

# Khaos Research

# Introducción a Python 3

**Carlos Canicio Almendros**  
**Antonio J. Nebro**

Esta obra se publica bajo licencia [Creative Commons](https://creativecommons.org/licenses/by-nc/4.0/)

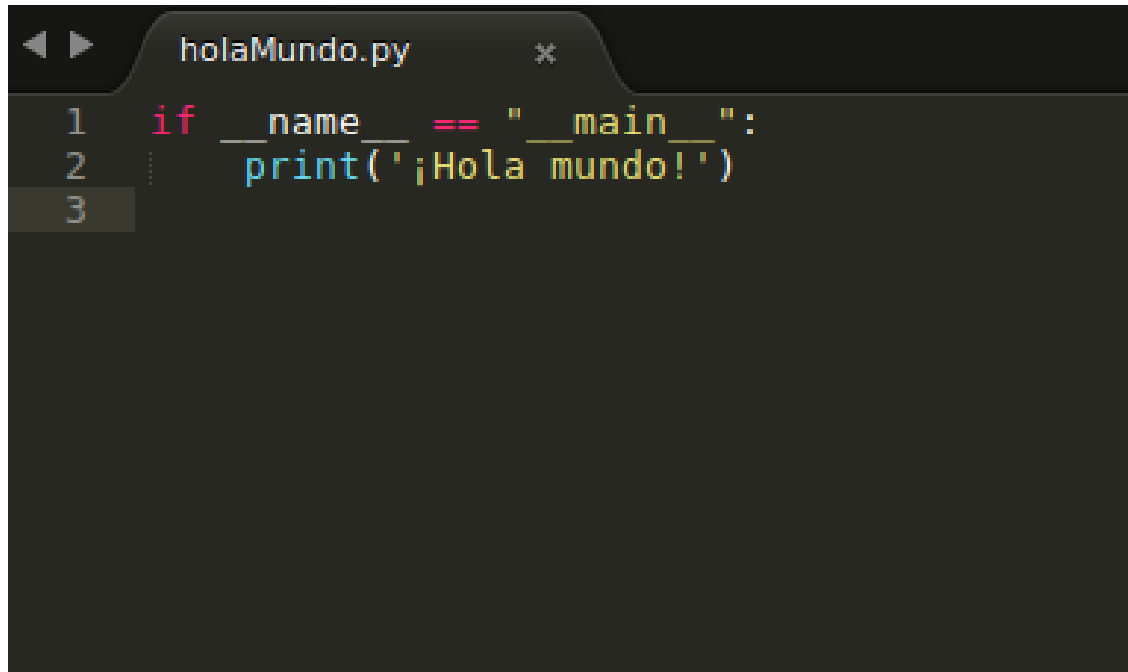


# Índice

- ¿Qué es Python?
- Indentación
- Tipos de datos
- Operadores
- Control de flujo
- Programación orientada a objetos
- Funciones predefinidas
- Estilo de código: PEP8
- Dependencias
- Pruebas unitarias

# ¿Qué es Python?

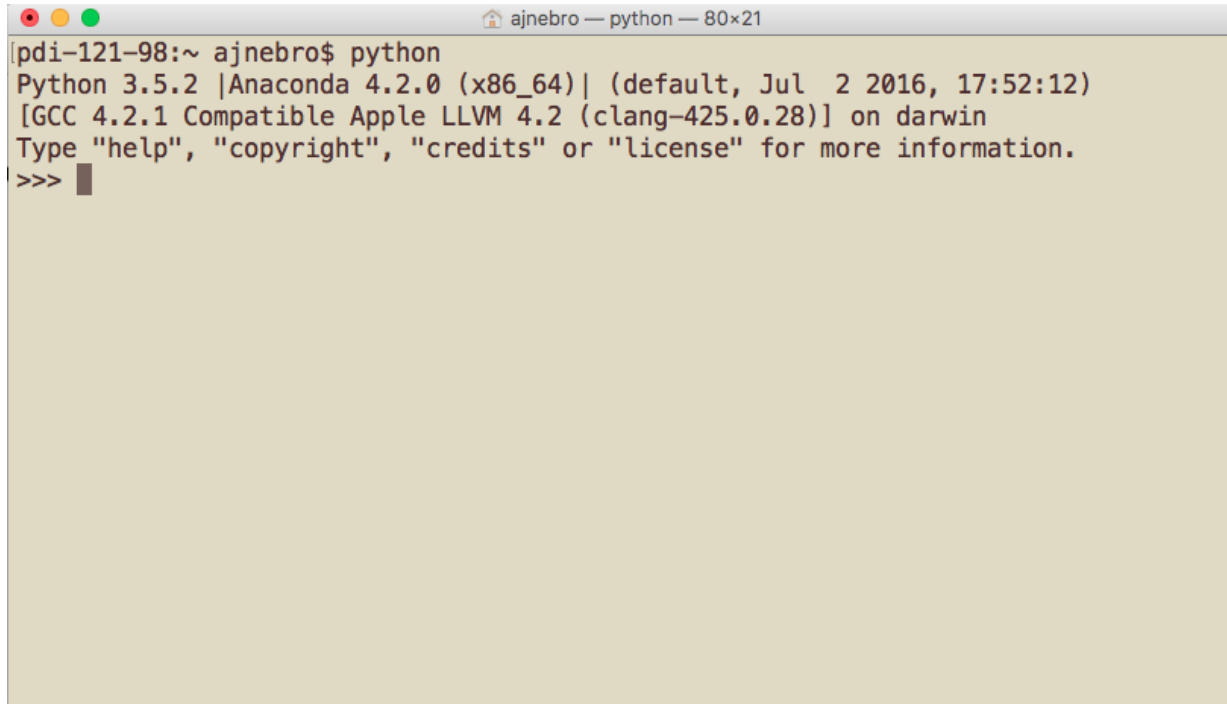
Es un lenguaje de programación de alta abstracción, interpretado, dinámico y de propósito general, ideal para el desarrollo rápido de aplicaciones.



```
holaMundo.py *  
1  if __name__ == "__main__":  
2      print('¡Hola mundo!')  
3
```

# Modo interactivo

- Se puede acceder al intérprete de Python mediante la consola:



```
ajnebro — python — 80x21
pdi-121-98:~ ajnebro$ python
Python 3.5.2 |Anaconda 4.2.0 (x86_64)| (default, Jul  2 2016, 17:52:12)
[GCC 4.2.1 Compatible Apple LLVM 4.2 (clang-425.0.28)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> █
```

# Indentación

```
public boolean esPrimo(int numero){  
    if (numero < 2){  
        return false;  
    }  
    int contador = 2;  
    boolean primo=true;  
    while ((primo) && (contador < numero)) {  
        if (numero % contador == 0) {  
            primo = false;  
        }  
        contador++;  
    }  
    return primo;  
}
```

JAVA

```
def es_primo(numero):  
    if numero < 2:  
        return False  
    contador = 2  
    primo = True  
    while primo and contador < numero:  
        if numero % contador == 0:  
            primo = False  
            contador += 1  
    return primo
```

PYTHON

**En Python, el ámbito de las funciones, clases, métodos, etc lo define la indentación o sangrado. Por tanto, no se utilizan llaves para definir el ámbito**

# Tipos de datos

- El tipo de una variable lo define el propio valor de la variable:

```
variable = 'cadena'           # Tipo cadena
variable = 5                   # Tipo entero
variable = 0.05                # Tipo real
variable = [1, 'cadena', 1.6]  # Lista
variable = (1, 'cadena', 1.6)  # Tupla
variable = {'x': 2, 'y': 1, 'z': 4} # Diccionario
variable = set([1, 2, "hola"])  # Conjunto
variable = None                # Ausencia de valor
type(variable)                # Para conocer el tipo de la variable
```

# Tipos de datos

- Tuplas
  - Pueden contener elementos de igual o distinto tipo
  - Son inmutables
  - Se declaran usando paréntesis ()

```
>>> tupla = (1, "Hola", 4, False, 55)          # Creacion
>>> tupla
(1, 'Hola', 4, False, 55)
>>> print(tupla[4])
55
>>> tupla[2] = 5                                # No se puede modificar
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tupla = (1, 2, 3) + (4, 5, 6)                # Operacion de concatenacion
>>> tupla
(1, 2, 3, 4, 5, 6)
>>>
```

# Tipos de datos

- Listas
  - Pueden contener elementos de igual o distinto tipo
  - Son mutables
  - Se declaran usando corchetes []

```
>>> lista = [3, "hola", False, 55] # Creacion
>>> lista
[3, 'hola', False, 55]
>>> print(lista[3]) # Acceso a un campo
55
>>> lista[0] = "cadena" # Modificacion
>>> print(lista[0])
cadena
>>> lista.append(502) # Añadir al final
>>> lista
['cadena', 'hola', False, 55, 502]
```



# Tipos de datos

- Diccionarios
  - Son listas de pares (clave, valor)
  - Son mutables
  - Se declaran usando llaves {}

```
>>> dict = {'clave':'dato', 'otra_clave':155}           # Creacion
>>> print(dict['otra_clave'])                           # Acceso
155
>>> dict['clave'] = "texto"                             # Modificacion
>>> print(dict['clave'])
texto
>>> dict['nueva_clave'] = 'nuevo_valor'                 # Extension
>>> print(dict['nueva_clave'])
nuevo_valor
>>> dict
{'otra_clave': 155, 'nueva_clave': 'nuevo_valor', 'clave': 'texto'}
>>> for key, value in dict.items():                       # Acceso claves y valores
...     print(key, value)
...
otra_clave 155
nueva_clave nuevo_valor
clave texto
```

# Tipos de datos

- Conjuntos
  - Contienen valores no repetidos
  - Son mutables
  - Admiten operaciones de conjuntos (unión, intersección, etc.)

```
>>> conjunto = set([1, 2, "hola"])          # Creacion
>>> conjunto
{1, 2, 'hola'}
>>> conjunto = set([1, 2, "hola", 1])        # Incluir duplicado (se ignora)
>>> conjunto
{1, 2, 'hola'}
>>> conjunto.add("adios")                   # Ampliar
>>> conjunto
{1, 2, 'hola', 'adios'}
>>> conjunto2 = set([1, "hola"])
>>> conjunto & conjunto2                    # Interseccion
{1, 'hola'}
>>> conjunto | conjunto2                    # Union
{1, 2, 'hola', 'adios'}
>>> conjunto - conjunto2                    # Diferencia
{2, 'adios'}
```

# Operadores aritméticos

- Suma: +
- Resta: -
- Multiplicación: \*
- División: / (float), // (integer)
- Resto: %

```
>>> print (3 + 2)
5
>>> print (3/2)
1.5
>>> print (3//2)
1
>>> print (3%2)
1
```

# Operadores lógicos

```
>>> edad = 15
>>> if edad >= 12 and edad <= 18:                                # And
...     print("Edad comprendida entre 12 y 18")
...
Edad comprendida entre 12 y 18
>>>
>>> x1 = 2
>>> x2 = 3.5
>>> if x1 < 5 or x2 < 5:                                          # Or
...     print("x1 o x2 son menores que 5")
...
x1 o x2 son menores que 5
>>> if not x1 > x2:                                              # Not
...     print("x1 no es mayor que x2")
...
x1 no es mayor que x2
```

# Expresiones

- `==` Igual a
- `!=` Distinto de (`<>` está obsoleto)
- `>` Mayor que
- `<` Menor que
- `>=` Mayor o igual que
- `<=` Menor o igual que
- `is` Igual a (solo para hacer comparaciones entre referencias de objetos o para saber si es `None`, `True` o `False`)

# Condiciones

```
>>> var = 100
>>> if var == 200:
...     print("var es 200")
... elif var == 150:
...     print("var es 150")
... elif var == 100:
...     print("var es 100")
... elif var is None:
...     print("var es None")
... else:
...     print("var no es ningún valor anterior")
...
var es 100
```

# Control de flujo

- Bucle For:

```
>>> colores = ["rojo", "azul", "verde"]
>>> for color in colores:
...     print("Color: ", color)
...
Color:  rojo
Color:  azul
Color:  verde
>>> numeros = (1, 2, 3)
>>> for numero in numeros:
...     print(numero)
...
1
2
3
```

- Bucle while

```
>>> colores = ["rojo", "azul", "verde"]
>>> count = 0
>>> while count < 3:
...     print("Color: ",
colores[count])
...     count += 1
...
Color:  rojo
Color:  azul
Color:  verde
```

- En Python no existen ni switch ni do while

# POO: Clases

Atributos  
(públicos)

Constructor

```
class Coche(object):  
    attr_clase = 'atributo estático o de clase'  
  
    def __init__(self, combustible, bateria):  
        self.combustible = combustible  
        self.bateria = bateria
```

```
    def repostar(self, combustible):  
        self.combustible += combustible
```

```
    def recargar(self, bateria):  
        self.bateria += bateria
```

Métodos

```
# Crear instancia  
coche = Coche(2, 5)
```

```
# Usar método  
coche.repostar(1)
```

```
# Mostrar combustible (atributo)  
print(coche.combustible)
```

```
# Acceder a atributo de clase  
print(Coche.attr_clase)
```



# POO: Atributos protegidos

```
class Coche(object):
```

```
    def __init__(self, combustible, bateria):  
        self._combustible = combustible  
        self._bateria = bateria
```

Los atributos que comienzan por un guión bajo son considerados protegidos. Solo se debería acceder dentro de la clase o las subclases

# Se puede acceder al atributo protegido combustible desde el exterior, pero no se debe. Cualquier IDE muestra un aviso de que tal acción no es adecuada

```
coche = Coche(12, 8)  
print(coche._combustible)
```

**Se puede acceder al atributo protegido exteriormente, pero no se debe**



# POO: Atributos privados

Los atributos que comienzan por dos  
guiones bajos son considerados privados.

```
class Coche(object):
```

```
    def __init__(self, combustible, bateria):  
        self.__combustible = combustible  
        self.__bateria = bateria
```

# NO se puede acceder al atributo  
privado combustible

```
coche = Coche(12, 8)
```

```
print(coche.__combustible)
```

**Error**



# POO: getters y setters

```
public class Coche {  
    private int combustible;  
    private int bateria;  
  
    public int getCombustible() {  
        return combustible;  
    }  
    public void setCombustible(int combustible) {  
        this.combustible = combustible;  
    }  
    public int getBateria() {  
        return bateria;  
    }  
    public void setBateria(int bateria) {  
        this.bateria = bateria;  
    }  
}
```

JAVA

```
class Coche(object):  
    def __init__(self, combustible, bateria):  
        self.combustible = combustible  
        self.bateria = bateria
```

**En Python, no definimos métodos *getters/setters*, sino que se accede directamente a los atributos, que por lo general se definen públicos.**

PYTHON

# POO: getters y setters

- Python sigue el **Uniform Access Principle**, el cual obliga a que el acceso a los atributos se deba hacer de una forma uniforme en todos los casos.
- Por tanto, si definimos métodos getter/setter incumplimos dicho principio, ya que tendríamos varias formas de acceder a los atributos dependiendo de si son públicos o privados:

```
coche = Coche(5, 7)  
print(coche.combustible) # forma de acceso común en Python  
print(coche.get_bateria()) # otra forma de acceso
```



# POO: getters, setters y properties

```
class Circulo(object):
```

```
    def __init__(self):  
        self.__radio = None
```

```
    @property
```

```
    def radio(self):  
        print('Accediendo a radio')  
        return self.__radio
```

```
    @radio.setter
```

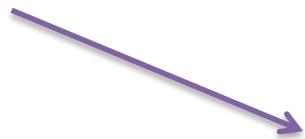
```
    def radio(self, radio):  
        if radio < 0:  
            raise ValueError("radio debe ser un numero no negativo")  
        self.__radio = radio
```

```
circulo = Circulo()  
circulo.radio = -1 # set  
print(circulo.radio) # get
```

Con *property* accedemos a los métodos getter/setter como si fueran accesos al atributo (no incumplimos el *Uniform Access Principle*). El uso de *property* es comparable a los métodos *getter/setter* en JAVA, pero en Python es más apropiado usarlo sólo cuando se requiera lógica en los accesos del atributo.

# POO: getters, setters y properties

```
class Circulo(object):  
    def __init__(self):  
        self.__radio = None  
  
    def __get_radio(self):  
        print('Accediendo a radio')  
        return self.__radio  
  
    def __set_radio(self, radio):  
        if radio < 0:  
            raise ValueError("radio debe ser un número no negativo")  
        self.__radio = radio  
  
radio = property(fget=__get_radio, fset=__set_radio)
```



Otra forma de definir property

# POO: Herencia

```
class A(object):  
    def __init__(self):  
        print('Soy constructor A')
```

La clase B hereda de A

```
class B(A):  
    def __init__(self):  
        print('Soy constructor B')  
        super(B, self).__init__()
```

Llama al constructor de la clase A

```
b = B()
```

```
# Resultado:  
Soy constructor B  
Soy constructor A
```

# POO: Herencia

```
class Animal(object):  
    def que_soy(self):  
        print("Soy Animal")
```

```
class Mamifero(Animal):  
    def que_soy(self):  
        print("Soy Mamifero")
```

```
class Sapiens(Animal):  
    def que_soy(self):  
        print("Soy Sapiens")
```

```
class Humano(Mamifero, Sapiens):  
    def que_soy(self):  
        super(Humano, self).que_soy()
```

```
humano = Humano()  
humano.que_soy()  
print("Orden de herencia:", Humano.__mro__)
```

## Herencia múltiple:

La clase *Humano* hereda de *Mamífero* y *Sapiens*

*super()* recorre el **MRO**, que es el orden de **herencia**, y delega en la primera clase que encuentra por encima de *Humano* que define el método *que\_soy()*.

Dicho orden da prioridad a los padres inmediatos antes que a los abuelos. Por tanto, si *Mamífero* no tuviera definido el método *que\_soy()*, se ejecutaría el método *que\_soy()* de *Sapiens*, y si ésta no lo tuviera, se ejecutaría el de la clase *Animal*, que es el abuelo.

Para ver el orden de herencia



# POO: Métodos estáticos

```
class Calculadora(object):  
    def __init__(self, modelo):  
        self.modelo = modelo  
  
    @staticmethod  
    def suma(x, y):  
        return x + y  
  
print(Calculadora.suma(2, 4))
```

**Definimos @staticmethod cuando el método no trabaja con los atributos, cuya lógica es propia de la clase, no de la instancia**

# POO: Métodos de clase

```
class Atomo(object):  
    def __init__(self, nombre):  
        self.nombre = nombre  
  
    @classmethod  
    def fision(cls):  
        h1 = cls("Isotopo primero")  
        h2 = cls("Isotopo segundo")  
        return h1, h2  
  
atomo = Atomo("Uranio")  
hijo1, hijo2 = atomo.fision()
```

Usamos **@classmethod** cuando queremos devolver objeto/s de la misma clase. Es como definir otro constructor, pero solo tenemos acceso a la clase, no al objeto.


Se usa **cls** en lugar de **self** por convención, ya que hacemos referencia a la clase, no al objeto

# POO: Sobrecarga de métodos

## JAVA

```
public class Raton {  
    String nombre;  
  
    public Raton() {  
        this.nombre = "";  
    }  
    public Raton(String nombre) {  
        this.nombre = nombre;  
    }  
}
```

## PYTHON

```
class Raton:  
    def __init__(self, nombre=""):   
        self.nombre = nombre
```

Valor por defecto

```
# Creamos instancias con y sin argumento  
raton = Raton('Perez')  
raton2 = Raton()
```

**En Python podemos simular la sobrecarga de métodos con el uso de valores por defecto en los argumentos**

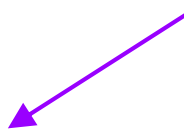
# POO: Interfaces

```
1 from abc import abstractmethod, ABCMeta
2
3 class ClassInterface(metaclass=ABCMeta):
4     @abstractmethod
5     def read(self, maxbytes = 1):
6         pass
7
8     @abstractmethod
9     def write(self, data: int):
10        pass
11
12
13 class ClassImplementation(ClassInterface):
14     def __init__(self):
15         pass
16
17     def read(self, maxbytes = 1):
18         print(maxbytes)
19
20     def write(self, data: int):
21         print(data)
22
23 if __name__ == '__main__':
24     a = ClassImplementation()
25     a.read()
26     a.write(23)
```

# POO: Excepciones

```
1 class Numero(object):
2     def __init__(self):
3         self.numero = -5
4
5     def comprobar_numero(self):
6         if self.numero < 0:
7             raise ValueError('El número es negativo')
8
9
10 if __name__ == '__main__':
11     try:
12         numero = Numero()
13         numero.comprobar_numero()
14     except ValueError:
15         print("Oops! Es un número negativo")
```

Lanza  
excepción



```
try:
    ...
except ArithmeticError as err:
    ...
except ValueError as err:
    ...
except Exception as err:
    ...
```

# POO: Excepciones

```
1 class TextoDemasiadoCortoError(ValueError):  
2     pass  
3  
4  
5 # función  
6 def validate(texto):  
7     if len(texto) < 10:  
8         raise TextoDemasiadoCortoError(texto)  
9  
10  
11 if __name__ == '__main__':  
12     validate(texto='hola')
```

← Excepción personalizada

# Funciones predefinidas

- Python cuenta con funciones predefinidas:
  - `min()`: devuelve el máximo de un iterable
  - `max()`: devuelve el máximo de un iterable
  - `round()`: redondea un número decimal
  - `isinstance()`: comprueba si un objeto es de una clase
  - `len()`: devuelve la longitud de una secuencia o colección
  - `int()`: convierte a entero
  - `str()`: convierte a cadena
  - `sorted()`: ordena un iterable, devolviendo una lista ordenada
  - ...

<https://docs.python.org/3.5/library/functions.html>

# Estilo de codificación

## JAVA

```
// Nombre de clase  
public class MotorTurbo {  
  
// Nombre de método  
public void escribirHola(){
```

## PYTHON (PEP8)

```
# Nombre de clase  
class MotorTurbo(object):  
  
// Nombre de método  
def escribir_hola(self):
```

**Python usa PEP8 como estilo de código.  
Hay muchas más normas de estilo en  
PEP8: espacios entre los métodos, entre  
los argumentos, etc**



# Gestion de dependencias

- Las dependencias en Python se gestionan con el comando pip

Buscar dependencia

```
pip search dependencia
```

Instalar dependencia

```
pip install dependencia
```

Desinstalar dependencia

```
pip uninstall dependencia
```

Generar fichero *requirements.txt* con todas las dependencias instaladas

```
pip freeze > requirements.txt
```

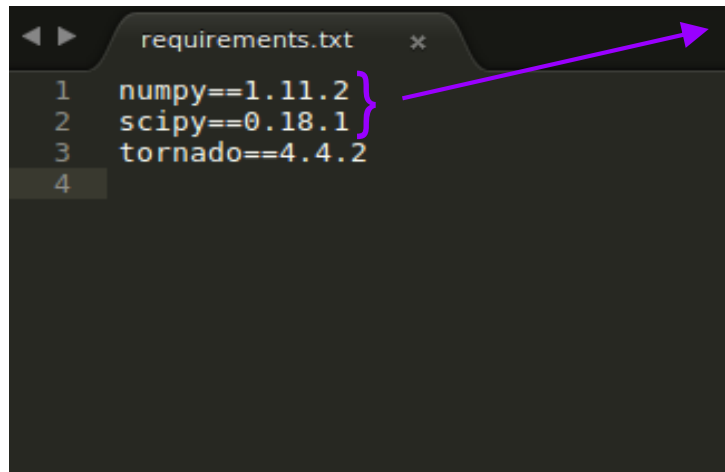
Instalar todas las dependencias definidas en *requirements.txt*

```
pip install -r requirements.txt
```

**El uso de un IDE como *Pycharm* permite realizar estas operaciones a través de una interfaz gráfica**

# Gestion de dependencias

## Ejemplo de *requirements.txt*



```
requirements.txt
1 numpy==1.11.2
2 scipy==0.18.1
3 tornado==4.4.2
4
```

### Dependencias:

- *numpy*: módulo para cómputo científico
- *scipy*: módulo para matemáticas, ciencia e ingeniería
- *tornado*: módulo para conexiones asíncronas

# Gestion de dependencias

- Si se usa el intérprete python del sistema, todas las dependencias que se instalen o eliminen se hacen globalmente, en el propio sistema local
- Para evitar esto, se suele usar virtualenv, que permite crear un entorno virtual para cada proyecto, incorporando un intérprete y la posibilidad de instalar dependencias particulares para para el proyecto

# Gestion de dependencias

Crear entorno virtual con el nombre de directorio *venv*

```
virtualenv venv --distribute
```

Acceder al entorno virtual

```
source venv/bin/activate
```

**Una vez dentro del entorno virtual, todas las dependencias que instalemos con *pip* se instalarán dentro del entorno virtual.** De igual forma, si generamos el fichero *requirements.txt*, éste tendrá declarado todas las dependencias instaladas en el entorno virtual

**El uso de un IDE como *Pycharm* permite realizar estas operaciones a través de una interfaz gráfica**

# Pruebas unitarias

```
class Triangulo(object):

    def __init__(self, lado1=None, lado2=None,
                 lado3=None):
        self.lado1 = lado1
        self.lado2 = lado2
        self.lado3 = lado3

    def es_equiletero(self):
        if self.lado1 is None or self.lado1 is None or \
            self.lado1 is None:
            return False
        elif self.lado1 == self.lado2 and \
            self.lado2 == self.lado3:
            return True
        else:
            return False
```

```
import unittest
```

Librería *unittest* para  
hacer pruebas unitarias

```
class TrianguloTestCase(unittest.TestCase):
```

```
# Se ejecuta justo antes de cada test
```

```
def setUp(self):
```

```
    print("setUp: INICIANDO TEST")
```

```
# Instanciamos de la clase Triangulo
```

```
    self.triangulo = Triangulo()
```

*setUp* se ejecuta antes  
de cada método test

```
# Se ejecuta despues de cada test
```

```
def tearDown(self):
```

```
    print("tearDown: FINALIZANDO TEST")
```

*tearDown* se ejecuta  
después de cada método  
test

```
def test_es_equilatero(self):
```

```
    print("Ejecutando test1")
```

```
    self.triangulo.lado1 = 4
```

```
    self.triangulo.lado2 = 4
```

```
    self.triangulo.lado3 = 8
```

```
    resultado = self.triangulo.es_equiletero()
```

```
    self.assertFalse(resultado)
```

El nombre de cada  
método test debe  
comenzar con *test*

# Pruebas unitarias

- Módulos para testing
  - unittest: para crear los tests
  - Nose: para ejecutar tests de forma automática
  - Coverage: para la medición de la cobertura de código
  - Mock: para crear objetos mocks