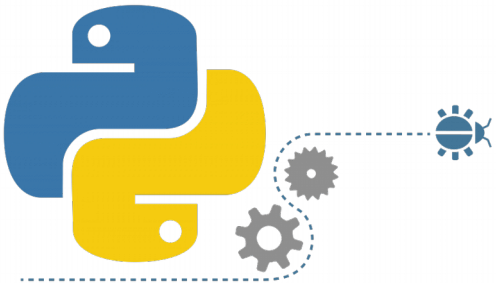


# From Nothing to Python

# Contacto y grupos interesantes

t.me/cyberh99

DevUco Group and projects :-)

https://t.me/joinchat/EWwrOk34lP-Qit7TLHoutg

Aula Software Libre

https://t.me/AulaSoftwareLibreUCO

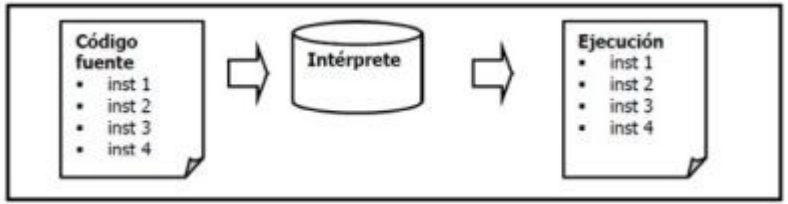
# Lenguajes interpretados vs Lenguajes compilados

Código fuente

```
#include <iostream>
using namespace std;

int main() {
        cout<<"Hola mundo";
        return 0;
}
```


Código máquina

```
100101101010
001001011110
101001000101
010100101010
101010010101
010101010101
000000110010
```



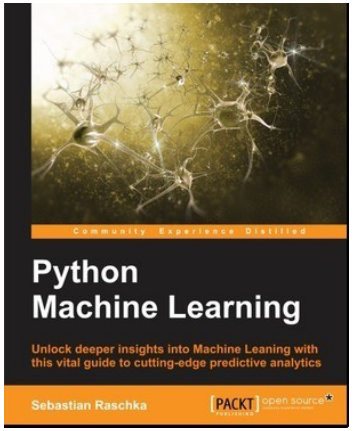
# Ventajas de lenguajes interpretados

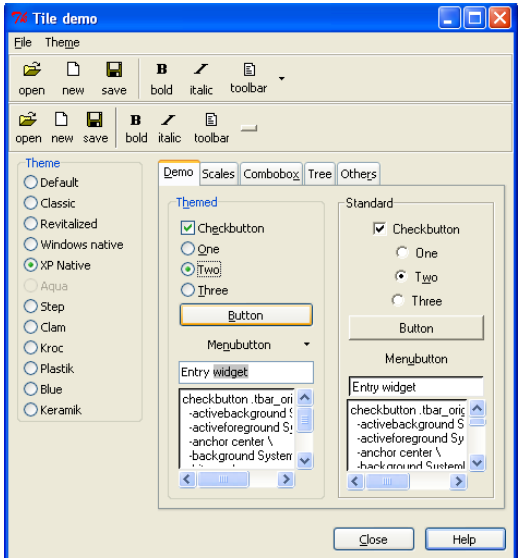
- Pueden ser ejecutados en cualquier plataforma.
- Ocupan menos espacio en la memoria.
- El framework es el que se encarga de que el hardware ejecute las instrucciones.
- Las variables de datos son dinámicas y no se restringen a un solo tipo.
- Son más utilizados en desarrollo web y en electrónica.

# Desventajas de lenguajes interpretados

- Su ejecución es más lenta con respecto a los lenguajes compilados.
- Son más difíciles de debuggear.
- Requieren de una máquina virtual que sirva como intérprete las instrucciones y el procesador.
- No todas las máquinas virtuales están disponibles para todas las plataformas.

# Python for all




# Instalando Python

From terminal (Linux system):

- apt install python

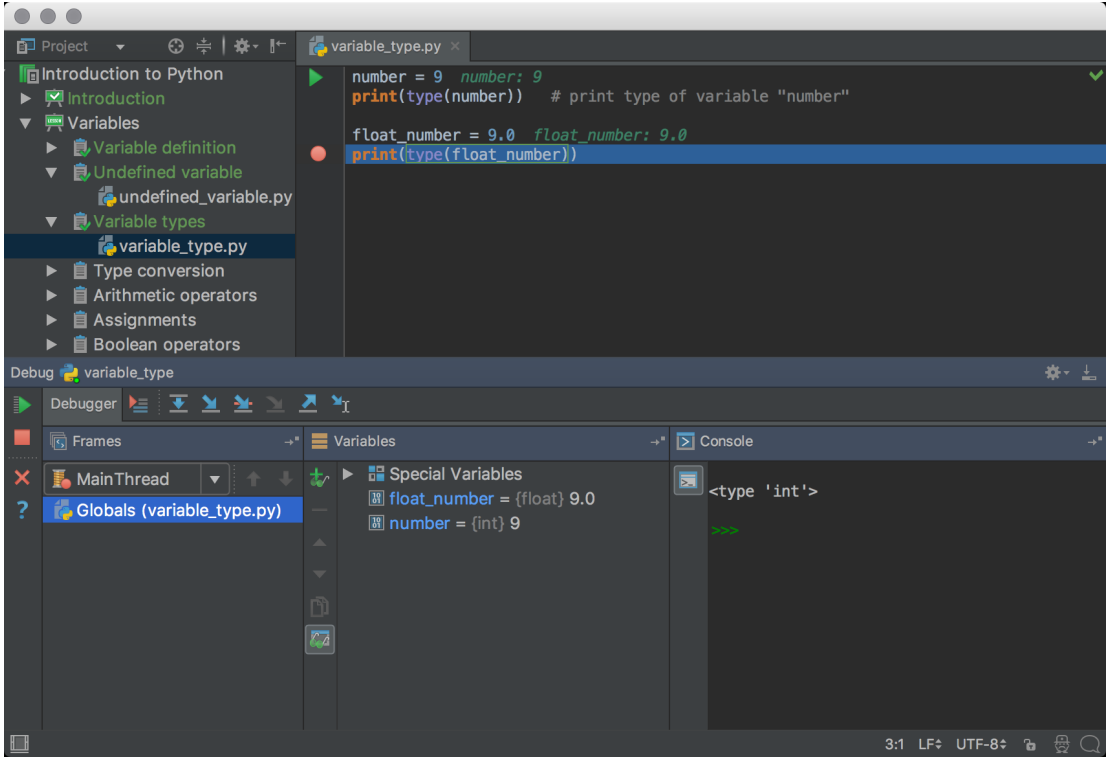
From Windows:

- https://www.python.org/

# Características de Python

- Python es orientado a objetos, cualquier cosa en python es un objeto (incluyendo listas, diccionarios)
- Lenguaje de alto nivel
- Lenguaje interpretado ( aunque puede ser compilado con ayuda de módulos externos)
- Disponibilidad de librerias de forma extensa
- Open Source

# IDE para python



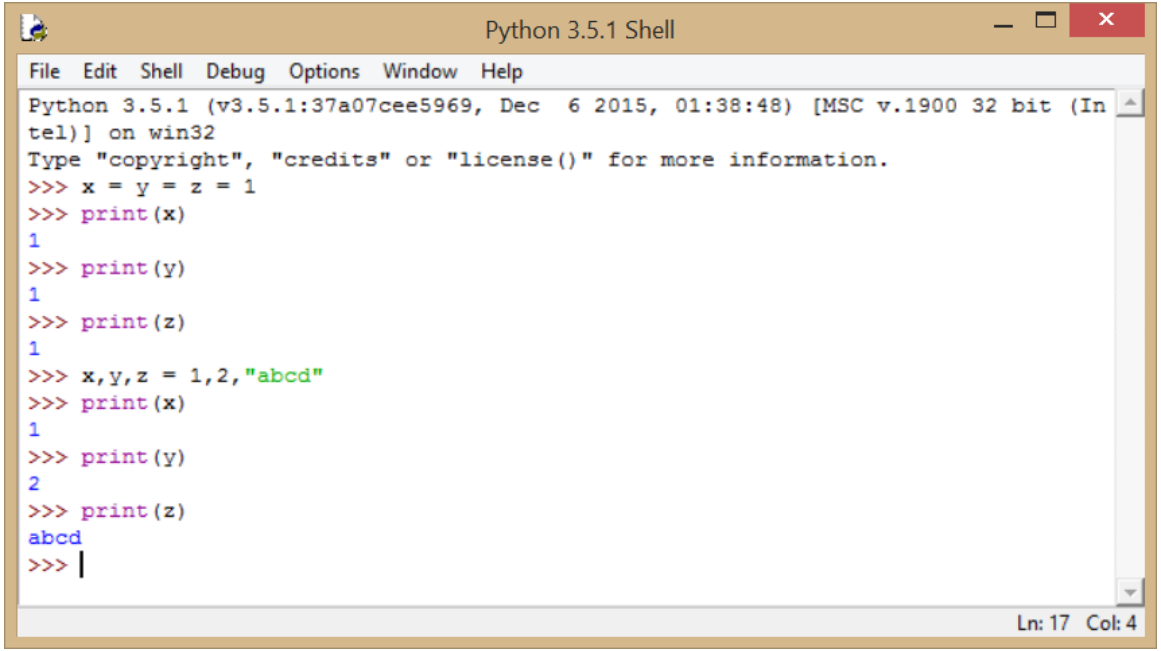

https://www.jetbrains.com/pycharm/

# Tipos de variables

- Int
- Long
- Float
- Bool
- string
- Unicode

- Listas
- Tuplas
- Diccionarios

- type(variable)

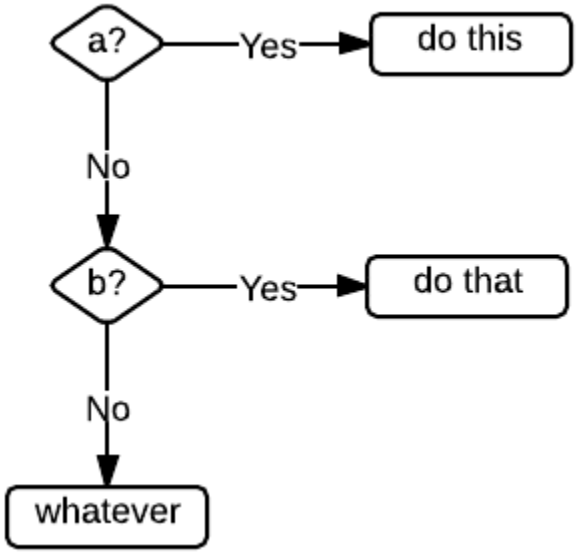


# Listas / Tuplas / Diccionarios

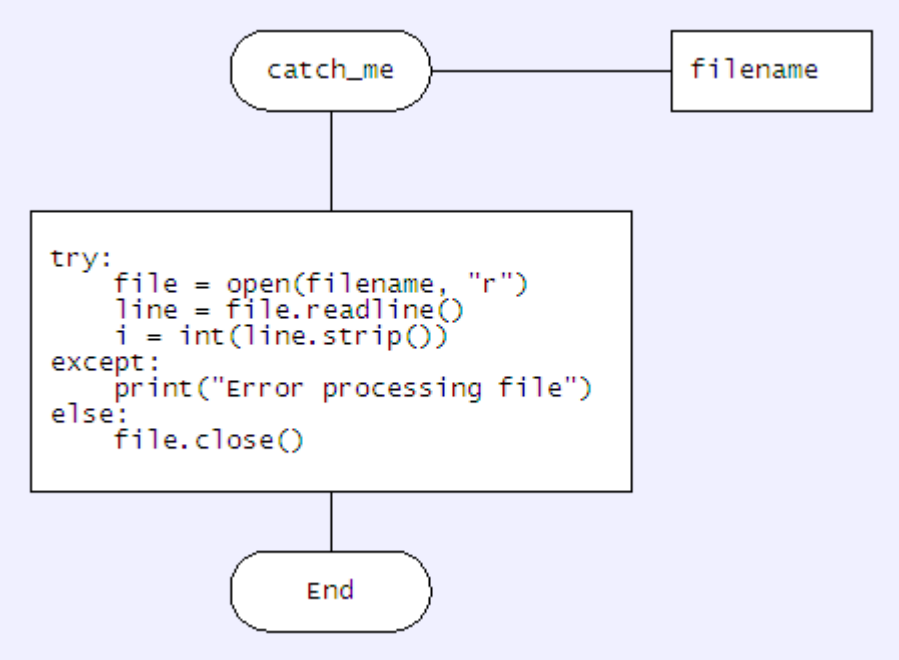
- ★ Listas:
  - ○ Almacenan datos de mismo (o diferente) tipo
  - ○ Los datos pueden ser modificados sobre la marcha
  - ○ lista = [“asd”,2,3,True]
- ★ Tuplas:
  - ○ Almacenan datos de mismo (o diferente tipo)
  - ○ Los datos NO pueden modificarse sobre la marcha
  - ○ tupla = (“AD”,2,3,False)
- ★ Diccionario:
  - ○ Almacenan valores clave:valor
  - ○ diccionario ={“hola”:”pepe”, “Adiós”:”tito”}

# If / else / elif

```
if a:
    do this
elif b:
    do that
else:
    whatever
```



# Try / Except


catch_me
filename
try:
file = open(filename, "r")
line = file.readline()
i = int(line.strip())
except:
print("Error processing file")
else:
file.close()
End

# For Loop

```
string = "Hello World"
for x in string:
    print x
```

```
collection = ['hey', 5, 'd']
for x in collection:
    print x
```

```
list_of_lists = [ [1, 2, 3], [4, 5, 6], [7, 8, 9]]
for list in list_of_lists:
    for x in list:
        print x
```

# While Loop

```
n = raw_input("Please enter 'hello':")
while n.strip() != 'hello':
    n = raw_input("Please enter 'hello':")
```

```
while True:
    n = raw_input("Please enter 'hello':")
    if n.strip() == 'hello':
        break
```

# ¿Pointers on python?

34

> I want `form.data['field']` and `form.field.value` to always have the same value

This is feasible, because it involves decorated names and indexing -- i.e., **completely** different constructs from the **barenames** `a` and `b` that you're asking about, and for with your request is utterly impossible. Why ask for something impossible **and** totally different from the (possible) thing you actually *want*?!

Maybe you don't realize how drastically different barenames and decorated names are. When you refer to a barename `a`, you're getting exactly the object `a` was last bound to in this scope (or an exception if it wasn't bound in this scope) -- this is such a deep and fundamental aspect of Python that it can't possibly be subverted. When you refer to a *decorated* name `x.y`, you're asking an object (the object `x` refers to) to please supply "the `y` attribute" -- and in response to that request, the object can perform totally arbitrary computations (and indexing is quite similar: it also allows arbitrary computations to be performed in response).

Now, your "actual desiderata" example is mysterious because in each case two levels of indexing or attribute-getting are involved, so the subtlety you crave could be introduced in many ways. What other attributes is `form.field` suppose to have, for example, besides `value`? Without that further `.value` computations, possibilities would include:

https://stackoverflow.com/questions/3106689/pointers-in-python

# Comentarios

```
5  #asad
6
7  '''
8  asd
9  asd
10 '''
```

#!/usr/bin/env python

#-*- coding: utf-8 -*-

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

print "Hello World"
```

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-

class Hello:
    def __init__(self,nombre):
        self.nombre = nombre

    def say_hello(self):
        print "Hello %s"%(self.nombre)

Hello("Hector").say_hello()
```

```
#include <iostream>


int  main(int argc, char const *argv[]) {
  /* code */
```

```
#include <iostream>


class hello{

public:
  void say_hello(){
    std::cout<<"Hello World \n";
  }
};

int  main(int argc, char const *argv[]) {
  /* code */

  hello h = hello();
  h.say_hello();

  return 0;
}
```

# Strings Stuff

## Accediendo al contenido

- Accediendo por índices:
  - string[0]
- Accediendo rangos:
  - string[0:6]
- A la inversa:
  - string[-2]

## Strip y Split

split(“.”) → Separa las palabras según un carácter

strip(“0”) → Elimina el carácter especificado de la cadena

## Contenido dinámico

```
nombre = raw_input()

saludos = “Buenas tardes %s” %nombre
```

| Conversion | Meaning |
|---|---|
| 'd' | Signed integer decimal. |
| 'i' | Signed integer decimal. |
| 'o' | Signed octal value. |
| 'u' | Obsolete type – it is identical to 'd'. |
| 'x' | Signed hexadecimal (lowercase). |
| 'X' | Signed hexadecimal (uppercase). |
| 'e' | Floating point exponential format (lowercase). |
| 'E' | Floating point exponential format (uppercase). |
| 'f' | Floating point decimal format. |
| 'F' | Floating point decimal format. |
| 'g' | Floating point format. Uses lowercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise. |
| 'G' | Floating point format. Uses uppercase exponential format if exponent is less than -4 or not less than precision, decimal format otherwise. |
| 'c' | Single character (accepts integer or single character string). |
| 'r' | String (converts any Python object using repr()). |
| 's' | String (converts any Python object using str()). |
| '%' | No argument is converted, results in a '%' character in the result. |

https://docs.python.org/2.7/library/stdtypes.html#string-formatting-operations

# Listas Stuff

- Midiendo tamaños de listas:
  - len(lista)
- Accediendo valores de la lista:
  - Lista[0]
- Agregando valores de la lista:
  - lista[1] = “Python”
- Agregando elementos al final de la lista:
  - lista.append(variable)
- Eliminando elementos de la lista:
  - del lista[9]

# Diccionarios Stuff

- Listando las claves de los diccionarios:
  - diccionario.keys()
- Accediendo a los valores de diccionarios:
  - diccionario[key]
- [RE]Asignando valores:
  - diccionario[key] = “Adios”
- Obteniendo la longitud de los diccionarios:
  - len(diccionario)

# Let’s do the terminal stuff

# I don’t know what to do

- Pedir entrada por teclado hasta que el usuario introduzca la palabra exit.
- Pedir una serie de datos por teclado y almacenarlos en una lista. Imprimiendo esta última antes de salir del programa.
- Crear un programa que lleve el registro del nombre, apellidos y peso de los alumnos de un instituto.

# Funciones

```
#!/usr/bin/env python
#-*- coding:utf-8 -*-


def ooo_some_code():
    print ("Hello World \n")


ooo_some_code()
```

```
#!/usr/bin/env python
# -*- coding:utf-8 -*-


name=raw_input("¿Cual es tu nombre?")

def say_hello(nombre):
    print ("Hello %s"%(nombre))

say_hello(name)
```

# Variables / For / listas / diccionarios/ TODO JUNTO :-)

Crea un programa que imprima los datos de nombre de los alumnos con sus índices de masa corporal. Este programa tiene que incluir:

- ➔ Utilizar funciones
- ➔ Utilizar diccionarios

## Mr Chicken as a class

```
class Chicken(object):
        "Describes a chicken."
        def __init__(self, name):
                self.name = name
        def make_sounds(self):
                "Print chicken sound."
                print "squeaaaak!"
```

# POO


CLASE
Propiedad y comportamiento de un objeto concreto
ATRIBUTO
Propiedad del objeto
MÉTODO
Lo que un objeto puede hacer (algoritmo)
OBJETO
Instancia de una clase
MENSAJE
Comunicación dirigida a un objeto ordenándole que ejecute uno de sus métodos

# Clases, Objetos etc...

```
class dog():

    '''code ''
```

A. Definición de una clase en Python

```
class dog():
    def new_dog(self):
        self.born()
        self.breathe()
```

1. Definición de funciones dentro de clases (método)

```
c = dog()

c.new_dog()
```

I. Llamada a la clase(objeto) y a una función dentro de la misma

# Our friend *self*

show 10 more comments

18 Answers

active oldest votes

535

The reason you need to use `self.` is because Python does not use the `@` syntax to refer to instance attributes. Python decided to do methods in a way that makes the instance to which the method belongs be *passed* automatically, but not *received* automatically: the first parameter of methods is the instance the method is called on. That makes methods entirely the same as functions, and leaves the actual name to use up to you (although `self` is the convention, and people will generally frown at you when you use something else.) `self` is not special to the code, it's just another object.

Python could have done something else to distinguish normal names from attributes -- special syntax like Ruby has, or requiring declarations like C++ and Java do, or perhaps something yet more different -- but it didn't. Python's all for making things explicit, making it obvious what's what, and although it doesn't do it entirely everywhere, it does do it for instance attributes. That's why assigning to an instance attribute needs to know what instance to assign to, and that's why it needs `self.`.

| Sr.No. | Method, Description & Sample Call |
|---|---|
| 1 | **__init__ ( self [,args...] )**<br>Constructor (with any optional arguments)<br>Sample Call : *obj = className(args)* |
| 2 | **__del__( self )**<br>Destructor, deletes an object<br>Sample Call : *del obj* |
| 3 | **__repr__( self )**<br>Evaluable string representation<br>Sample Call : *repr(obj)* |
| 4 | **__str__( self )**<br>Printable string representation<br>Sample Call : *str(obj)* |
| 5 | **__cmp__ ( self, x )**<br>Object comparison<br>Sample Call : *cmp(obj, x)* |

```python
#!/usr/bin/env python
#-*- coding: utf-8 -*-

class Simple_class:

    homer =  "ouch"

    def __init__(self, nombre):
        self.nombre = nombre
        print "Obiously it's a simple class"

    def say_hello(self):
        print "Hello %s"%(self.nombre)

    def say_goodbye(self):
        print "Goodbye %s"%(self.nombre)

    def flirt(self):
        self.say_hello()
        self.say_goodbye()
        print self.homer

objet = Simple_class("tito")
objet.flirt()
```

# Herencia

La herencia nos permite que una clase herede las variables y métodos a varias subclases.

Estas subclases, aparte de los métodos propios, tiene incorporados los atributos y métodos heredados de la anterior clase (superclase)

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

class Animal:
    def __init__(self):

        print "It's an animal"

    def born(self):
        print "It's borning a new animal"

    def breathe(self):
        print "It's breathing"

class dog(Animal):
    def new_dog(self):
        self.born()
        self.breathe()

c = dog()

c.new_dog()

```

# Poliformismo

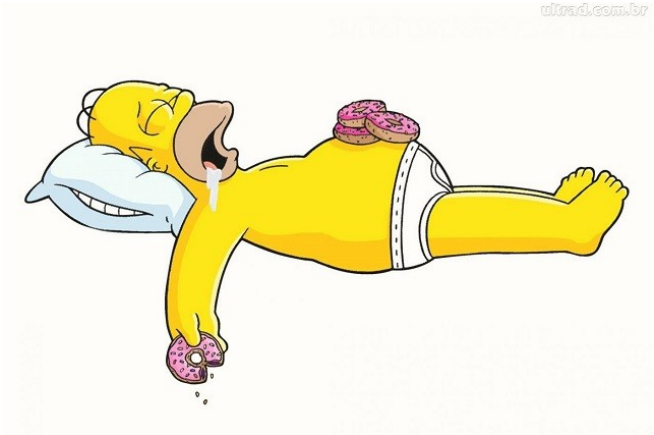
En el ámbito de la orientación a objetos el polimorfismo hace referencia a la habilidad que tienen los objetos de diferentes clases a responder a métodos con el mismo nombre, pero con implementaciones diferentes.

```python
#!/usr/bin/env python
#-*- coding:utf-8 -*-

class Animal:
    def __init__(self):

        print "It's an animal"

    def born(self):
        print "It's borning a new animal"

    def breathe(self):
        print "It's breathing"

class dog(Animal):
    def __init__(self):
        self.born()
        self.breathe()
        print "It's a dog :-) "

    def look_a_cat(self):
        print "The dog try to catch the cat"

class cat(Animal):
    def __init__(self):
        self.born()
        self.breathe()
        print "It's a cat"

    def look_a_cat(self):
        print "Tha cat fell in 'love'"

rayo = dog()
rayo.look_a_cat()
bigotes = cat()
bigotes.look_a_cat()
```

# Encapsulación

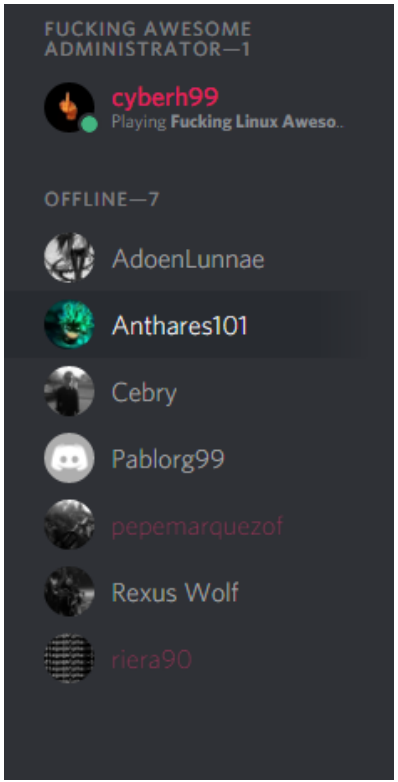
La encapsulación nos permite crear funciones privadas dentro de nuestras clases, de forma que no pueden ser accesibles desde el exterior de dicha clase.

```
class Fecha
{
   public:
      /* Comentado por ineficiente
      int anho;
      int mes;
      int dia;   */

       long numeroSegundos;

       void metodoMaravilloso1();
       void metodoMaravilloso2();
 };
```



```python
#!/usr/bin/env python
#-*- coding: utf-8 -*-


class super_calculator:
    def __init__(self,num1,num2):
        print "It's too simple"
        self.num1 = num1
        self.num2 = num2

    def __mult(self):

        f_number = self.num1 * self.num2
        return f_number

    def operate(self):
        print self.__mult()


casio = super_calculator(2,2)
casio.operate()
```

# You Think something




# Jurassic park python's version

```
## Animals_2a3.py -- a complete OO zoo !!              DMQ 6/10/04
'''
# Animal    -->    Reptile
# -------          -------
# Animal    -->    Mammal    -->    Bovine
# -------          -------          -------
# Animal    -->    Mammal    -->    Canine
# -------          -------          ------
# Animal    -->    Mammal    -->    Feline    -->    Cat
# -------          -------          -------          -------
# _numAnimals      _numMammals      _numFelines      _numCats
# home                              genus
# __init__()       __init__()       __init__()       __init__()
#                    .sound                             .sound
#                                                     .name
# show()             show()           show()           show()
#                  talk()                             talk()
#
'''
```

# Compilando archivos .py


Pyinstaller ‹script.py›

# Bibliografia recomendada


- Python Machine Learning → http://books.tarsoit.com/Python%20Machine%20Learning.pdf
- Python for Linux and Unix Administration → http://docs.linuxtone.org/ebooks/Python/OReilly.Python.for.Unix.and.Linux.System.Administration.Sep.2008.pdf
- Web Development with Django cookbook → https://doc.lagout.org/programmation/Django/Web%20Development%20with%20Django%20Cookbook%20%5BBendoraitis%202014-11-17%5D.pdf
- Documentación oficial → https://docs.python.org/3/

# Relax.. Last diapo