myCar.year
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    myCar.year
AttributeError: 'Car' object has no attribute 'year'
>>> myCar.setYear(1995)
>>> myCar.year
1995
```

SHELL

Classes 5/10

But…

```python
class Car:

    def __init__(self, name, manuf):
        self.name = name
        self.manufacturer = manuf

    def setYear(self, year):
        self.year = year
```

SCRIPT

```
>>> myCar.doors = 2
>>> myCar.doors
2
```

SHELL

Classes 6/10

What if we try to print our object? (In this new `class` I decided to set the `self.year` attribute equal to None as default)

```
class Car:

    def __init__(self, name, manuf):
        self.name = name
        self.manufacturer = manuf
        self.year = None
```

```
>>> print myCar
<__main__.car instance at 0x00000000043FF488>
```

Mmm, not so useful informations for me…

Classes 7/10

So, let's try by using the `__str__ special function`.

```
class Car:

    def __init__(self, name, manuf):
        self.name = name
        self.manufacturer = manuf
        self.year = None
        print("A new 'Car' object has been created: %s" % self.name)

    def __str__(self):
        string1 = "Manufacturer: %s\n" % self.manufacturer
        string2 = "Name: %s\n" % self.name
        string3 = "Year: %s" % self.year
        return(string1 + string2 + string3)
```

Classes 8/10

So, let's try by using the `__str__ special function`. And…

```python
class Car:

    def __init__(self, name, manuf):
        self.name = name
        self.manufacturer = manuf
        self.year = None
        print("A new 'Car' object has been created: %s" % self.name)

    def __str__(self):
        string1 = "Manufacturer: %s\n" % self.manufacturer
        string2 = "Name: %s\n" % self.name
        string3 = "Year: %s" % self.year
        return(string1 + string2 + string3)
```

```
>>> myCar = Car("Viper GTS", "Dodge")
A new 'Car' object has been created: Viper GTS
>>> print(myCar)
Manufacturer: Dodge
Name: Viper GTS
Year: None
>>> myCar.year = 1995
>>> print(myCar)
Manufacturer: Dodge
Name: Viper GTS
Year: 1995
```

Classes 9/10

Another interesting special function: __dict__

```
class Car:

    def __init__(self, name, manuf):
        self.name = name
        self.manufacturer = manuf
        self.year = None
        print("A new 'Car' object has been created: %s" % self.name)

    def __str__(self):
        string1 = "Manufacturer: %s\n" % self.manufacturer
        string2 = "Name: %s\n" % self.name
        string3 = "Year: %s" % self.year
        return(string1 + string2 + string3)
```

```
>>> myCar.__dict__
{'name': 'Viper GTS', 'manufacturer': 'Dodge', 'year': 1995}
```

Classes 10/10

Some interesting special functions.

```
object.__str__(self)
object.__repr__(self)
object.__dict__(self)
object.__bytes__(self)
object.__del__(self)
object.__format__(self)
object.__hash__(self)
```

If you modify special functions you modify the way how Python automatically interact with your objects.

Exceptions

What is an Exception?

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called `exceptions`.

```
>>> 4 + "alessio"

Traceback (most recent call last):
  File "<pyshell#138>", line 1, in <module>
    4 + "alessio"
TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> 2/0

Traceback (most recent call last):
  File "<pyshell#141>", line 1, in <module>
    2/0
ZeroDivisionError: integer division or modulo by zero

>>> 2 + a

Traceback (most recent call last):
  File "<pyshell#143>", line 1, in <module>
    2 + a
NameError: name 'a' is not defined
```

Handling Exception 1/3

Let's try write this script and to launch it:

```
while True:

    yourAge = input("Please insert your age: ")

    result = 0.0

    for numbers in str(yourAge):
        result += int(numbers)

    print(2/result)
```

```
Please insert your age: 28
0.2
Please insert your age: 2
1.0
Please insert your age: 23a

Traceback (most recent call last):
  File "C:/Users/Alessio Vaccaro/Desktop/Pytho
n Classes/exceptions.py", line 3, in <module>
    yourAge = input("Please insert your age: "
)
  File "<string>", line 1
    23a
      ^
SyntaxError: unexpected EOF while parsing
```

Handling Exception 2/3

Ok, let's try to handle the **exception** with the **try-except** statement.

```
while True:

    result = 0.0

    try:
        yourAge = input("Please insert your age: ")
        for numbers in str(yourAge):
            result += int(numbers)
        print(2/result)
    except:
        print("Error. Retry...")
```

```
Please insert your age: 28
0.2
Please insert your age: 87j
Error! Retry...
```

Handling Exception 3/3

We can choose to handle different **exceptions** separately if we can predict them.

```python
while True:

    result = 0.0

    try:
        yourAge = input("Please insert your age: ")
        for numbers in str(yourAge):
            result += int(numbers)
        print(2/result)
    except SyntaxError:
        print("Error. Retry...")
    except ZeroDivisionError:
        print("You can't divide by zero. Retry...")
    except NameError:
        print("No letters please. Retry...")
    except:
        print("Generic Error")
```

```
Please insert your age: 28
0.2
Please insert your age: ud
No letters please. Retry...
Please insert your age: 23i
Error. Retry...
Please insert your age: 0
You can't divide by zero. Retry...
```

Lambda Functions

Lambda Functions 1/2

`Lambdas` are one line `functions` called also `anonymous functions`. `Lambdas` are useful when you don't want to use a `function` twice in a program (*we will see later why*).

```
>>> my_function = lambda x : x**2

>>> my_function(3)
9
```

Here we declared and used a `lambda` function. In this example, by declaring its name, we did not use its main property: *anonymity*.

In this example it works exactly as a more compact *normal* `function`.

Lambda Functions 2/2

Lambdas can also handle more attributes (just like functions).

```
>>> my_function = lambda x, y : 2*x + 3*y

>>> my_function(23, 12)
82
```

Map & Filter

Map 1/5

Map – *is a function that* – applies a **function** to all the elements of a **list**.

```
>>> my_list = range(10)
>>> my_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> map(type, my_list)
[<type 'int'>, <type 'int'>, <type 'int'>, <type 'int'>, <type 'int'>, <type
'int'>, <type 'int'>, <type 'int'>, <type 'int'>, <type 'int'>]


>>> my_strange_list = ["Alessio", 28, {"key" : None} ]
>>> map( type, my_strange_list )
[<type 'str'>, <type 'int'>, <type 'dict'>]
```

Map 2/5

Suppose we want to calculate lengths and sums of the **`lists`** of my **`list`**.

```
>>> numbers_list = [ [1,2,3], [4,5,6,7,8], [9,10] ]

>>> map( len, numbers_list )
[3, 5, 2]
>>> map( sum, numbers_list )
[6, 30, 19]
```

But what if we want the results in a single row?

```
numbers_list = [ [1,2,3], [4,5,6,7,8], [9,10] ]  →  [(6, 3), (30, 5), (19, 2)]
```

Map 3/5

Suppose we want to calculate lengths and sums of the `lists` of my `list` putting them in a single `list`.

```
numbers_list = [ [1,2,3], [4,5,6,7,8], [9,10] ]
```

→

```
[(6, 3), (30, 5), (19, 2)]
  sum len
```

Map 3/5

Suppose we want to calculate lengths and sums of the **`lists`** of my **`list`** putting them in a single **`list`**.

```
numbers_list = [ [1,2,3], [4,5,6,7,8], [9,10] ]  ⟶  [(6, 3), (30, 5), (19, 2)]
                                                     sum len
```

```
>>> numbers_list = [ [1,2,3], [4,5,6,7,8], [9,10] ]

>>> map( lambda x: (sum(x), len(x)), numbers_list )
[(6, 3), (30, 5), (19, 2)]
```

LAMBDA!

```
lambda x: (sum(x), len(x))
```

We could have also used a `list comprehension` expression.

```
>>> [ (sum(x), len(x)) for x in numbers_list]
[(6, 3), (30, 5), (19, 2)]
```

Map 4/5

Let's do the same task comparing **for** and **map**.

```
numbers_list = [ [1,2,3], [4,5,6,7,8], [9,10] ]  →  [(6, 3), (30, 5), (19, 2)]
                                                     sum len
```

```
>>> result = []
>>> for element in numbers_list:
        temp = ( sum(element), len(element) )
        result.append(temp)


>>> result
[(6, 3), (30, 5), (19, 2)]
```

```
>>> map( lambda x: (sum(x), len(x)), numbers_list )
[(6, 3), (30, 5), (19, 2)]
```

Map 5/5

So, when to use the **for** syntax, the `list comprehension` expression or the **map** function?

for

- Loops (**for**, **while**) are slower
- Single-threaded locked
- Sequential
- They can be messy

```
As little as possible
```

L.C.

- <u>Perfect for simple operations</u>
- Single-threaded locked
- Sequential

```
[ 15+2*x for x in numbers_list ]
```

map

- Slowly for simple operations
- <u>Perfect for cycling a function</u>
- Multi-thread capability

```
map( myCrazyFunction, numbers_list )
```

Filter 1/2

Filter is a useful **function** to filter an element.

```
>>> numbers_list = range(10)
>>> numbers_list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> filter( lambda x : x%2 == 0, numbers_list )
[0, 2, 4, 6, 8]
```

I used the **filter function** to select even numbers inside my **numbers_list**.

I used a **lambda function** and the `module operator` (x%2) in fact:

```
>>> map( lambda x : x%2 == 0, numbers_list )
[True, False, True, False, True, False, True, False, True, False]
```

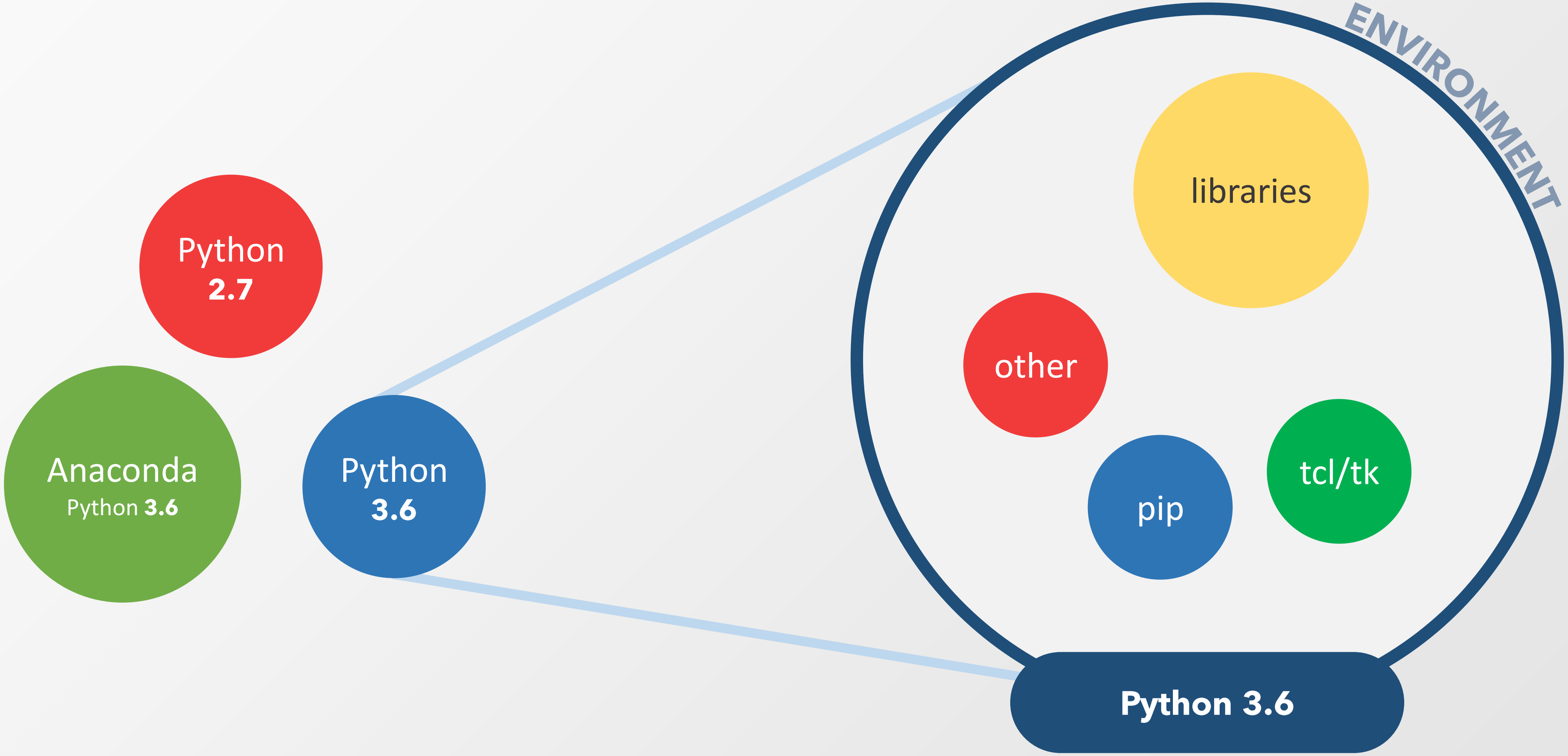
Filter 2/2

Another example with **filter** and a **list** of **strings**:

```
>>> my_strings = ["Pen", "Book", "Phone", "TV", "Camera"]

>>> filter(lambda x : len(x) > 3, my_strings)
['Book', 'Phone', 'Camera']
```

PIP and **Virtual Environments**

PIP and Environments

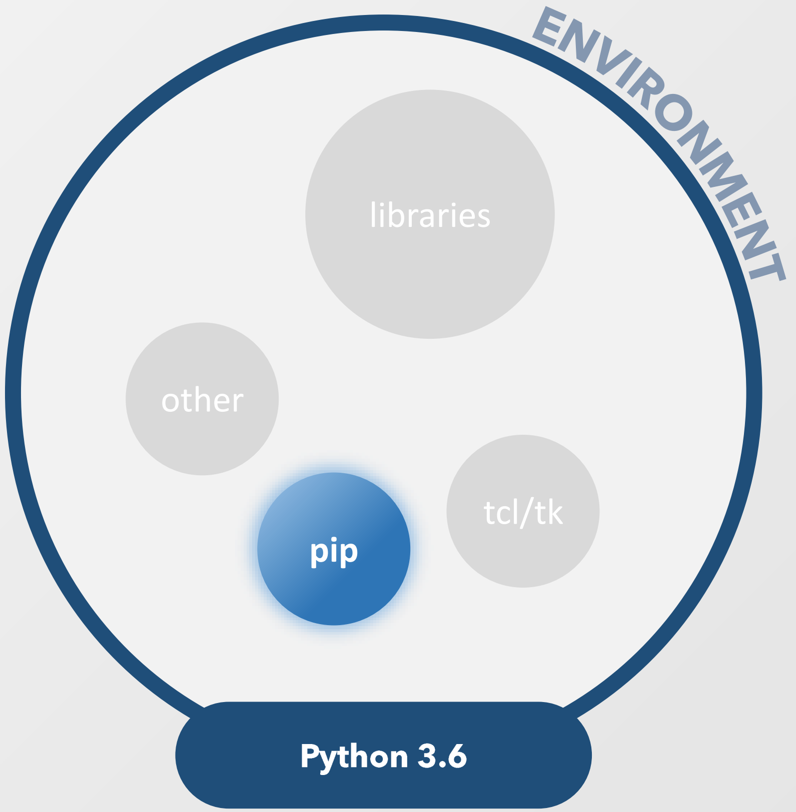


PIP

PIP is a package and modules manager for your Python environments.

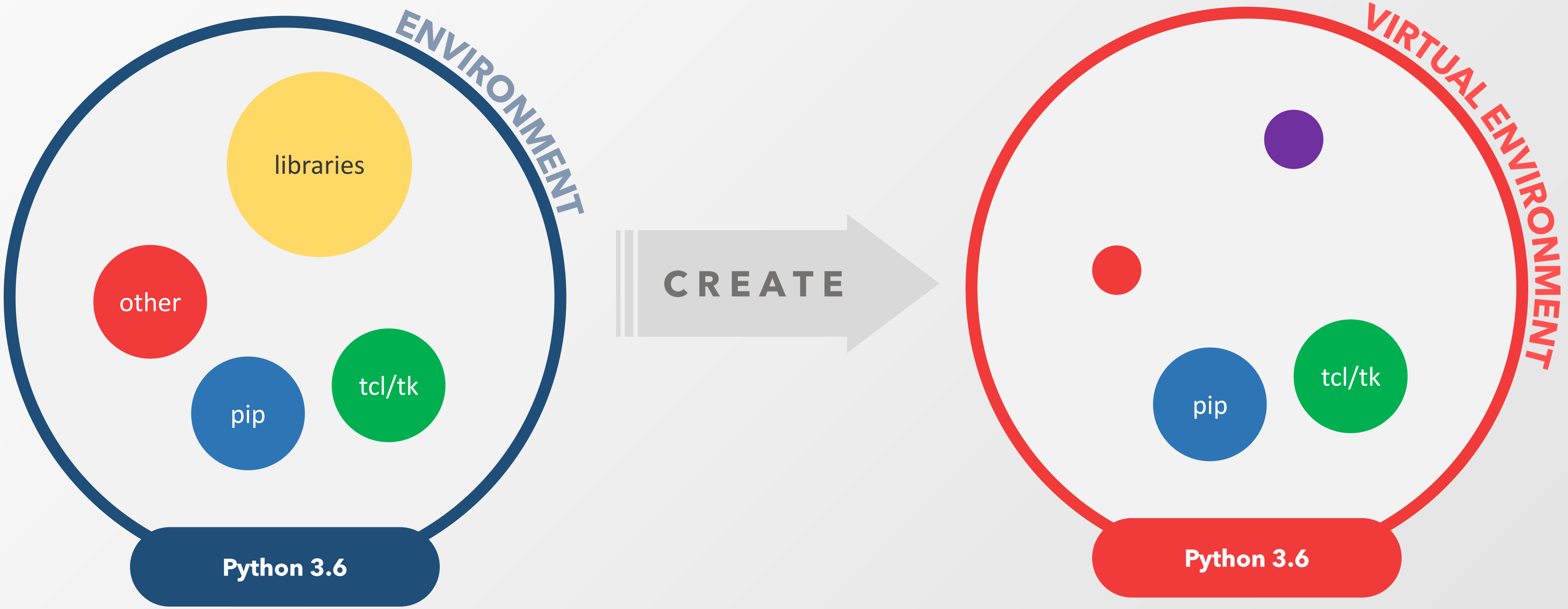
Try:

```
> pip --version
> pip 18.1

> pip install matplotlib
> # to install something


> py -2 -m pip install matplotlib
> # …selecting the right version of Py


> pip install --upgrade matplotlib
> # to install and upgrade something
```



Virtual Environments 1/4

The main purpose of Python virtual environments is to create an **isolated environment** for Python projects. This means that each project can have its own modules.

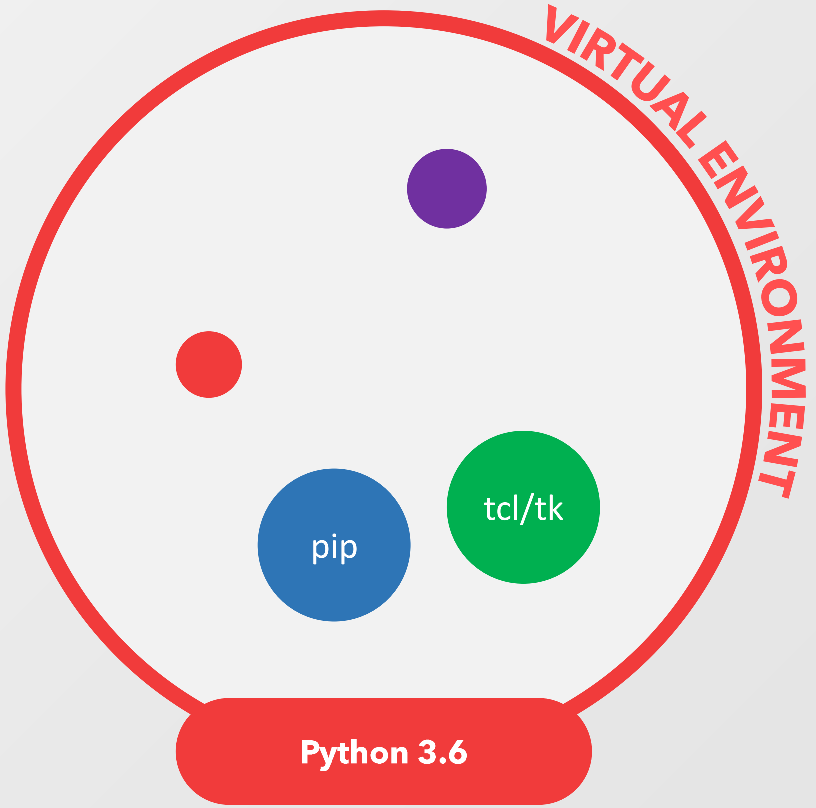


Virtual Environments 2/4

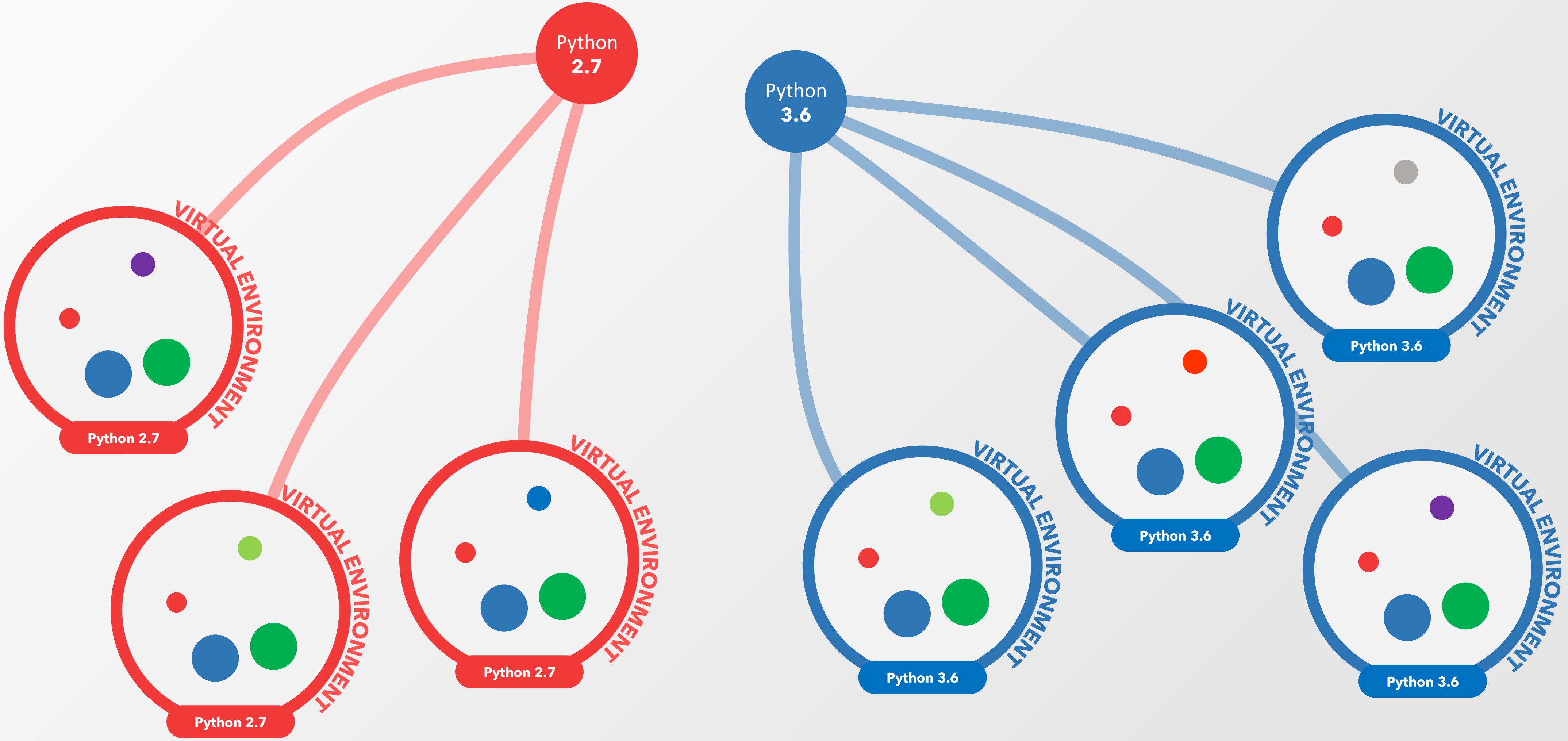
The virtual environment will be created in our selected folder.

At the time of creation the **virtual environment** is empty, without any modules.

Main Python files (*interpreter, compiler, others…*) are taken from the main environment (*father*).



Virtual Environments 3/4



Virtual Environments 4/4

ON WINDOWS	ON LINUX
`> virtualenv -p /usr/bin/python3.6 «my_project»`	`$ virtualenv -p /usr/bin/python3.6 «my_project»`
`> ‘path/to/env/Script/activate’`	`$ source ‘path/to/env/bin/activate’`
`(my_project) C://path/to/env >`	`(my_project) user@localhost:~$`
… do things in the environment…	*… do things in the environment…*
`(my_project) C://path/to/env > deactivate`	`(my_project) user@localhost:~$ deactivate`
`>`	`$`

In Python 3 to create a virtual environment eventually try: `py -3 -m venv «my_project»`

GIT

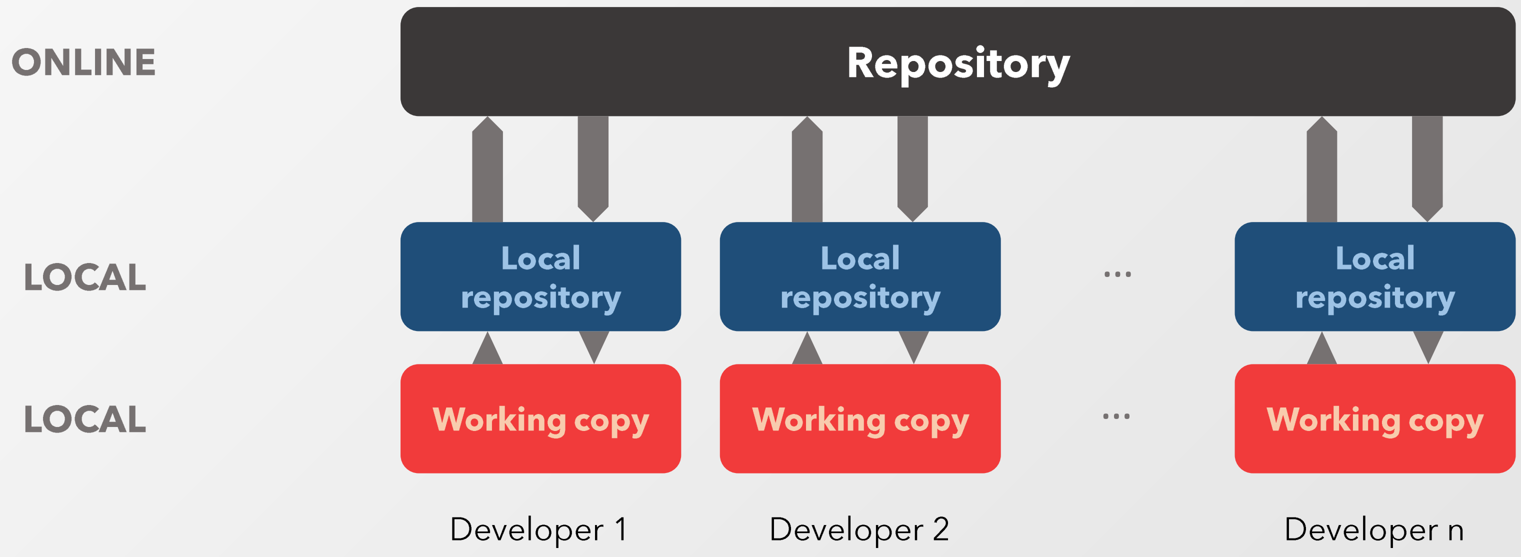
Who uses GIT?

DVCSs

GIT is a **Distributed Version Control System** developed by *Linus Torvalds* in 2005.

A **Version Control System** is a tool that helps developers work at the same time on the same project.

In a DVCS every client has a local copy of the **repository**.



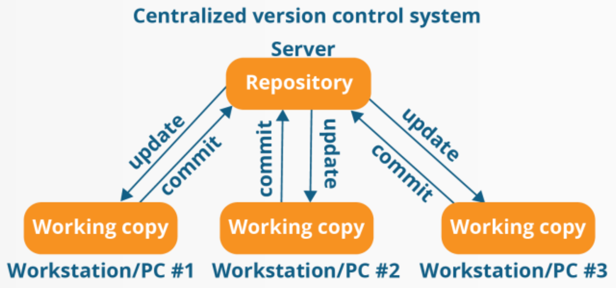
The evolution

1986

CVS

Concurrent Versions System

<u>CENTRALIZED</u>

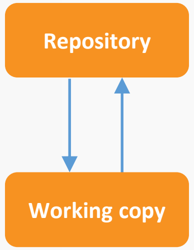


2000

SVN

Subversion

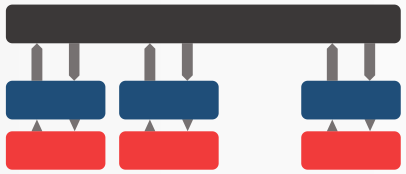
<u>CENTRALIZED</u>



2005

GIT

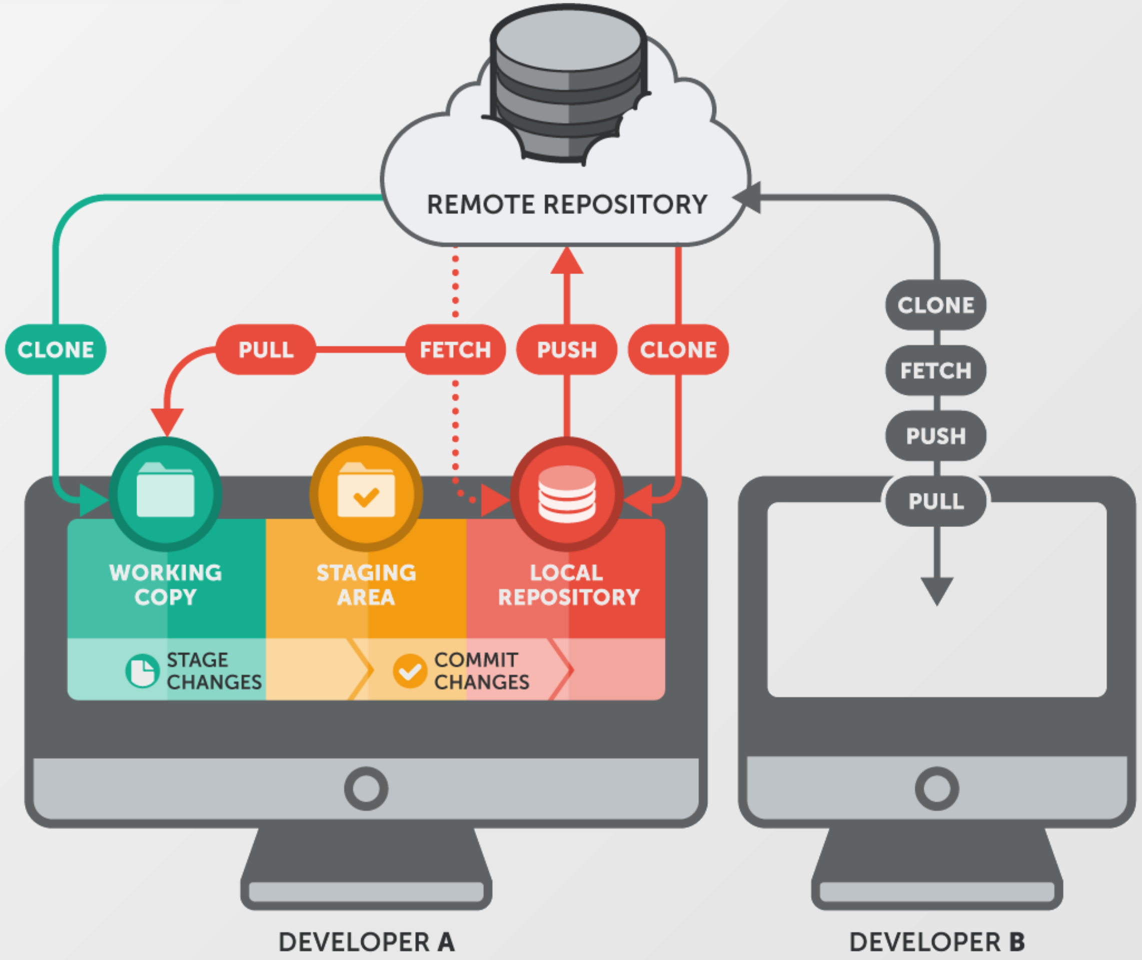
<u>DECENTRALIZED</u>



GIT: Example

Suppose you want to work on **project A**.

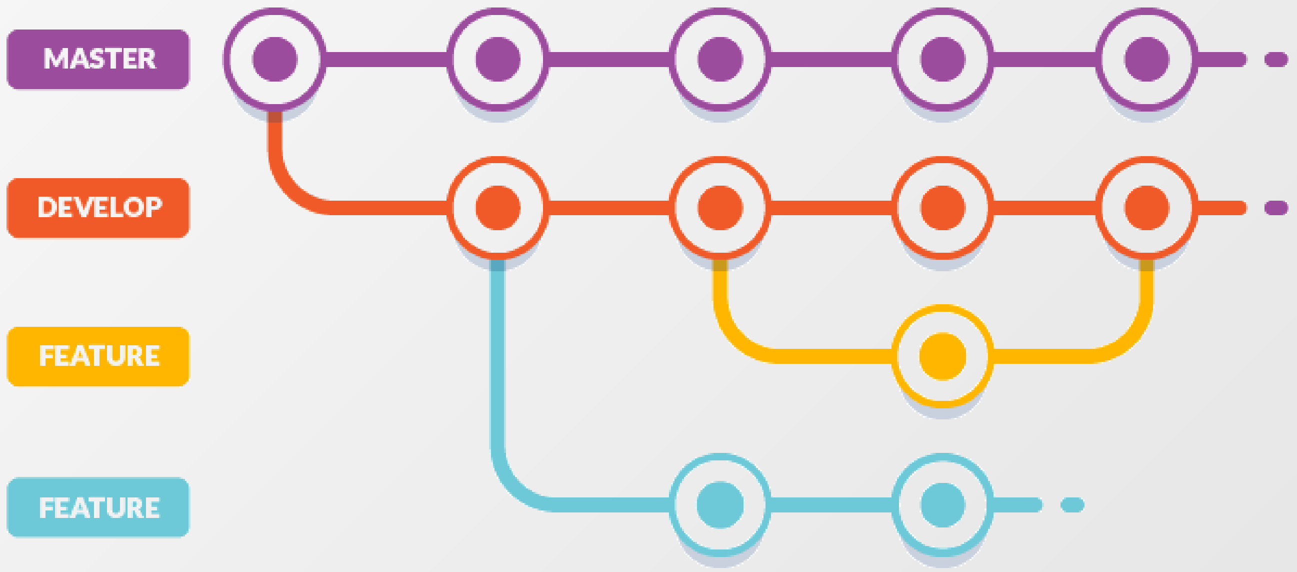
1. You clone **project A** from the **remote repository**;
2. You work on the project adding a new functionality. Your project goes in the staging area because you modify the code;
3. You are proud of your progress on the project so you commit them to your local repository by adding a comment «*Added functionality X*»;
4. You want to share the progress with the community so you decide to push your code to the **remote repository**.
5. Developer B can now work on the **project A** modified by you.



GIT: il Branching

A **branch** in Git is a development path of a project.

The default branch name in Git is **master**. You can add as many nodes as you want and you can go back to a particular node whenever you want.



GIT: Popular Git repository managers

Their principal feature is to work as a "code/project container" namely as a **remote repository**.

GIT: Configuration

After installing configure it by using these commands.

```
$ git config --global user.name "[my username]"

$ git config --global user.email "[my email address]"
```

GIT: Test

Download Git, register to GitLab and try this.

```
$ git init
# I initialized my folder as a local repository

$ git clone git://github.com/link_to_project.git
# I copy inside my local repository a perfect copy of the remote repository

… hours of work on the code …

$ git add .
# my code is now in the staging area ready to be confirmed and committed into my local repository

$ git commit -m ‘Added support to X functionality‘
# I committed my code to the local repository

$ git push origin master
# I uploaded the content of my local repository to the remote repository
```

GIT: Common commands

`git init`	Initialize an existing folder as a local repository
`git clone [url]`	Downloads a project and its entire cronology from a remote repository
`git add [file]`	Add a file in the «staging area» ready for commit towards local repository
`git add .`	Add everything in the «staging area» ready for commit towards local repository
`git commit -m "[message]"`	Registers files and their contents in a permanent way in the version history
`git push [alias] [branch]`	Uploads all local branches in the remote repository
`git pull`	Updates your local repository to the more recent remote repository commit

END

Thank you for reading!

Alessio Vaccaro

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