# CURSO AVANZADO PYTHON 3.5

python™

AVANZADO


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

# CONTENIDOS

- Algo de programación funcional
- Funciones lambda
- Any, All, Map, filter, zip,reduce
- List comprehensions
- Dict comprehensions
- Iteradores
- Generadores
- Clausuras y decoradores
- Decoradores built-in
- Algo de Metaprograming

# Objetivos de esta charla

- Hablaremos de ciertas características de python, algunas propias del lenguaje que harán nuestros programas más simples, claros y rápidos.

- Si no tienes conocimientos básicos de Python revisa la charla anterior y un montón de libros...

- Esta charla puede ser en ciertos momentos intensa.
- Si lo consideráis necesario pararemos para enfriar la cpu y tomarnos un café.

# PROGRAMACIÓN FUNCIONAL

# Algo de programación funcional

- Es un paradigma que enfatiza la utilización de funciones puras (matemáticas)
  - Funciones que no tengan efectos secundarios
  - Funciones que no manejen datos mutables o de estado
  - Lo contrario de la programación imperativa
- Ventajas:
  - Los programas son fáciles de testear
  - Mejoran la programación concurrente y paralela
  - Python optimiza el rendimiento en algoritmos que usan colecciones.
- Veremos sólo algunas características de programación funcional. Este tema da para muchas charlas.....

- Una función lambda o función anónima es una función que se puede definir en una línea
- variable = lambda parametro1, parametro2, parametro9=<defvalue>: instrucción retornada

```
cuadrado = lambda x:x*x

lista = [1,2,3,5,8,13]
for elemento in lista:
   print(cuadrado(elemento))

area_triangulo = lambda b,h: b*h/2

medidas = [(34, 8), (26, 8), (44, 18)]
for datos in medidas:
   base = datos[0]
   altura = datos[1]
   print(area_triangulo(base, altura))

fact = lambda x: 1 if x == 0 else x * fact(x-1)

print(fact(4))
```

Ventaja: podemos usarla directamente sin tener que asignarla o definirla previamente

# Programación funcional. *Funciones Lambda*

- ¿Os acordáis de la calculadora de la charla anterior?
- Ahora usando programación funcional. En este caso Lambdas
- En 10 transparencias más esto se puede simplificar
- Es normal no entenderlo a la primera. Tranquilidad hay más ejemplos

```
calcular={"+":lambda x,y:x+y,"-":lambda x,y:x-y,"/":lambda x,y:x/y,"*":lambda x,y:x*y,"E":lambda
x=0,y=0:"ERROR"}

operandos=lambda cad:cad.split(" ")  if len(cad.split(" "))==3 and cad.split(" ")[1] in ["+","-","*","/"] else
("0","E","0")

calculadora = lambda op1,op,op2:calcular[op](int(op1),int(op2))

print(calculadora(*operandos("20 + 10")))  #  retorna 30
print(calculadora(*operandos("20 * 10")))  #  retorna 200

print(eval("12-8"))  # con eval tenemos una calculadora en una línjuea  usando el propio intérprete.  salida 4
```

# Programación funcional. *any,all,map, filter, reduce, zip*

- Son funciones de orden superior.
- Las usamos cuando tenemos que hacer operaciones sobre listas, tuplas ...
- Iteran colecciones, **evitamos bucles**
- La mayoría son built-ins, palabras reservadas en Python.
- Están optimizadas para aumentar el rendimiento
- Pueden usar directamente funciones lambda
- La salida natural de estas funciones es un iterador
- Simplificamos y agilizamos nuestros códigos

# Programación funcional. *Any, All*

ANY

- **Any** retorna True si algún elemento del iterable es verdadero. Si el iterable es vacío retorna False.
- Any es una secuencia de OR consecutiva. Un valor True, para la iteración

```
print(any([0,1,2,3]))              # True
print(any([0, False, "", {}, []]))  # False
print(any([]))                     # False
a="hola mundo"
print(any(a))                      # True
```

ALL

- All retorna True si todos los valores de un iterable son verdadero. Si el iterable es vacío retorna True.
- All es una secuencia de AND consecutiva. Un valor False, para la iteración

```
l = [1, 3, 4, 5]
print(all(l)) # True
l = [0, False]
print(all(l))  # False
l = [1, 3, 4, 0]
print(all(l))  # False
l = [0, False, 5]
print(all(l))  # False
l = []
print(all(l))  #True
```

MAP

- Aplica una función a una lista de datos y devuelve un iterador con los resultados.

```
import math

def area_circulo(radio):
   return math.pi * radio ** 2

radios = [1, 2, 3,8,9]
# Devuelve iterador que es convertido a lista
areas = list(map(area_circulo, radios))
print(areas)

# [3.141592653589793, 12.566370614359172, 28.274333882308138,
201.06192982974676, 254.46900494077323]
```

- Podemos usar una función lambda y un bucle para iterar

```
import math

radios = list(range(200))

for areas in map(lambda r:math.pi*r**2, radios):
   print(areas)
```

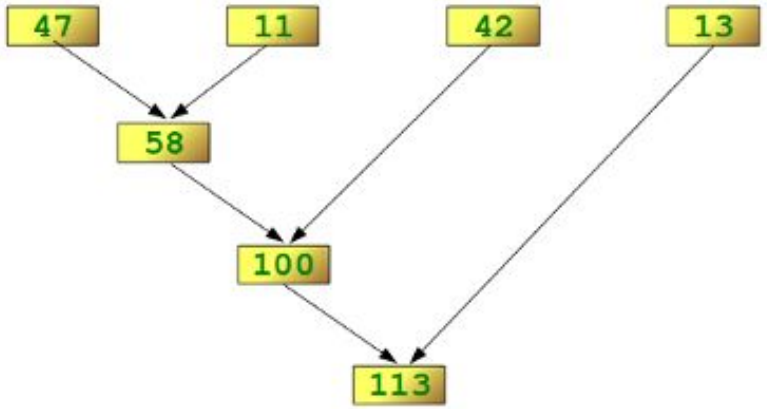
- Si la función tiene varias entradas, podemos aportar varias listas para los datos

```
import math

numeros = list(range(10))
potencias = list(range(10))

for val in map(lambda x,y:math.pow(x,y),numeros,potencias):
   print(val)

4.0
27.0
256.0
3125.0
46656.0
823543.0
16777216.0
387420489.0
```

REDUCE:

- Convierte una lista en un único valor aplicando una función reductora

```
import math
from functools import reduce
#en python3 reduce se encuentra en modulo functools

items= list(range(100))
suma = reduce(lambda x, y: x + y, items)
print(suma)

# la salida es 4950
```




- ¿Que hace esta reductora?

```
from functools import reduce

items= [100,20,104,30,203,200]

print(reduce(lambda a, b: a if a>b else b, items))
```

FILTER:

- Aplica una función booleana para poder extraer datos que cumplen la condición.
- La función lambda debe retornar una salida booleana

```
items=range(100)
for x in filter(lambda x: x % 2 == 0, items):
   print(x)
```

```
lista = ["lunes","martes","miércoles","jueves","viernes","sábado","domingo"]

for x in filter(lambda x: len(x)>6,lista):
   print(x)

miércoles
viernes
domingo
```

# Programación funcional. zip

- ZIP reorganiza listas
- Toma los elementos de todas las listas y genera una tupla iterable formada por los n-esimos elementos de cada una

```
def frec_palabras(cadena):
   frecuenciaPalab = []
   for p in cadena.split(" "):
      frecuenciaPalab.append(cadena.count(p))
   return dict(zip(cadena.split(" "),frecuenciaPalab))

frec=frec_palabras("python para todos esto es una prueba es una prueba")
print(frec)

{'python': 1, 'para': 1, 'todos': 1, 'esto': 1, 'es': 3, 'una': 2, 'prueba': 2}
```

```
def traspuesta(m):
   return list(zip(*m)) # el operador * desempaqueta una lista

matriz = [[1, 2, 3, 4],
       [5, 6, 7, 8],
       [9, 10, 11, 12]]

print(traspuesta(matriz))

[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

- Es una herramienta que permite definir listas expresando de forma funcional los elementos que debe contener.
- Aumentamos el rendimiento ya que el intérprete implementa estas instrucciones directamente en C.

```
c = []
for x in range(10):
   c.append(2 ** x)
print(c)

# en programacion funciona es:

d=[2**x for x in range(10)]
print(d)

[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

# Programación funcional. *list comprehensions*

- Es una forma elegante, limpia y efectiva para definir listas
- ¡ Nuestra mente no está preparada para esto !
- Somos imperativos. A veces se convierte en un vicio

```
lista=["paco","pedro","juan", "antonio","manolo"]

lista_po=[x for x in lista if x[-1]=="o" and x[0]=="p"]
print(lista_po)


# ['paco', 'pedro']
```

- Cuando queremos crear una lista que depende de dos variables, necesitamos más de un bucle o más de una condición

```
matrix = [[1, 2, 4],
          [0, 3, 1],
          [5, 2, 7]]

matrix_traspuesta = [[row[i] for row in matrix] for i in range(3)]

lista_mat = [x for a in matrix for x in a]

diagonal = [columna for f, fila in enumerate(matrix) for c, columna in enumerate(fila) if f == c]


print("matriz origen")
print(matrix)
print("matriz traspuesta")
print(matrix_traspuesta)
print("matriz en una lista")
print(lista_mat)
print("diagonal principal")
print(diagonal)
```

```
matriz origen
[[1, 2, 4], [0, 3, 1], [5, 2, 7]]
matriz traspuesta
[[1, 0, 5], [2, 3, 2], [4, 1, 7]]
matriz en una lista
[1, 2, 4, 0, 3, 1, 5, 2, 7]
diagonal principal
[1, 3, 7]
```

# Programación funcional. *Dict comprehensions*

- También podemos formar diccionarios con la misma técnica que hemos usado con las listas

```
personas = ["paco", "pepe","juan","manolo"]
edades = [10,30,25,56]


personas_ed ={k:v for k,v in zip(personas,edades)}

print(personas_ed)


{'paco': 10, 'pepe': 30, 'juan': 25, 'manolo': 56}
```

¿Qué contiene este diccionario?

```
diccionario={(k, v): k*v for k in range(10) for v in range(10)}
print(diccionario)
```

# Iteradores

- Ya sabemos que los iteradores nos permiten recorrer colecciones de datos. Implícito para los bucles for.
- Nuestras clases pueden tener la capacidad de iteración
- Para ello debemos sobrecargar los métodos __iter__ y __next__

```
class Potencia2:
    """esta clase itera potencias de dos hasta un max"""
    def __init__(self, max=0):
        """constructor de la clase"""
        self.max = max
    def __iter__(self):
        """inicializa el iterador"""
        self.n = 0
        return self
    def __next__(self):
        """retorna valores iterados hasta max """
        if self.n <= self.max:
            result = 2 ** self.n
            self.n += 1
            return result
        else:
            raise StopIteration  # esta excepción para la
iteración
```

```
pot=iter(Potencia2(100))

for x in range(3):
    print("potencia {}".format(next(pot)))

for i in Potencia2(4):
    print(i)
```

# Generadores

- Un generador es una  forma más simple de hacer iteradores
- Es una función que contiene la **palabra reservada yield** (return)
- La función retorna una nueva iteración por cada paso por yield
- En términos de memoria es muy eficiente
- La función retorna un objeto **generator** que es iterable

```
def fibonacci():
    """ generador fibbonacci infinito"""
    x, y = 0, 1
    while True:
        yield x
        x, y = y, x + y

def fibonacci_n(max):
    """ generador fibbonacci con limite"""
    x, y = 0, 1
    _max=0
    while _max<max:
        yield x
        _max+=1
        x, y = y, x + y

for i in fibonacci_n(100):
    print(i)
```

# Generadores

- Nos permite integrar perfectamente nuestros código con el propio lenguaje, ya que son iterables
- Supongamos un acceso a un sensor que nos devuelve temperaturas.
- No vamos a hacer un acceso intensivo
- La función se mantiene en memoria hasta que llegue la excepción stopiteration

```
import time
import random

def get_sensor(max=100):
   """ generador simulador de acceso a un sensor"""
   _max=0
   while _max<max:
      try:
         x=random.uniform(-10,40)
         time.sleep(0.001)
         yield round(x,2)
         _max+=1
      except:
         raise StopIteration("error de lectura")

lista=list(get_sensor(6)) #convertir el iterable en
lista
print (lista)
print("::::::::::")
it=iter(get_sensor(10))  #iterado con next()
print(next(it))
print(next(it))
print("__")
for i in get_sensor(5):  #iteración sobre el
generador
   print(i)
```

# Generadores

- Otra forma de definirlos es usando programación funcional
- En vez de usar [ ] usamos ( )

```
pares = (x for x in range(100) if x%2==0)

print(type(pares))
print(next(pares))
```

# Closures y Decoradores

- Una función puede tener funciones anidadas y por tanto el espacio de nombres de la función anidada es el local y el de su función padre.
- Si una función anidada quieres modificar una variable superior usa la palabra reservada nonlocal

```
def funcion():
    y=1
    def interna():
        var_int=25

print("var_int:{}".format(var_int)
)
    interna()
    print("var y:{}".format(y))

funcion()

#var_int: 25
#var y: 1
```

```
def funcion():
    y=1
    def interna():
        nonlocal y
        var_int=25
        y=y+var_int

print("var_int:{}".format(var_int)
)
    interna()
    print("var y:{}".format(y))

funcion()
#var_int: 25
#var y: 26
```

# Closures y Decoradores

- Un o una CLOSURE (clausura) es una función que retorna su función interna anidada.
- La característica principal :
  - Recuerda sus espacios de nombres local y superior
- Precursor de un constructor.  Base programación funcional

```
X: 10
contador: 11
X: 20
contador: 21
X: 20
contador: 22
X: 10
contador: 12
```

```
def contador(x):
    contador=x
    def interna():
        nonlocal contador
        contador+=1
        print("X: {}".format(x))
        print("contador: {}".format(contador))
    return interna

c1=contador(10)
c2=contador(20)

c1()
c2()
c2()
c1()
```

# Closures y Decoradores

- Caso práctico:
- Cálculo de iva. Por supuesto esto se puede hacer con otro parámetro.
- Esta forma es funcionalmente pura

```
def calcular_iva(base_iva):
    def estimar_neto(importe_bruto):
        return importe_bruto + (importe_bruto *   base_iva / 100)
    return estimar_neto

# Productos gravados con el 21%
get_neto_base_21 = calcular_iva(21)
ordenador = get_neto_base_21(500)
zapatos = get_neto_base_21(87)
cine = get_neto_base_21(6)
# Productos gravados con el 10.5%
get_neto_base_105 = calcular_iva(10.5)
carne = get_neto_base_105(10)
huevos = get_neto_base_105(2)

print(huevos)
print(ordenador)

2.21
605.0
```

# Closures y decoradores

- Un decorador es un Closure que toma como parámetro o argumento una función y retorna una función de reemplazo

```
def decorador(fun):
    def interna():
        print("esto es antes de la funcion",fun)
        return fun
    return interna

def mi_funcion():
    return 1

decora =decorador(mi_funcion())
decora()

# esto es antes de la funcion 1
```

- En este caso la variable decora nos retorna la versión decorada de mi_funcion()

# Closures y Decoradores

- Python como otros lenguajes tiene azúcar sintáctico, en este caso a través del símbolo @ facilita la aplicación de decoradores a nuestros códigos

```
def decorador(fun):
    def interna():
        print("antes de la funcion")
        return fun
    return interna


@decorador
def mi_funcion():
    return 1

print(mi_funcion())
```

```
def decorador(funcion):
    def funcion_envoltura():
        print("antes")
        funcion()
        print("despues")
    return funcion_envoltura

@decorador
def mi_funcion():
    print("funcion")
    return 1

print(mi_funcion())
```

- Por supuesto se pueden poner varios decoradores a una misma función pero no es aconsejable puesto que se hace difícil la traza de errores

# Closures y Decoradores

- Un caso práctico de ejemplo
- Un decorador para todas nuestra funciones que nos permita hacer logger de nuestros parámetro en cada llamada

```
def log(func):
    def interna(*args, **kwargs):
        print("{} :LLamada con argumentos : {},{}".format(func.__name__,args, kwargs))
        return func(*args, **kwargs)
    return interna

@log
def fun_producto(x,y=10,z=1):
    print ("Producto:{}".format(x*y*z))
    return x*y*z

@log
def procesar_l(f,lista):
    salida=[]
    for i in lista:
        salida.append(f(i)) #esta funcion es un map
    return salida

fun_producto(1,2,3)
l=procesar_l(lambda x: bin(x),[1,2,3,4,5,6])
print(l)
```

```
fun_producto :LLamada con argumentos : (1, 2, 3),{}
Producto:6
procesar_l :LLamada con argumentos : (<function <lambda> at
0x7faccad64620>, [1, 2, 3, 4, 5, 6]),{}
['0b1', '0b10', '0b11', '0b100', '0b101', '0b110']
```

- Podemos hacer clases decoradoras
- Podemos decorar clases
- Hay decoradores integrados (built-in) en python
  - Properties
  - classmethods
  - staticmethods

- La ventaja de hacer clases decoradoras es que el código está más organizado y podemos incluir más funcionalidad
- debemos cargar los métodos __init__ y __call__

```
class log(object):
   def __init__(self,func):
      self.func=func
   def __call__(self,*args,**kwargs):
      print("{} :LLamada con argumentos :
{},{}".format(self.func.__name__,args, kwargs))
      return self.func(*args, **kwargs)

@log
def fun_producto(x,y=10,z=1):
   print ("Producto:{}".format(x*y*z))
   return x*y*z

@log
def procesar_l(f,lista):
   salida=[]
   for i in lista:
      salida.append(f(i))
   return salida

fun_producto(1,2,3)
l=procesar_l(lambda x: bin(x),[1,2,3,4,5,6])
print(l)
```

- cuando un decorador se aplica a una clase, este obtiene por su método __call__ la clase a decorar

```
class Decorator(object):
    def __init__(self, arg):
        """ inyecta arg como atributo y el metodo: mimetodo"""
        self.arg = arg
        print(self.arg)
    def mimetodo(self,a):
        print("este es mi metodo")
        return a*1.21
    def __call__(self, cls):
        #print(cls)
        #print(cls.__dict__)
        print("injecting attribute and method")
        setattr(cls,self.mimetodo.__name__,self.mimetodo)
        setattr(cls,self.arg,21)
        return cls

@Decorator("iva")
class TestClass(object):
    def new_method(self, value):
        return value * 3

a=TestClass()
print(a.new_method(8))
print(a.iva)
print(a.mimetodo(21))
```

```
iva
injecting attribute and method
24
21
este es mi metodo
25.41
```

# Closures y decoradores. *Built-in Property*

- Es una forma de implementar métodos getter y setter para atributos de una clase
- Nos permite encapsular la asignación del atributo y sobrecargarlo

```
class persona(object):
    """Vamos a hacer una clase persona """
    def __init__(self,nombre,edad,peso):
        """este es el construtor de persona"""
        self._nombre = nombre
        self._edad = edad
        self._peso = peso
        self.pet_peso=0

    @property
    def nombre(self):
        """este es el getter de nombre"""
        return self._nombre

    @nombre.setter
    def nombre(self,nom):
        """este es el setter de nombre"""
        if isinstance(nom,str):
            self._nombre=nom
    @property
    def peso(self):
        """getter de peso"""
        self.pet_peso+=1
        return self._peso
    @peso.setter
    def peso(self,pes):
        """setter de peso"""
        self._peso=pes


per1 = persona("paco",25,50)
per1.nombre="paco andres hernandez"
per1.peso = 45
print(per1.nombre,per1.peso,per1.pet_peso)


paco andres hernandez 45 :1
```

# Closures y decoradores. *Built-in staticmethod*

- Es un método asociado con la clase y NO con las instancias
- No tiene self, sólo argumentos. No puede modificar ni la clase ni las instancias
- Solo sirve para definir utilidades. Mejor que tenerlas fuera de la clase. Accesible desde la clase y la instancia

```
class Date(object):

    def __init__(self, day=0, month=0, year=0):
        self.day = day
        self.month = month
        self.year = year

    def __str__(self):
        return str(self.day)+"-"+str(self.month)+"-"+str(self.year)

    @classmethod
    def from_string(cls, date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        date1 = cls(day, month, year)
        return date1

    @staticmethod
    def is_date_valid(date_as_string):
        day, month, year = map(int, date_as_string.split('-'))
        return day <= 31 and month <= 12 and year <= 3999
```

```
date1 = Date.from_string('20-02-2018')
date2 = Date(21,2,2018)
is_date = Date.is_date_valid('20-02-2018')
print(date1)
print(date2)
print(is_date)
```

```
20-2-2018
21-2-2018
True
```

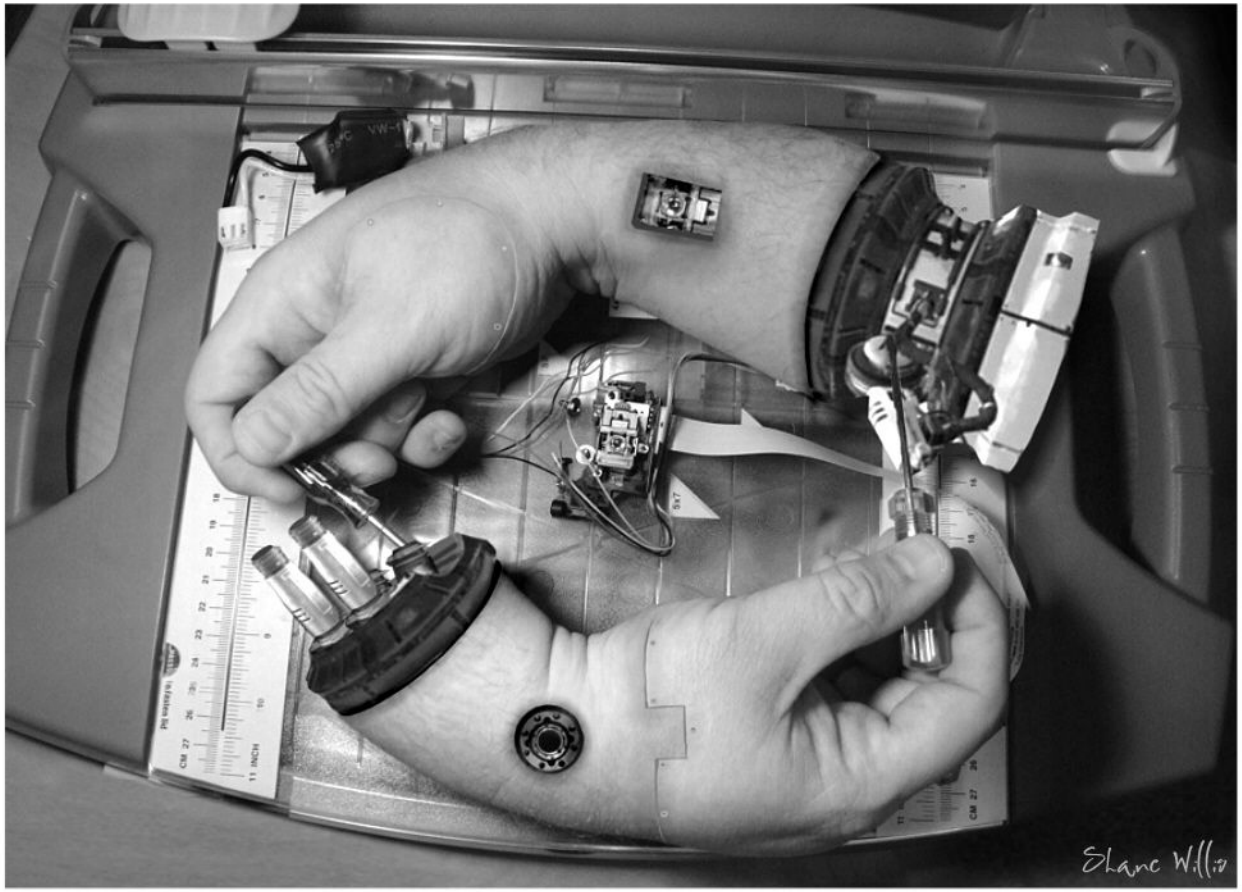
- Un método de clase puede modificar el estado de una clase, accediendo a los atributos de dicha clase, aún cuando el método es invocado desde un objeto.
- En lugar de definirse utilizando *self* como primer parámetro, se utiliza *cls*.
- Se usa para crear factory methods. Ejemplo distintos constructores

```
class PoblacionCensada():
    '''Clase que registra la cantidad de habitantes de todas sus instancias.'''
    poblacion = 0  # es un atributo de clase y de instancia
    '''Crea censos de población. '''

    @classmethod
    def opera_poblacion(cls, operador, cantidad):
        '''registra el número total de población de todas las instancias de la clase.'''
        cls.poblacion = eval(str(cls.poblacion) + operador + str(cantidad))

    @classmethod
    def despliega_total(cls):
        '''retorna el atributo de clase cls. población.'''
        return cls.poblacion

    def __init__(self, nombre, numero=0):
        print("Se ha creado la población {} con {} habitantes.".format(nombre, numero))
        self.nombre = nombre
        self.poblacion = numero
        self.opera_poblacion('+', self.poblacion)

    def __del__(self):
        self.opera_poblacion('-', self.poblacion)
```

```
extremadura = [PoblacionCensada("plasencia", 20000),
PoblacionCensada("badajoz", 120000),
PoblacionCensada("caceres", 80000),
PoblacionCensada('merida', 50000)]

print(extremadura[0].despliega_total())

del extremadura[3]

print(PoblacionCensada.despliega_total())
```

```
Se ha creado la población plasencia con 20000 habitantes.
Se ha creado la población badajoz con 120000 habitantes.
Se ha creado la población caceres con 80000 habitantes.
Se ha creado la población merida con 50000 habitantes.
270000
220000
```

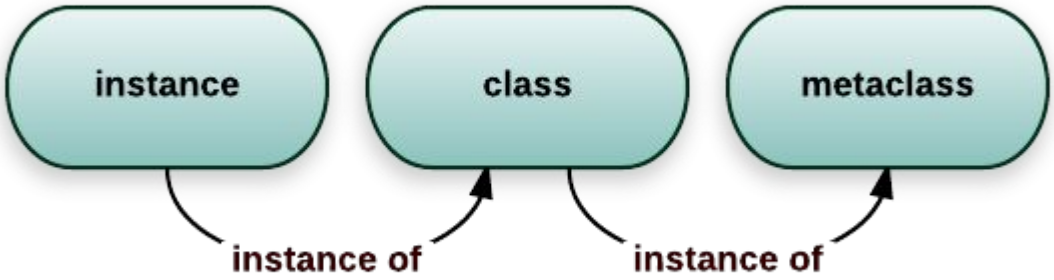
# METAPROGRAMACIÓN



# Algo de metaprograming

- Metaprogramar consiste en modificar el comportamiento o construir CLASES antes de instanciarlas.
- En este apartado veremos cómo se construyen las clases desde el más bajo nivel de Python.
- Este aspecto es bastante complejo por lo que sólo veremos ciertos conceptos de iniciación.
- Con los decoradores podemos conseguir este objetivo como hemos podido comprobar.

- Si los **objetos** son instancias de las **clases,** las clases son instancias de las **metaclases**.
- En Python todo está creado por clases. Los tipos básicos también.
- ¿De qué tipo son las clases?
- Una clase es un OBJETO y deriva de una metaclase llamada **TYPE**




```
class A(object):
   pass
numero=1
cadena="paco"
print(type(numero))
print(type(cadena))
print(type(int))
print(type(str))
print(type(A))

<class 'int'>
<class 'str'>
<class 'type'>
<class 'type'>
<class 'type'>
```

# Algo de metaprograming

- UNA CLASE ES UN OBJETO
- Podemos asignar sus atributos como en cualquier objeto.
- Como ejemplo es interesante

```
class BASE:pass

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

BASE.atributo="esto es una prueba"
BASE.funcion= lambda self:print("hola mundo")
setattr(BASE,"__init__",__init__)
setattr(BASE,"__doc__","documentacion de la
clase")

obj=BASE(10,"ufffs")
print(obj.atributo)
obj.funcion()
print(obj.a,obj.b)
print(BASE.__dict__)
```

```
esto es una prueba
hola mundo
10 ufffs
{'__module__': '__main__', '__dict__': <attribute '__dict__' of 'BASE' objects>,
'__weakref__': <attribute '__weakref__' of 'BASE' objects>, '__doc__': 'documentacion de
la clase', 'atributo': 'esto es una prueba', 'funcion': <function <lambda> at
0x7fbe83203158>, '__init__': <function __init__ at 0x7fbe832bae18>}
```

# Algo de metaprograming

- Podemos crear clases usando
  - type(<name>, (<herencia>,) ,dict(<atributo>:<value>, <method>:<value>)

```
class BASE(object):
   def base_metodo(self):
       print("base_metodo")

def nuevo_metodo(self):
   print("soy nuevo metodo")

A=type("A",(BASE,), dict(y="paco",inyectado=nuevo_metodo))

obj=A()
print(obj.y)
obj.base_metodo()
obj.inyectado()
print(A.__dict__)

paco
base_metodo
soy nuevo metodo
{'y': 'paco', 'inyectado': <function nuevo_metodo at 0x7f3991878e18>, '__module__': '__main__',
'__doc__': None}
```

# Algo de metaprograming

- Otra forma de modificar comportamientos de clases es sobreescribir su método __new__
- __new__ se ejecuta antes de instanciar el objeto con __init__ y toma como argumentos la clase.
- Con metaclass podemos derivar la metaclase tipo para la construcción de las clases.

```
class control(object):
    def control_mas(self,a,b):
        return a+b
    def control_menos(self,a,b):
        return a-b

class sensores(type):
    def __new__(cls, clsname, bases, clsdict):
        bases=bases+(control,) # TODO SENSOR DEBE HEREDAR DE CONTROL

        return super().__new__(cls, clsname, bases, clsdict)

class Base(metaclass=sensores):
    pass

class A(Base):
    pass

class B(Base):
    pass
print(B.__mro__) #method resolution order. orden de herencia
obj=A()
print(obj.control_mas(1,3))
print(obj.control_menos(10,8))

(<class '__main__.B'>, <class '__main__.Base'>, <class '__main__.control'>, <class 'object'>)
4
2
```

# Algo de metaprograming

- Python tiene un paquete de funciones que actúan sobre otras funciones (functools)
- En functools puedes encontrar funciones de orden superior y un conjunto de funciones que modifican comportamientos
- Si un objeto es callable ( implementa __call__ ) también podemos aplicar functools.
- Entre ellas podemos encontrar:
  - Partial:  permite fijar determinados parámetros a las funciones
  - lru_cache: decorador que almacena una memoria cache de resultados de la función decorada
  - ....
- Mejor echarle un vistazo

Contar con mi ayuda en la medida de mis posibilidades

# Gracias