CURSO BÁSICO PYTHON 3.7

python™

BÁSICO

Francisco Andrés Hernández
pacoan@unex.es
Robótica y visión artificial

CONTENIDOS

- Objetivos de esta charla
- Introducción
- Características del lenguaje
- Versiones e implementaciones
- Elementos de un programa
- Tipos de datos básicos y operadores
- Colecciones
- Control de flujo e iterables
- Funciones
- Módulos
- Clases

Objetivos de esta charla

- Esto NO es un curso de programación
- Si ya tienes experiencia en lenguajes (C++, Java).....
- ¿Buenas prácticas para programar en Python?
- ¿Cual es su filosofía ?
  - Cualquiera puede escribir Python pero...
  - Otra cosa es escribir código efectivo, simple.....

  Pythonico

Introducción

- Creado por Guido van Rossum en 1989...
- Está instalado en Linux desde sus inicios
- En los últimos años han aumentado su adeptos
- Es algo más lento que C++ (0.8% pérdida)
- Usado en muchos ámbitos de la informática
    - Administración de sistemas
    - Ámbito científico
    - Deep learning, robótica

Es un lenguaje pegamento

Dos grandes principios: legibilidad y transparencia

Características

- Es interpretado
- Hace hincapié en una sintaxis que genere código legible
- Multiplataforma
- Multiparadigma:
  - Programación orientada a objetos
  - Programación imperativa
  - Programación funcional
- Tipado dinámico
- Extensas librerías para todo. (pip3)
- Regla: Todo es un objeto. type(obj)
- Gramática limpia:
  - No hay aperturas y cierres de bloque
  - Código totalmente indentado (4 espacios)
  - Guía de estilo para python PEP8

```
>>> a=25
>>> type(a)
<type 'int'>
>>> b="paco andres"
>>> type(b)
<type 'str'>
```

Versiones

- Hay dos versiones del lenguaje:
  - Python 2.7
  - Python 3.x

- Es preferible usar 3.x siempre que se pueda.

- Usa python 3.x:
  - Soporte oficial de 2.7 termina en 2020
  - Más packages integrados de serie
  - La inmensa mayoría de los paquetes están para 3.x
  - No hay problemas de codificación de texto UTF8

    "# -*- coding: utf-8 -*-"
  - Fácil de aprender

Implementaciones

- El intérprete genera bytecode para ejecutar. Más rápido.
- Cuando hablamos de implementaciones nos referimos principalmente a la forma de ejecutar el intérprete de python:
  - **Cpython (C nativo), Jpython (java) , Ironpython (C#)**

  - **Pypy**
    - Rpython:
      - python con tipos estáticos
      - Reduced Python
    - JIT:just in time compiler
    - Mayor velocidad de ejecución. 6x
    - Evita el GIL (global interpreter lock).
    - stackless. podemos ejecutar en modo stackless, proporcionando microhilos de concurrencia masiva

  - **Micropython**
    - Microcontroladores con python. IOT (esp32, teensy,esp8266)

Elementos de un programa

➢Dos formas de usarlo:

- Usando directamente el intérprete. Comando Python consola >>>
- Generando un fichero Python. fichero.py

➢Anaconda (gestión de entornos Python con herramientas)

- Notebooks (python en navegador)

➢Un programa contiene:

- Palabras reservadas (keywords)
- Funciones integradas (built-in functions)
- Literales
- Operadores
- Delimitadores de expresiones
- Identificadores

```
Radio = 5
for x in range(100):
   area = 3.14159265358979323846 * Radio ** 2
   print(area)
```

Elementos de un programa

- Palabras reservadas:
  - Forman parte del núcleo del lenguaje
  - No pueden usarse para nombrar otros elementos

False	class	finally	is	return
None	continue	for	lambda	try
True	def	from	nonlocal	while
and	del	global	not	with
as	elif	if	or	yield
assert	else	import	pass	
break	except	in	raise	

- Funciones integradas de forma predeterminada:

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Elementos de un programa

- literales:
  - Datos simples
  - Números: valores lógicos, enteros ,decimales,complejos…
  - Cadenas de texto

- Operadores:

```
+     -     *     **     /     //     %     @
<<    >>    &     |      ^     ~
<     >     <=    >=     ==    !=
```

- Delimitadores (delimitar,separar expresiones):

```
'     "     #     \
(     )     [     ]     {     }
,     :     .     ;     =     ->
+=    -=    *=    /=    //=   %=    @=
&=    |=    ^=    >>=   <<=   **=
```

- Identificadores:
  - Variable, objetos creados por el usuario
  - Primer carácter una letra

Tipos de datos básicos

- Comentarios:

  - Una línea:

    ```
    # esto es un comentario
    ```

  - Varias líneas o documentación:

    ```
    """ esto es
      un comentario de varias línea
    """

    def funcion():
        """ esto es la doc de función"""
    ```

Tipos de datos básicos

- Operaciones matemáticas básicas
- Divisiones enteras con el operador //
- Divisiones en coma flotante con el operador /

```
a=10
b=.12
c= 5
print(a+b)
print(a-c)
print(a/c)
print(a//c)

10.12
5
2.0
2
```

Tipos de datos básicos

- El tipo booleano es una primitiva
  - *True*
  - *False*

- Operadores booleanos
  - *and*
  - *or*
  - *not*

```
a=True
b=False
print(a and b) →  False
print(a or b)  → True
```

- 0 es False
- 1 es True (sólo 1)

- La igualdad es ==
- La desigualdad !=

```
cero = 0
uno =1

print(cero == False) → True
print(uno == True) → True
print(2 == True) → False
```

Tipos de datos básicos

- **<, >, =, >=, <=** son operadores booleanos
- Las comparaciones pueden ser encadenadas

```
a=1
b=2
c=4

print(a>b)  → False
print(a<=c) → True
print(a<b<c) → True
```

- El operador **is** se usa para comparar objetos
- No confundir con **==**
- **Is** se usa para determinar si dos objetos se refieren al mismo objeto (apuntan)
- **==** para comparar el contenido

```
a=1
b=2
c=a
d=b
print (id(a),id(b),id(c),id(d))

print(a is b)
print(a is c)
print(a is d)
print(a is not b)

10910400 10910432 10910400 10910432
False
True
False
True
```

Tipos de datos básicos

- **None** se usa en conjunción con **is** para evaluar objetos
- **None** es de tipo (NoneType)
- se puede asignar pero no crear objetos NoneType
- No es 0, False ni cadena vacía
- **NO** es recomendable usarlo con == para hacer comparaciones

```
a=None

print (type(a))
print(a is None)

<class 'NoneType'>
True
```

- Sirve para asignar una variable si no sabemos de antemano qué tipo va a tener.
- Como variable comodín para auto-asignarle un valor.

Tipos de datos básicos

- Los string se definen con “ o con ‘
- Si son literales:
  - Pueden concatenarse con + o con espacio
  - Lo mejor es usar esta segunda opción

```
a="hola" " mundo"
b="hola "+"mundo"

c=' esto es "hola mundo" cadena'


print(a)
print(b)
print(c)
print(len(c))
print(c[9])


hola mundo
hola mundo
esto es "hola mundo" cadena
27
“
```

- Son listas (vectores) en sí mismos
- Sobre strings hay multitud de operaciones que podemos aplicar.

Tipos de datos básicos

- Hay diferentes formas de formatear strings:

```
print("Número de msg: {}. Tipo {}".format(10,"entero"))

print("la {0} de color {1} es {1}".format("casa","azul") )

print("la {ob} de color {col} es {col}".format(col="azul",ob="azul") )

Número de msg: 10. Tipo entero
la casa de color azul es azul
la azul de color azul es azul
```

Tipos de datos básicos

- La función **print** se usa para escribir por terminal:
- Para pedir datos al usuario usamos **input**:

```
print("esto es PYTHON para todos")
print("esto es PYTHON para todos",end="-->")
print("fin")

a=input("dame un número:")
```

```
esto es PYTHON para todos
esto es PYTHON para todos-->fin
dame un número:
```

Tipos de datos básicos

- Si hablamos de variables:
  - No hay declaración de tipo
  - Podemos asignarla con diferentes tipos
  - No podemos usarla si al menos no hay una asignación

```
var=10
print(var)
var="pepe"
print(var)
print(a)

10
pepe
Traceback (most recent call last):
 File "/home/paco/PycharmProjects/untitled/paco.py", line 7, in <module>
   print(a)
NameError: name 'a' is not defined
```

- Podemos usar como en otros lenguaje operadores ternarios basados en una condición.

```
var="pepe"
print(var)

a= "manuel" if var=="pepe" else 3
print(a)
```

Tipos de datos básicos

- Podemos hacer asignaciones múltiples:

```
A,B = 1,2
print(A,B)

B,A = A,B
print(A,B)

1 2
2 1
```

Colecciones

- Aparte de los tipos simples que hemos visto, python cuenta con colecciones de objetos:

  - Listas. Almacenan colecciones de objetos [ ]
  - Tuplas. Almacenan colecciones de objetos (inmutables) ( )
  - Set. Colecciones de objetos distintos. { }
  - Diccionarios. Colecciones de objetos asociados a una clave. {k:v}

- NO hay más tipos estructurados.

- Recordar : En Python todo es un objeto.
- Casi todas las combinaciones son válidas

¡ genial !

Listas

Las listas almacenan colecciones de objetos heterogéneos.
Todo en python son objetos

```
lis=[]
lis1=[1,2,3,4]
mat=[lis,lis1]
print(lis)
print(lis1)
print(mat)

[]
[1, 2, 3, 4]
[[], [1, 2, 3, 4]]
```

Listas

- Operaciones de inserción y borrado:

```
lista=[1,2,3,4] #crear lista

lista.append(5) #insertar valores al final

lista.append(10) # [1, 2, 3, 4, 5, 10]

lista.pop()      # recuperar valores y borrarlos[1, 2, 3, 4, 5]

del(lista[2])    # borrar un valor determinado [1, 2, 4, 5]

lista.remove(2)  # borrar una ocurrencia concreta [1, 4, 5]

lista.remove(2)   #  error

lista.insert(0,30) # insertar en una posición un valor [30, 1, 4, 5]

lista.extend(["e","d"]) # concatenar dos listas [30, 1, 4, 5, 'e', 'd']

print(lista)
```

Listas

La forma de acceder a partes o trozos de una lista se denomina **slicing**
lista[start:end:step]

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

# el primer elemento
milista[0]  # [1]

# el ultimo elemento. en negativo seleccionas hacia atras
milista[-1] # [10]

# desde un indice a otro indice intervalo cerrado abierto
milista[2:4] # [3,4]

# el tercer parametro es el paso
milista[::2] # [1, 3, 5, 7, 9]

# para invertirla
milista[::-1] # [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

# el tamanio
len(milista) # 10

#podemos preguntar si x pertenece a la lista
print (1 in milista) # True
```

Listas

Los string son listas y por tanto todo lo que se aplica a listas se aplica a string

```
cad="PERICO EL DE LOS PALOTES"

print(cad[0:6])
otra=cad[::-1].lower()
print(otra)
print(cad.split(" "))
```

```
PERICO
setolap sol ed le ocirep
['PERICO', 'EL', 'DE', 'LOS', 'PALOTES']
```

Tuplas

Las tuplas tienen las mismas propiedades que las listas excepto que son **inmutables**

Por tanto no podemos aplicar operaciones de inserción y borrado
Su uso se centra en el la entrada y retorno de parámetros

```
# para crear una tupla con un elemento hay que poner coma

entero=(1) # entero vale 1
tupla=(10,)

# se puede crear sin usar paréntesis
tupla2=1,2,3  # (1,2,3)

# podemos usar como en las listas el operador +
t=tupla2+(4,5,6) # (1,2,3,4,5,6)
```

Tuplas

Otra característica de tuplas y listas es el **desempaquetado**

```
tupla1=('aa','bb','cc')
tupla2=1,2,3,4,5,6

#podemos separarla en valores. deben de coincidir el n de valores
a,b,c=tupla1 #a='aa' b='bb' c='cc'

# podemos obviar ciertos valores
_,a,_=tupla1 # a='bb'

#podemos separar listas grandes en distinto metraje PYTHON3
a,*b,c=tupla2 # a=1 b=(2,3,4,5) c=6
```

Conjuntos

- Los set son listas de valores únicos.
  - El contenido de los set sólo pueden ser valores inmutables
    - string, int, float, y tupla
  - Se definen con { } o con la palabra reservada set()

Operaciones de creación e inserción:

```
# crear sets
con1 = {1, 1, 2, 2, 3, 4}
con2=set((1,2,2,1))
con_vacio=set()  #  no usar {} esto crea un dict

con3={(1,2),(2,3)}

# insertar y borrar
con1.add(5)
con1.add(6)

con1.remove(2)  # si no existe retorna error
con1.discard(2) # si existe lo borra
con1.clear()    # vacía el conjunto
```

Conjuntos

Operaciones sobre conjuntos que todos conocemos:

```
con1 = {1, 1, 2, 2, 3, 4}
con2=set((1,3,5,7,9))

a=con1 & con2  # interseción
a=con1.intersection(con2)

b=con1 | con2 # union
b=con1.union(con2)

c=con1 - con2 # diferencia
c=con1.difference(con2)

d=con1 ^ con2 # diferencia simetrica
d=con1.symmetric_difference(con2)

print(con1 >=con2) # con1 es superset de con2
print(con1.issuperset(con2))

print(con1 <=con2) # con1 es subset de con2
print(con1.issubset(con2))

if 2 in con1:  # 2 esta en el set
   print("si")
```

Diccionarios

- Un diccionario es una array asociativo
- Asocia un valor con una clave
  - la clave sólo puede ser un tipo inmutable
  - string, int, float, y tupla

```
#un diccionario se define con {}
midict = {"one": 1, "two": 2, "three": 3}
otro_dict={}


# podemos leer, insertar y modificar sus valores usando como índice su clave
midict["one"]=12  # modifica el valor asociado a "one"

a=midict["one"] # a=1  si la clave no existe genera un error

midict["four"]=11 # asigna el valor 11 a la clave four

# borrar una entrada
del(midict["four"])

# podemos chequear si una clave existe. El nombre del dict  como iterador de claves
"one" in midict
```

Diccionarios

- Otras funcionalidades de los diccionarios:

```
midict = {"uno": 1, "dos": 2, "tres": 3}
# get no genera error si la clave no existe

a=midict.get("cuatro") # a vale None

# para insertar datos en un diccionario
midict.setdefault("cuatro",4) # sólo inserta si la clave NO existe
midict.setdefault("cuatro",5)

# unir dos diccionarios
midict.update({"cinco":5,"cuatro":4})
print (midict)
# {'uno': 1, 'dos': 2, 'tres': 3, 'cuatro': 4, 'cinco': 5}
```

- También podemos extraer las claves y los valores separados:

```
print (list(midict.keys()))
print (list(midict.values()))

# ['uno', 'dos', 'tres', 'cuatro', 'cinco']

# [1, 2, 3, 4, 5]
```

Control de flujo e iteradores

- condiciones if
- bucles while
- iterables
- bucles for (iteradores)
- excepciones

Control de flujo e iteradores

```
if <condition> :
      <if stament>
elif <condition>:    # opcional
      <elfi stament>
else:                          # opcional
      <else stament>
```

```
lista=[1,2,3,4,5,6,7]
a=5
if a in lista:
   print("{} esta en la lista".format(a))
else:
   print("{} no esta en la lista".format(a))

var=24
if var >25:
   print("{} mayor que 25".format(var))
elif var<25:
   print("{} menor que 25".format(var))
else:
   print("{} es igual que 25".format(var))
```

Control de flujo e iteradores

while <condition>:
    <while stament>

y....

Esto no da para más. veréis pocos bucles while en python

```
var=0
while  var<30:
   print(var)
   var+=1
```

```
while True:
   pass
# tipico bucle infinito
# la palabra reservada pass permite completar
el bloque sin que de error pero no hace nada
```

Esto es porque nuestros datos son colecciones

Nos interesa recorrerlas ... ***iterar colecciones***

while no soporta esta característica de forma implícita

Control de flujo e iteradores

- Python tiene una abstracción llamada iterable
- Permite recorrer colecciones obteniendo cada uno de sus valores
- Es un objeto que puede ser tratado con una secuencia
- Se usa de forma implícita en el bucle for
- Podemos hacer que nuestros objetos sean iteradores

```
a=[1,2,3,4]
iterador=iter(a) # genera un objeto iterador sobre la lista
print(next(iterador))  # next retorna valores 1
print(next(iterador))  # 2
print(next(iterador))  # 3
print(next(iterador))  # 4
print(next(iterador))

# Traceback (most recent call last):
#   File "/root/PycharmProjects/paco/prueba.py", line 7, in
<module>
#     print(next(iterador))
# StopIteration

print (list(iterador))  #para convertir un iterador en lista usamos
list
```

Control de flujo e iteradores

for <var> in <colection>:
    do anything with <var>

De forma interna genera un iterador
Con next extrae cada elemento y lo asigna a <var>
Hasta que se genere la excepción stopiteration

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

for a in ["casa","perro", "gato"]:
   print(a)

for x in lista:
   print("valor {}".format(x))

for x in range(100): # range genera una lista de 0 a 99 para ser iterada
   print(x*x)       # más efectivo usar xrange(100) es un generator
```

Control de flujo e iteradores

- Los diccionarios contienen tres iteradores implícitos
  - <dict>.keys() o <dict>
  - <dict>.values()
  - <dict>.items()

```
midict = {1: "casa", 2: "perro", 3: "gato"}

for k in midict:  # recorre las claves
   print(k)

for v in midict.values():  # recorre los valores
   print(v)

for k, v in midict.items(): # recorre los pares clave, valor
   print("el {} es {}".format(k, v))
```

Control de flujo e iteradores

- Las excepciones son objetos que nos permiten controlar posibles errores en bloques de código.
- Podemos forzar excepciones:
  - raise <exception>("This is an error")

```
try:
   a = 6 / 0
   # bloque en el que queremos controlar errores

except ZeroDivisionError as e:  # al menos una
   print("errror {}".format(e))

except (TypeError, NameError):
   pass  # podemos tratar un conjunto de excepciones

else:  # es opcional. debe estar justo aquí.
   print("perfecto!")  # si no hay errores se ejecuta

finally:  # finally se ejecutará siempre. opcional
   print("finalizado bloque")
```

Funciones

- Las funciones son bloques de código:
  - Llamable
  - Admiten parámetros de entrada
  - retornan datos procesados
  - tienen su propio ámbito de variables

```
x=1

# definimos funcion1
def funcion1(a,b):
   print("valor a:{} valor b.{}".format(a,b))
   return a+b

funcion1(x,10)   #funcion llamada
d=funcion1(10,10)
k=funcion1(b=1,a=2)
print(d)
```

```
valor a:1 valor b.10
valor a:10 valor b.10
valor a:2 valor b.1
20
```

- Dos reglas a tener en cuenta:
  - Los parámetros son siempre referenciados
  - Recuerda, todo son objetos... hasta las funciones

Funciones

- En cuanto al ámbito de las variable:
  - Las variables que se definen en una función son locales
  - Para modificar una variable de ámbito superior usamos global

```
x=10

def funcion1(a,b):
   global x # está haciendo referencia a la
variable global
   x=a-b
   print("valor a:{} valor b.{}".format(a,b))
   return a+b

print(x)
funcion1(100,10)
print("la var x vale {}".format(x))
```

```
10
valor a:100 valor b.10
la var x vale 90
```

Funciones

- Podemos definir la función con parámetros por defecto
  - Asignando un valor al parámetro en la definición
  - Estos parámetros deben estar a la derecha
  - Cuando es llamada, si NO usamos esos parámetros se asignarán los valores por defecto

```
def mifuncion(a,b=1,c=10):
   return (a*b)/c

print(mifuncion(10,10,12))

a=mifuncion(4)
print(a)

b=mifuncion(1,12)
print(b)
```

```
8.333333333333334
0.4
1.2
```

Funciones

- Podemos definir funciones que tomen un número variable de argumentos posicionales.

```
def otra_funcion(*args):
    for x in args:
        print(x)
    return args

arg=(10,11,12)

otra_funcion(1,2,3)
print("------")
otra_funcion(*arg)
```

```
1
2
3
------
10
11
12
```

- También funciones que sus argumentos sean claves de un diccionario:

```
def otra_funcion(**kwargs):
    for k,x in kwargs.items():
        print("argumento {}: {}".format(k,x))
    return kwargs

otra_funcion(a=1, b=True, h=50, z="Hello, world!") # pasamos los parámetros identificados

kwargs={"a":1,"b":"pepe","k":0.2} #pasamos un diccionario cargado previamente
otra_funcion(**kwargs)
```

```
argumento a: 1
argumento b: True
argumento h: 50
argumento z: Hello, world!
argumento a: 1
argumento b: pepe
argumento k: 0.2
```

Funciones

- Las funciones pueden retornar cualquier valor o valores y de cualquier tipo. Esta característica unido con la facilidad del paso de parámetros hace que los programas se simplifiquen mucho

```
def swap_var(x,y):
    return y,x

def mirango(x=100,y=1):
    return list(range(0,x,y))

x,y=swap_var(10,20)
print(x,y)

primero,*resto,ultimo=mirango(40,3)
print(primero,resto,ultimo)
```

```
20 10
0 [3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36] 39
```

Funciones

- Para terminar con funciones, os propongo un ejemplo en el que se usa un diccionario de funciones para una pequeña calculadora

```
def suma(x,y):
   return x+y
def resta(x,y):
   return x-y
def multiplica(x,y):
   return x*y

oper={"+":suma,"-":resta,"*":multiplica}

def operar(op):
   global oper
   a,op,b=op.split()
   return oper[op](int(a),int(b))


operacion="10 - 5"
print(operar(operacion))
operacion="10 * 5"
print(operar(operacion))
```

Módulos -- Packages

- La potencia de python se debe en gran medida a los módulos y packages
- Un package es uno o más módulos (ficheros.py) que contienen:
  - variables
  - funciones
  - clases
- Incluimos nuevos módulos en nuestros proyectos usando **import**

```
import time
import os
```

- Hay un comando en el S.O. que permite descargar nuevos paquetes

```
pip3 search pil
```

Módulos -- Packages

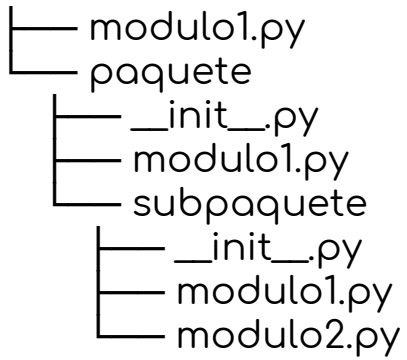
- Igual que en otros lenguajes, python usa espacios de nombres para referenciar objetos dentro de cada paquete
- Dependiendo cómo usemos la instrucción import, cambiará la forma de referenciar los contenidos.

```
import sound.effects.echo
sound.effects.echo.echofilter(.....)

#importar submodule
from sound.effects import echo
echo.echofilter(....)

from sound.effects.echo import echofilter
echofilter(....)

#podemos reducir el espacio de nombres
import sound.effects.echo as s
s.echofilter(.....)
```

```
sound/
    __init__.py
    formats/
        __init__.py
        wavread.py
        wavwrite.py
        aiffread.py
        aiffwrite.py
        auread.py
        auwrite.py
        ...
    effects/
        __init__.py
        echo.py
        surround.py
        reverse.py
        ...
    filters/
        __init__.py
        equalizer.py
        vocoder.py
        karaoke.py
        ...
```

Módulos -- Packages

- Una forma de estructurar nuestros programas es hacer nuestros propios paquetes de código.
- Un paquete es básicamente un directorio que contiene ficheros.py (modulos) y otros directorios
- Para que un paquete sea importable debe contener un fichero llamado __init__.py
- __init__ No es necesario que contenga nada, tan sólo que exista



Clases

- No vamos a hablar de programación orientada a objetos
- Veamos cómo se definen clases y cómo se usan:

```
class <nombre>(<herencia>):
    def metodo1(self,....):
        ......
    def metodo2(self,....):
        .....
```

- Aunque no es necesario en python 3.x es conveniente que las clases hereden de object
- Los atributos podemos definirlos en cualquier sitio dentro de la clase:
  - Al inicio sin self
  - Dentro de los métodos con self

```
class primera_clase(object):
   atrib1="un valor"
   atrib2="otro valor"
   def mas_valores(self):
       self.atrib3=10

a=primera_clase()
a.mas_valores()

print(a.atrib3)
print(a.atrib1)
```

Clases

- En realidad una clase puede ser tan simple como en nombre de la clase

```
class nada:
    pass
```

- Las clases pueden tener un método constructor que se ejecuta cuando la clase es instanciada. __init__
- Podemos crear destructores __del__ pero tienen menos uso puesto que se encarga el intérprete del borrado

```
class mi_clase(object):
    __privado="un valor"# un atributo privado debe empezar por __

    def __init__(self,a,b):
        self.op1=a
        self.op2=b

    def suma(self):
        return self.op1+self.op2

objeto=mi_clase(10,20)
print(objeto.suma())  # la salida es 30
```

Clases

- En python la herencia está soportada.
- No obstante cuando heredamos debemos llamar a al constructor de la clase heredada

```
class Clase_A(object):
   def __init__(self,v):
      print("Clase_A.init")
      self.v=v

   def mimetodo(self):
      print("el valor de v es {}".format(self.v))

class Clase_A1(Clase_A):
   def __init__(self,v):
      print("Clase_A1.init")
      super().__init__(v) # necesario para iniciar la clase a

   def metodo2(self):
      print("Clase_A1.metodo2()")

obj=Clase_A1(25)
obj.mimetodo()
```

```
Clase_A1.init
Clase_A.init
el valor de v es 25
```

Clases

- La herencia múltiple también está soportada. Aunque se puede complicar controlarla

```
class Clase_A1():
  def __init__(self):
    print("Clase_A1.init")

  def metodo2(self):
    print("Clase_A1.metodo2()")

class Clase_A2():
  def __init__(self):
    print("Clase_A2.init")

  def metodo2(self):
    print("Clase_A2.metodo2()")


class Clase_X(Clase_A1, Clase_A2):
  def __init__(self):
    print("Clase_X.init")

    Clase_A1.__init__(self)
    Clase_A2.__init__(self)

  def metodo1(self):
    print("Clase_X.metodo1()")


objeto1 = Clase_X()
```

Clases

- Hay una serie de métodos especiales en todas las clases denominados **métodos mágicos**
- Modifican la clase base de python.
- Permiten alterar el comportamiento de la clase
- Ya hemos usado uno (__init__)
- Estos métodos empiezan y terminan por doble guión bajo __<método>__

```
class forma(object):
   def __init__(self,ancho,alto):
      self.ancho=ancho
      self.alto=alto

   def __gt__(self,other):
      return (self.ancho*self.alto) >
(other.ancho*other.alto)

   def __str__(self):
      return "La forma tiene {} de ancho por {} de
largo".format(self.ancho,self.alto)


a=forma(10,20)
b=forma(5,5)
if a>b:
   print(a)
else:
   print(b)
```

Mañana...

Python con salsa.....

- Hablaremos de cosas más avanzadas
- Algo de programación funcional
- Funciones lambda
- Map, filter, zip,reduce
- List comprehensions
- Dict comprehensions
- Properties
- Iteradores
- Generadores
- Closures
- Decoradores
- Algo de metaprograming

Gracias