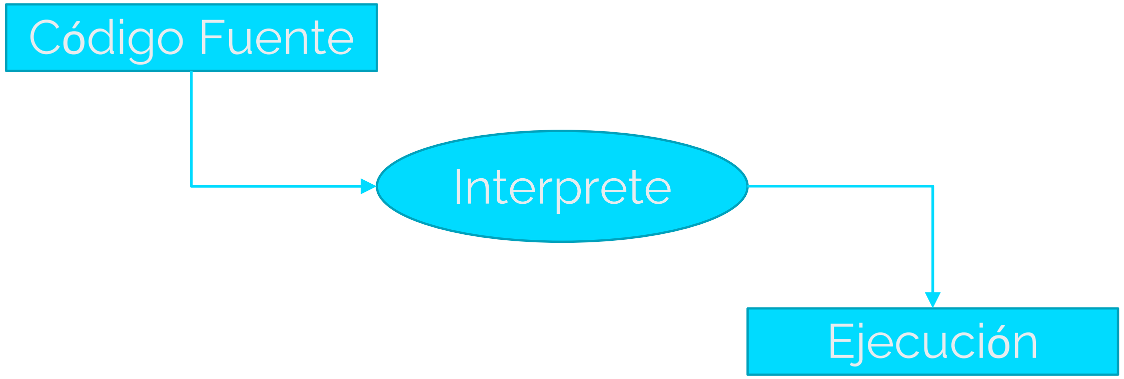
plain concepts

TRAINING: INTRODUCTION TO PYTHON

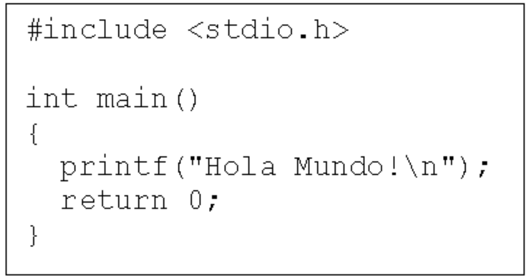
PYTHON

- Lenguaje interpretado.
- Imperativo
- Funcional
- Orientado a objetos
- De tipado dinámico

Filosofía de código legible.

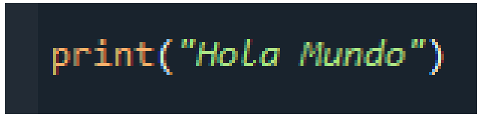


ejemplo C: Hola Mundo!



```c
#include <stdio.h>

int main()
{
  printf("Hola Mundo!\n");
  return 0;
}
```



```python
print("Hola Mundo")
```

ANACONDA

Distribución open source de Python

- Jupyter
- Spyder
- Consola de comandos

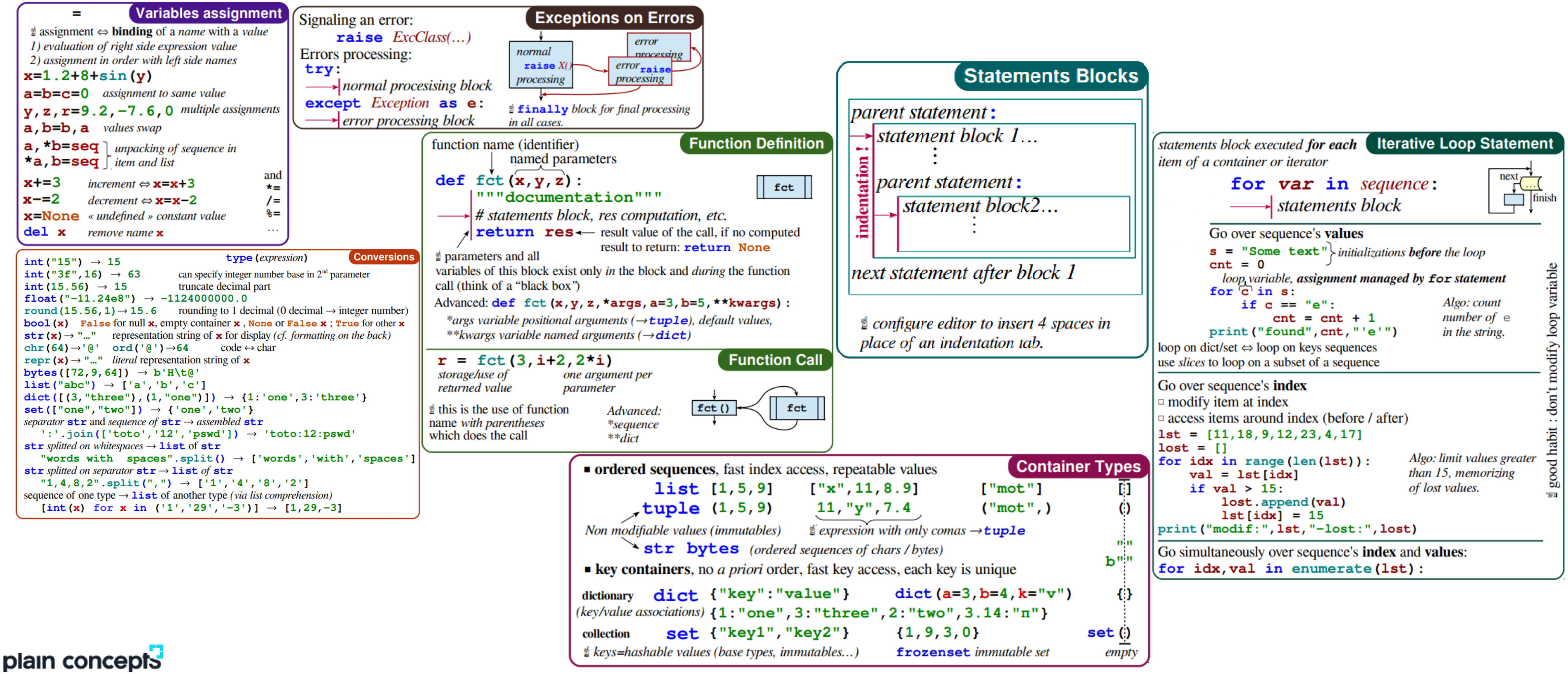
Nos facilita el uso

- Pequeñas pruebas
- Editor básico texto
- Entornos

Descarga Windows

```
conda --help
conda env list
conda create –n test python=3.7
conda activate test
conda list
conda activate base
conda env remove -n test
```

DIFERENCIAS DE PYTHON



Variables assignment

=

☝ assignment ⇔ **binding** of a *name* with a *value*

1) evaluation of right side expression value

2) assignment in order with left side names

```
x=1.2+8+sin(y)
a=b=c=0          assignment to same value
y,z,r=9.2,-7.6,0 multiple assignments
a,b=b,a          values swap
a,*b=seq  } unpacking of sequence in
*a,b=seq  } item and list
x+=3      increment ⇔ x=x+3             and
x-=2      decrement ⇔ x=x-2             *=
x=None    « undefined » constant value  /=
del x     remove name x                 %=
                                        …
```

Conversions

type(*expression*)

```
int("15") → 15
int("3f",16) → 63         can specify integer number base in 2nd parameter
int(15.56) → 15           truncate decimal part
float("-11.24e8") → -1124000000.0
round(15.56,1)→15.6       rounding to 1 decimal (0 decimal → integer number)
bool(x)  False for null x, empty container x , None or False x ; True for other x
str(x)→"…"   representation string of x for display (cf. formatting on the back)
chr(64)→'@'  ord('@')→64      code ↔ char
repr(x)→"…"  literal representation string of x
bytes([72,9,64]) → b'H\t@'
list("abc") → ['a','b','c']
dict([(3,"three"),(1,"one")]) → {1:'one',3:'three'}
set(["one","two"]) → {'one','two'}
separator str and sequence of str → assembled str
   ':'.join(['toto','12','pswd']) → 'toto:12:pswd'
str splitted on whitespaces → list of str
   "words with  spaces".split() → ['words','with','spaces']
str splitted on separator str → list of str
   "1,4,8,2".split(",") → ['1','4','8','2']
sequence of one type → list of another type (via list comprehension)
   [int(x) for x in ('1','29','-3')] → [1,29,-3]
```

Exceptions on Errors

Signaling an error:

```
    raise ExcClass(…)
```

Errors processing:

```
try:
  → normal procesising block
except Exception as e:
  → error processing block
```

☝ **finally** *block for final processing in all cases.*

Function Definition

function name (identifier)

named parameters

```
def fct(x,y,z):
    """documentation"""
    # statements block, res computation, etc.
    return res  ← result value of the call, if no computed
                  result to return: return None
```

☝ parameters and all variables of this block exist only *in* the block and *during* the function call (think of a “black box”)

Advanced: `def fct(x,y,z,*args,a=3,b=5,**kwargs):`

**args variable positional arguments (→`tuple`), default values,*

***kwargs variable named arguments (→`dict`)*

Function Call

```
r = fct(3,i+2,2*i)
```

storage/use of returned value

one argument per parameter

☝ this is the use of function name *with parentheses* which does the call

Advanced:
**sequence*
***dict*

Statements Blocks

parent statement :
  statement block 1…
  ⋮
  parent statement :
    statement block2…
    ⋮
next statement after block 1

☝ *configure editor to insert 4 spaces in place of an indentation tab.*

Container Types

■ **ordered sequences**, fast index access, repeatable values

list	[1,5,9]	["x",11,8.9]	["mot"]	[]
tuple	(1,5,9)	11,"y",7.4	("mot",)	()

Non modifiable values (immutables)

☝ *expression with only comas →* `tuple`

`str` `bytes` *(ordered sequences of chars / bytes)* — `""` `b""`

■ **key containers**, no *a priori* order, fast key access, each key is unique

dictionary	dict	{"key":"value"}	dict(a=3,b=4,k="v")	{}
(key/value associations)		{1:"one",3:"three",2:"two",3.14:"π"}		
collection	set	{"key1","key2"}	{1,9,3,0}	set()

☝ *keys=hashable values (base types, immutables…)*

`frozenset` *immutable set*

empty

Iterative Loop Statement

*statements block executed **for each** item of a container or iterator*

```
for var in sequence:
  → statements block
```

Go over sequence's **values**

```
s = "Some text"  } initializations before the loop
cnt = 0
   loop variable, assignment managed by for statement
for c in s:
    if c == "e":               Algo: count
        cnt = cnt + 1          number of e
print("found",cnt,"'e'")       in the string.
```

loop on dict/set ⇔ loop on keys sequences

use *slices* to loop on a subset of a sequence

Go over sequence's **index**

□ modify item at index

□ access items around index (before / after)

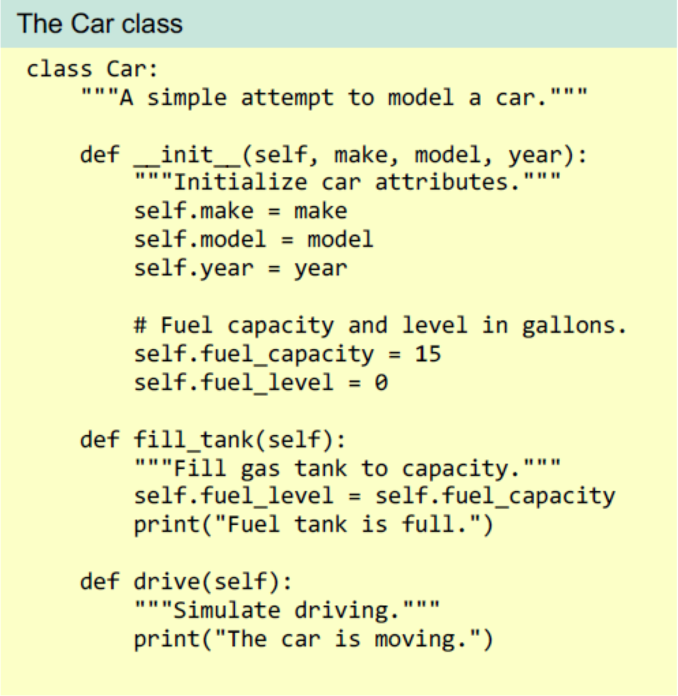
```
lst = [11,18,9,12,23,4,17]
lost = []
for idx in range(len(lst)):            Algo: limit values greater
    val = lst[idx]                     than 15, memorizing
    if val > 15:                       of lost values.
        lost.append(val)
        lst[idx] = 15
print("modif:",lst,"-lost:",lost)
```

Go simultaneously over sequence's **index** and **values**:

```
for idx,val in enumerate(lst):
```

☝ good habit : don't modify loop variable

EJEMPLO DE CLASS



The Car class

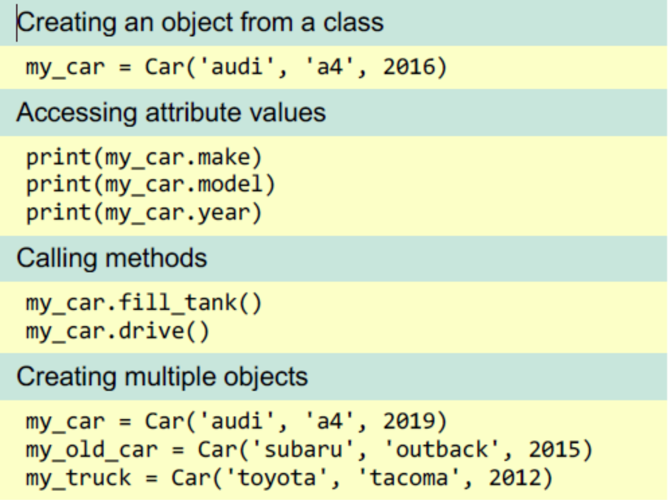
```
class Car:
    """A simple attempt to model a car."""

    def __init__(self, make, model, year):
        """Initialize car attributes."""
        self.make = make
        self.model = model
        self.year = year

        # Fuel capacity and level in gallons.
        self.fuel_capacity = 15
        self.fuel_level = 0

    def fill_tank(self):
        """Fill gas tank to capacity."""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """Simulate driving."""
        print("The car is moving.")
```



Creating an object from a class

```
my_car = Car('audi', 'a4', 2016)
```

Accessing attribute values

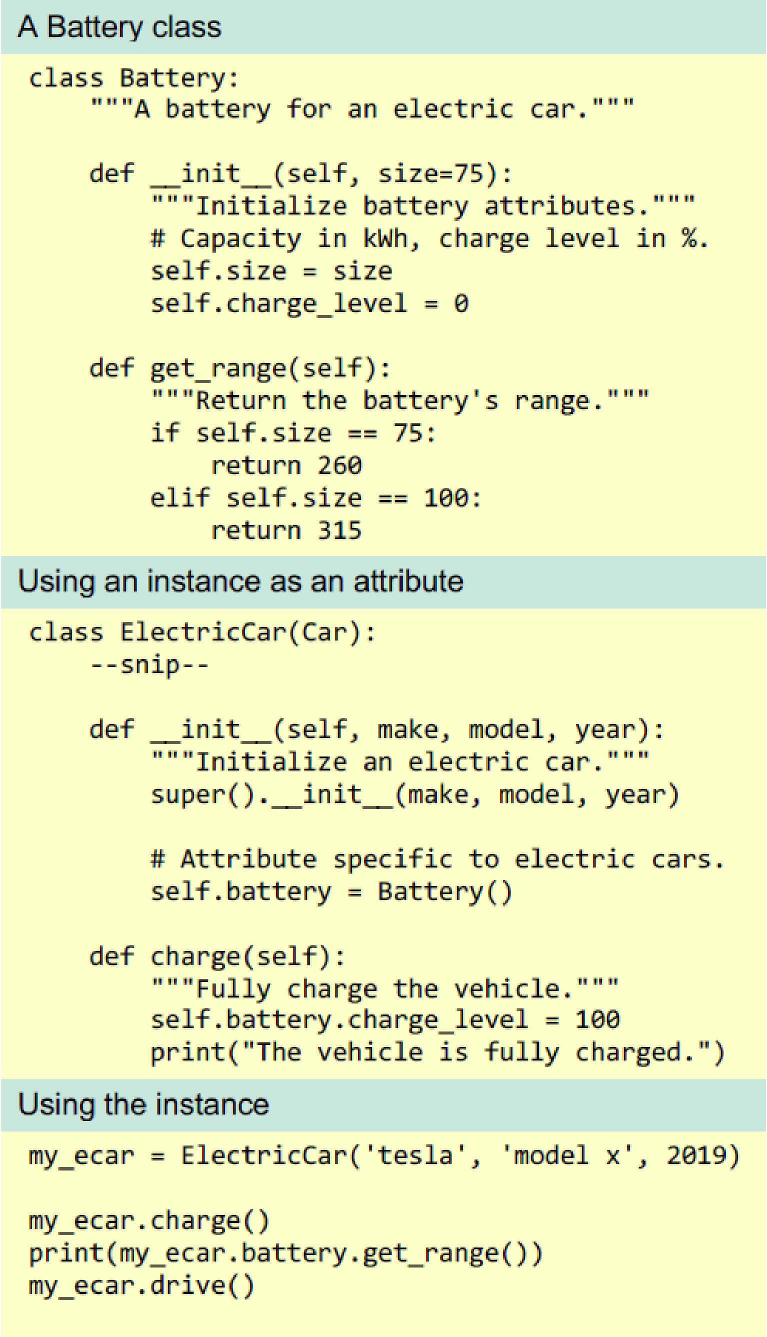
```
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

Calling methods

```
my_car.fill_tank()
my_car.drive()
```

Creating multiple objects

```
my_car = Car('audi', 'a4', 2019)
my_old_car = Car('subaru', 'outback', 2015)
my_truck = Car('toyota', 'tacoma', 2012)
```



A Battery class

```
class Battery:
    """A battery for an electric car."""

    def __init__(self, size=75):
        """Initialize battery attributes."""
        # Capacity in kWh, charge level in %.
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """Return the battery's range."""
        if self.size == 75:
            return 260
        elif self.size == 100:
            return 315
```

Using an instance as an attribute

```
class ElectricCar(Car):
    --snip--

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attribute specific to electric cars.
        self.battery = Battery()

    def charge(self):
        """Fully charge the vehicle."""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

Using the instance

```
my_ecar = ElectricCar('tesla', 'model x', 2019)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

plain concepts

python -m pytest tests/ --junitxml=junit/unit-test.xml

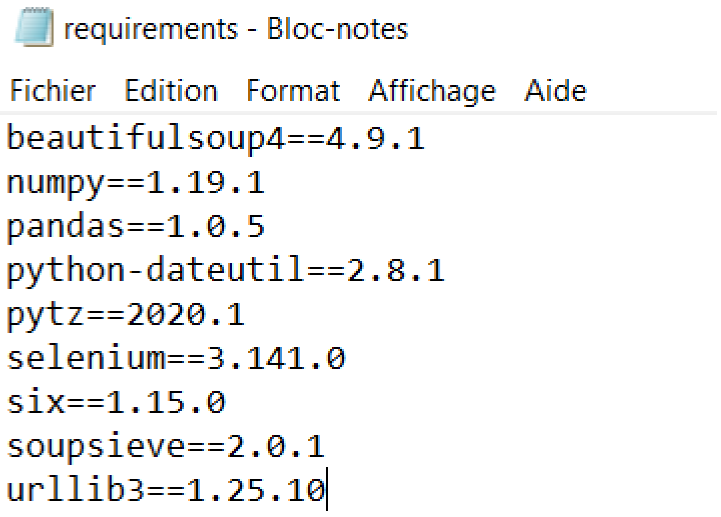
PYTHON PROYECT

src tienes todo el código ejecutable con models y packages

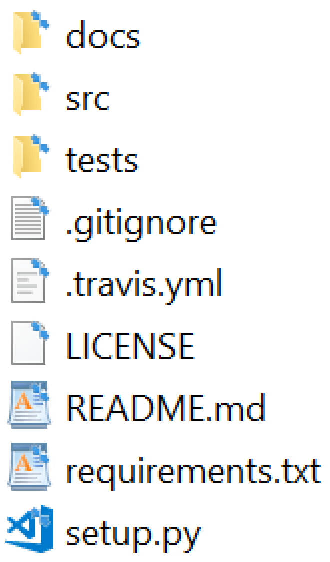
Todo esto se organiza con los **__init__.py**

Es buena práctica el uso de versiones en requirements.txt, debido a las discorcondancias cuando hayan nuevas

tests como el nombre indica se encargara almacezar y organizar la parte del testing



```
beautifulsoup4==4.9.1
numpy==1.19.1
pandas==1.0.5
python-dateutil==2.8.1
pytz==2020.1
selenium==3.141.0
six==1.15.0
soupsieve==2.0.1
urllib3==1.25.10
```



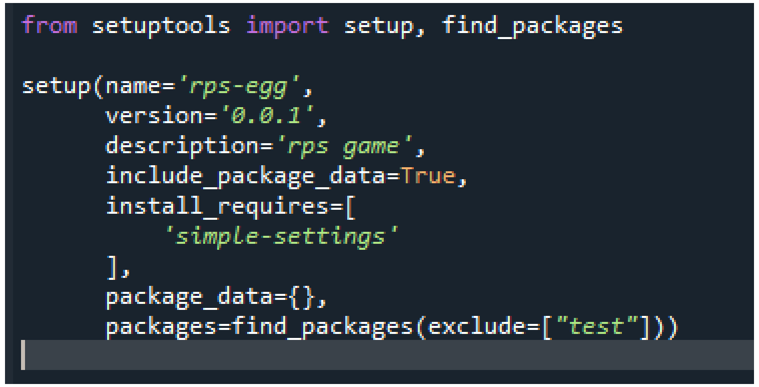
```
docs
src
tests
.gitignore
.travis.yml
LICENSE
README.md
requirements.txt
setup.py
```

PYTHON EGG

Un egg en Python es como un .jar para java

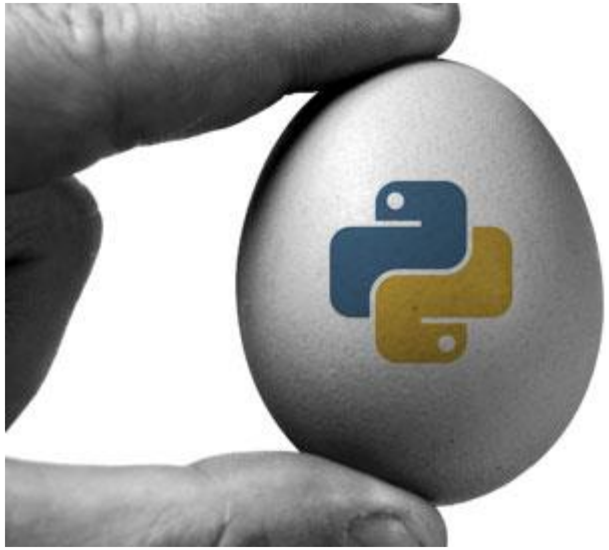
- **python setup.py bdist_egg**

- Añadr a ruta del egg al PYTHONATH



```
from setuptools import setup, find_packages

setup(name='rps-egg',
      version='0.0.1',
      description='rps game',
      include_package_data=True,
      install_requires=[
          'simple-settings'
      ],
      package_data={},
      packages=find_packages(exclude=["test"]))
```



PLAIN CONCEPTS

David Cotilla

dcotilla@plainconcepts.com