

Funciones

- En todos los lenguajes estructurados se definen funciones, en **Python** también.
- Para crear una función, en cualquier parte del código utilizaremos la palabra clave **def** seguida del nombre de la función y entre paréntesis fijaremos los parámetros necesarios (entre cero y los que necesitemos).


```
def nombre_funcion(parametro1, parametro2):
```

```
    print "El primer valor es ", parametro1
```

```
    print "El segundo valor es ", parametro2
```

- Una vez definida haremos uso de ella simplemente utilizando su nombre y entre paréntesis los valores a usar.

```
    nombre_funcion("Lunes ", "Martes")
```

- 
- La **ventaja** que presenta Python con respecto a otros lenguajes es la posibilidad de ***pasar en los parámetros cualquier tipo de datos válido*** siendo la función la encargada de realizar las conversiones necesarias.
 - En una función necesitamos devolver un valor a través de la sentencia **return**, si no lo hacemos el intérprete devolverá automáticamente el valor **None**.
 - En ocasiones puede ser útil llamar a una función con menos parámetros que los que se han definido, inicializando los parámetros que falten a valores por defecto.
 - Para realizar esta tarea podemos añadir una asignación a continuación de la definición del parámetro indicando al sistema que si dicho valor no está establecido se use el que nosotros estamos definiendo.

- Los **valores por defecto para los parámetros** se definen situando un signo igual después del nombre del parámetro y el valor por defecto. En el ejemplo, si no indicamos un valor para el segundo parámetro, se utilizará 0.21.

```
def calcular_impuesto(cantidad, iva = 0.21):  
    print iva * cantidad
```

- También **es posible modificar el orden de los parámetros si indicamos el nombre del parámetro** al que asociar el valor a la hora de llamar a la función. Utilizando la función `calcular_impuesto` del ejemplo anterior:

```
calcular_impuesto(iva = 0.18, cantidad = 122.99)
```

- Por último, tenemos la posibilidad de recoger un número variable de parámetros con solo añadir al último el operador asterisco (*).

- En ese caso, el sistema pasará una tupla con los parámetros restantes una vez asignados valores a los primeros.

```
def mostrar_numeros(parametro1, parametro2, *mas_parametros):
```

```
    for valor in mas_parametros:
```

```
        print valor
```

```
mostrar_numeros(5, 7)
```

```
mostrar_numeros(5, 7, 9)
```

```
mostrar_numeros(5, 7, 9, 11)
```


- En el caso de preceder al último parámetro de **, se crea un **diccionario**.
- Las claves de este diccionario serían los nombres de los parámetros indicados al llamar a la función y los valores del diccionario, los valores asociados a estos parámetros.

- En el caso de los diccionarios, podemos utilizar la función **items**, que devuelve una lista con sus elementos, para imprimir los parámetros que contiene el diccionario. Por ejemplo:

```
def mostrar_valores(parametro1, parametro2, **mas_parametros):  
    for i in mas_parametros.items():  
        print i  
  
varios(120, 240, t = 350)
```

Paso de parámetros por valor o por referencia

- En el paso por **referencia** lo que se pasa como argumento es una **referencia** o puntero a la variable, es decir, la dirección de memoria en la que se encuentra el contenido de la variable, y no el contenido de la misma.

- 
- En el paso por **valor**, por el contrario, lo que se pasa como argumento es el valor que contiene la variable.
 - La diferencia entre ambos está en que en el paso por **valor** los cambios que se hagan sobre el parámetro no se ven fuera de la función, dado que los argumentos de la función son variables **locales** a la función. Es decir, en realidad lo que se le pasa a la función son copias de los valores.
 - Para modificar el valor de uno de los argumentos y que estos **cambios se reflejaran fuera de la función** tendríamos que pasar el parámetro por **referencia**.
 - En **Python**, los valores mutables se comportan como paso por **referencia**, y los inmutables como paso por **valor**.

Actividad 5.18

Crea un proyecto llamado **Funciones** y un módulo **Actividad 5-18** con el código siguiente. Ejecútalo.

```
# -*- coding: utf-8 -*-
# definición de la función
print "    definición de funciones    "
def miFun(param):
    if param > 2: return "Mayor de dos"
    else: return "Menor o igual que dos"
print miFun(2)
print miFun(3)
print miFun("abcd")

# parámetros optativos
print "    parámetros optativos o por defecto    "
def miFun2(par1, par2="Hola", par3=(True, 2)):
    print "Parámetros:", par1, par2, par3
miFun2("Adios")
miFun2(23, "Adiós")
miFun2(4, par3="Adios")
miFun2(2 + 5j, par3=23, par2="Par1")

# parámetros variables (el parámetro es una tupla si *, diccionario si **)
print "    parámetros variables    "
def miFun3(*params):
    for ele in params: print ele
miFun3("a")
miFun3(1, 2, "abc")
```

Console

```
<terminated> C:\Users\Web\workspace\Funciones\
    definición de funciones
Menor o igual que dos
Mayor de dos
Mayor de dos
    parámetros optativos o por defecto
Parámetros: Adios Hola (True, 2)
Parámetros: 23 Adiós (True, 2)
Parámetros: 4 Hola Adios
Parámetros: (2+5j) Par1 23
    parámetros variables
a
1
2
abc
```

Actividad 5.19

En el proyecto anterior crea un módulo **Actividad 5-19** con el código siguiente. Ejecútalo.

```
# -*- coding: utf-8 -*-
#Variables globales y locales, sólo las listas y diccionarios mantienen cambio
print "variables globales y locales"
varExtFija = 9;
varExtMod = [0, 0]
def miFun4():
    varExtFija = 1;
    varExtMod[0] = 1;
    print "Dentro"
    print "El valor de la externa fija", varExtFija
    print "El valor de la externa variable", varExtMod

print "Antes"
print "El valor de la externa fija", varExtFija
print "El valor de la externa variable", varExtMod
miFun4()
print "Después"
print "El valor de la externa fija", varExtFija
print "El valor de la externa variable", varExtMod

varExtFija1 = 9;
varExtMod1 = [0, 0]
def miFun5(p1, p2):
    p1 = 1; # Variable interna no usa el parámetro
    p2[0]=1;
    print "Dentro"
    print "El valor de la externa fija", p1
    print "El valor de la externa variable", p2


print "Antes"
print "El valor de la externa fija", varExtFija1
print "El valor de la externa variable", varExtMod1
miFun5(varExtFija1, varExtMod1)
print "Después"
print "El valor de la externa fija", varExtFija1
print "El valor de la externa variable", varExtMod1
```


Console

<terminated> C:\Users\Web\workspace\Funciones\src\A

variables globales y locales

Antes
El valor de la externa fija 9
El valor de la externa variable [0, 0]
Dentro
El valor de la externa fija 1
El valor de la externa variable [1, 0]
Después
El valor de la externa fija 9
El valor de la externa variable [1, 0]
Antes
El valor de la externa fija 9
El valor de la externa variable [0, 0]
Dentro
El valor de la externa fija 1
El valor de la externa variable [1, 0]
Después
El valor de la externa fija 9
El valor de la externa variable [1, 0]

- 
- Una vez vistas las funciones es momento de aprender **tres funciones útiles para la gestión de las listas**. Las introducimos ahora porque se necesitaba haber visto las funciones para entenderlas.
 - Se pueden usar tres **sentencias para crear una lista y modificarla** a la vez:
 - **map(función,lista)**: aplica una función definida por nosotros a cada elemento creando una lista resultante
 - **filter(función,lista)**: creará una nueva lista únicamente con aquellos elementos que tras pasarlos a la función definida, esta haya devuelto **True**
 - **reduce(función,lista)**: opera dos a dos sobre los componentes de la lista hasta que obtiene un único resultado final reutilizando los resultados intermedios.

- 
- Ya hemos mencionado que **para Python todo es un objeto**.
 - Según esta característica cualquier tipo básico es un objeto: números, cadenas, etc.
 - Teniendo en cuenta a las funciones o métodos implementados por un objeto (un módulo también es un objeto) se puede conseguir en cualquier momento una referencia a una función usando **getattr(objeto,nombremétodo)**, siendo el valor obtenido un *puntero a dicho método* en ese objeto que podremos llamar simplemente añadiendo paréntesis y los parámetros necesarios.

Actividad 5.20

En el proyecto anterior crea un módulo **Actividad 5-20**. Ejecútalo.

```
# -*- coding: utf-8 -*-
print "    Funciones a listas    "
# map aplica una función a cada elemento
INum1 = [1, 2, 3]
def cuadrado(n):
    return n * n
print INum1, "map:", map(cuadrado, INum1)

# filter verifica si cumplen condición
INum1 = [1, 2, 3, 4, 5, 6, 7, 8]
def par(n):
    return (n % 2 == 0)
print INum1, "filter:", filter(par, INum1)

# reduce opera dos a dos de la lista
INum1 = [1, 2, 3]
def suma(n, m):
    return n + m
print INum1, "reduce:", reduce(suma, INum1)
```

Console

```
<terminated> C:\Users\Web\workspace\Funciones\src\Actividad5-20\
Funciones a listas
[1, 2, 3] map: [1, 4, 9]
[1, 2, 3, 4, 5, 6, 7, 8] filter: [2, 4, 6, 8]
[1, 2, 3] reduce: 6
```

Actividad 5.21

En el proyecto anterior crea un módulo **Actividad 5-21** con el código siguiente. Ejecútalo.

```
# -*- coding: utf-8 -*-
print "    Funciones básicas    "
def miFun5(p1, p2):
    p1 = 1; # Variable interna no usa el parámetro
    p2[0]=1;
    print "Dentro"
    print "El valor de la externa fija", p1
    print "El valor de la externa variable", p2
print type(miFun5) #devuelve el valor de lo pasado
print str(5) #convierte a cadena
print dir(miFun5) #devuelve el contenido de un objeto
print callable(miFun5) #determina si el objeto se puede ejecutar

print "    Punteros a funciones    "
INum1 = [1, 2, 3]
print INum1
INum1pop = getattr(INum1, "pop") # referencia al método pop de INum1, puntero a esa función
print callable(INum1pop)
INum1pop() # llamamos a la función
print INum1
```

Console

```
<terminated> C:\Users\Web\workspace\Funciones\src\Actividad5-21\Actividad5-21.py
    Funciones básicas
<type 'function'>
5
['_call__', '__class__', '__closure__', '__code__', '__defaults__', '__delattr__', '__dict__',
True
    Punteros a funciones
[1, 2, 3]
True
[1, 2]
```

Operadores lógicos y función lambda

- Los operadores lógicos funcionan en cortocircuito, es decir, cuando se ha determinado completamente el sentido de una expresión condicional no se sigue evaluando la expresión.
- Así, en una expresión del tipo **Verdadero or CualquierFunción()**, la función **CualquierFunción()** no será llamada nunca.
- También hay que comentar que la representación del valor falso en Python no es un único valor, sino un conjunto de valores que se pueden interpretar así.



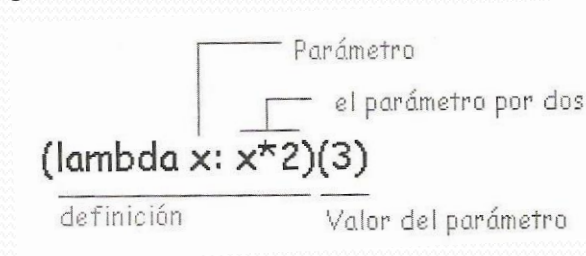
➤ Los **valores que se pueden interpretar como falsos** son:

- " " la cadena vacía,
- () la tupla vacía,
- { } el diccionario vacío,
- [] la lista vacía,
- 0 (el cero)
- y None,

el resto de valores se interpretará como verdadero no estando determinado qué valor se va a devolver en cada caso.

- Con los conocimientos sobre los operadores lógicos podemos implementar de otra manera más eficiente el operador condicional (?:) presente en otros lenguajes y no aquí: **expresión_booleana and valorSiVerdadero or ValorSiFalso**. Esta construcción fallará únicamente en el caso que **valorSiVerdadero** devuelve algún valor interpretable como falso, en tal caso los valores se pueden convertir en listas y evitaremos el error.
- Aunque no está permitida la asignación en línea, aparece una construcción tomada de los lenguajes funcionales: las **funciones lambda**. Esta construcción **permite crear una función simple en el lugar en el que necesitamos sin tener que definirla previamente**.

Ejemplo:



```
>>> myFunc = lambda x: x * 5
>>> print(myFunc(4))
20
```

Actividad 5.22

Crea un proyecto llamado **OperadoresLogyFuncLambda** y un módulo

Actividad 5-22 con el código siguiente. Ejecútalo.

```
# -*- coding: utf-8 -*-
# uso de and y or
print "uso de and y or"
print "ABC" and 0
print "ABC" and {}
print "ABC" and []
print "ABC" and ""
print "ABC" and False
print "ABC" and None
print "ABC" or 0
print "ABC" or {}
print "ABC" or []
print "ABC" or ""
print "ABC" or False
print "ABC" or None

print "and y or cortocircuito"
def miFun(p):
    print p
False and miFun("Valor AND") # la función no se ejecuta al ser falsa la expresión en
False or miFun("Valor Or") # la función se ejecuta al ser verdadera la expresión en

print "operador ?:"
sOp1 = "a"
sOp2 = "b"
iVal = 5
print iVal == 5 and sOp1 or sOp2
print iVal != 5 and sOp1 or sOp2
print "funciones en línea (lambda)"
print (lambda x: x * 3)(6) #función de un parámetro x, que se multiplica por tres (x*3). En este caso 18 (6)
```

Console

<terminated> C:\Users\Web\workspace\Op

uso de and y or

0

{}

[]

False

None

ABC

ABC

ABC

ABC

ABC

ABC

and y or cortocircuito

Valor Or

operador ?:

a

b

funciones en línea (lambda)

18

Estructura modular

- Un **paquete** en **Python** es un directorio que contiene un fichero especial llamado `__init__.py` más los ficheros necesarios.
- El paquete se crea directamente desde **Eclipse**.
- Dentro de un paquete pueden aparecer a su vez más sub-paquetes (otros directorios).
- Un **módulo** es un fichero **Python** cualquiera que es usado desde otro fichero.
- Para poder usar las variables y funciones definidas en un fichero fuente, se tiene que importar antes mediante el uso de la sentencia **import**.

- La **importación** se puede hacer de forma completa con la fórmula

import nombreModulo

o de forma parcial solo los elementos que necesitemos mediante

from nombreModulo import nombre[,nombre]

- Aunque parecen iguales las dos difieren en el modo de uso.
- La primera forma impone el uso del nombre del paquete cuando llamemos a los componentes separándolos con un punto, mediante la segunda se pueden usar los objetos importados de forma directa, sin cualificar con el nombre del paquete. En caso de que existan subpaquetes, extenderemos la notación del punto.

Actividad 5.23

Crea un **proyecto** de nombre **Actividad23** y un **paquete** del mismo nombre.

El paquete tendrá dos ficheros.

El código del fichero principal es el siguiente:

```
# -*- coding: utf-8 -*-
import Actividad23inc
from time import localtime

print "La hora local es %2d:%.2d" % (localtime().tm_hour, localtime().tm_min) #no es necesario el paquete
c = 'N'
while c != 'S':
    Actividad23inc.printMenu() #obligatorio el nombre del paquete
    c = raw_input("¿Opción?").upper()
    if c != 'S':
        op1 = raw_input("¿Primer operando?")
        op2 = raw_input("¿Segundo operando?")
        print Actividad23inc.calcula(c, op1, op2)
```

Actividad 5.23 (...continuación)

El módulo que invoca (**Actividad23inc**) tiene el siguiente código.

```
# -*- coding: utf-8 -*-
def calcula(op, val1, val2):
    """
    @param op: Operación a realizar +,-,*,/
    @param val1: Primer operando
    @param val2: Segundo operando
    @return: Resultado de la evaluación o false si hay error
    """

    try: #lo veremos en el apartado control de errores
        val1 = int(val1)
    except ValueError:
        return False
    try:
        val2 = int(val2)
    except ValueError:
        return False

    if op == '+':
        return val1 + val2
    elif op == '-':
        return val1 - val2
    elif op == '*':
        return val1 * val2
    elif op == '/':
        if val2 == 0: return False
        return val1 / val2
    else:
        return False

def printMenu():
    """
    Muestra el menú
    """
    print "+.-Suma"
    print "-.-Resta"
    print "*.-Multiplicación"
    print "/.-División"
    print "S.-Salir"
```


Programación orientada a objetos

- Python también es un lenguaje orientado a objetos, incorporando la mayoría de los paradigmas de dicha programación.
- Para **definir una clase** se utiliza la palabra clave **class** seguida del nombre de la clase, separada del resto de la definición mediante el uso de los dos puntos (:) y sangrando todo el contenido.

-*- coding: utf-8 -*-


class Mueble: # el primer parámetro siempre self en todos

```
    def __init__(self, tipo): # Constructor
        self.tipo = tipo #Propiedad pública
    def getTipo(self): # método getter
        return self.tipo
    def setTipo(self, tipo): # método setter
        self.tipo = tipo
```

- 
- Cada método perteneciente a la clase se creará como una función general, usando todas las características mencionadas hasta ahora.
 - El ejemplo muestra dos métodos, **getTipo()** y **setTipo()**, utilizados ambos para establecer el valor de la propiedad **tipo** definida en el **constructor** (**__init__(self.tipo)**).
 - A la hora de **definir métodos para una clase**, la única peculiaridad que tendremos que tener muy en cuenta es que el primer parámetro de **cada método obligatoriamente será un objeto introducido por el intérprete cuando le llame (self)**.
 - En caso del **constructor** dicho objeto es el nuevo objeto, en el resto es el mismo objeto.

- Este valor se introduce de forma automática no siendo necesario utilizarlo cuando el programador llame al método **(objetoMueble.getTipo())**.
- La **creación de propiedades** se debe hacer **en el constructor**, dando valor a todas las que necesitemos en el objeto self que nos pasan.
- El uso de las clases es similar al de cualquier lenguaje orientado a objetos, creando la variable al asignarle un valor, pasando entre paréntesis los datos al constructor exceptuando el parámetro self. Ejemplo:

```
mu = Mueble(3) # Llamamos al constructor pero no usamos self en ninguno al llamar
print "Mueble(3)"
print "El tipo es (método):", mu.getTipo( )
print "El tipo es (propiedad):", mu.tipo
```

- 
- La **herencia** se implementa **acompañando al nuevo nombre las clases de las que hereda separadas por comas y encerrándolas entre paréntesis.**
 - Como la **herencia** conlleva trasvase de métodos y propiedades, estas deberán ser inicializadas adecuadamente desde la ejecución del nuevo constructor (hijo) llamando al constructor (padre) de todas las clases de las que heredamos pasándoles los datos correctos, inicializando primero el contenido de los padres y, a continuación, los propios de la nueva clase.
 - En el siguiente **ejemplo**, declaramos una *clase Mesa que hereda de la clase Mueble y añade una propiedad nueva (color) y los métodos asociados a la misma.*

HERENCIA

```
class Mesa(Mueble):
```

```
    def __init__(self, tipo, color):
```

```
        Mueble.__init__(self, tipo) # Llamamos al padre para inicializar
```

```
        self.color = color #nuestra propiedad
```

```
    def getColor(self): #nuestros métodos
```

```
        return self.color
```

```
    def setColor(self, color):
```


```
        self.color = color
```

```
ms = Mesa(5, 7)
```

```
print "Mesa(5,7)"
```

```
print "El tipo es:", ms.getTipo() #método del padre
```

```
print "El color es:", ms.getColor() #método de la clase hija
```

- 
- **Python no permite la sobrecarga de métodos de un padre**, utilizando siempre la última definición que encuentre.
 - No podremos llamar a diferentes métodos en función del tipo de la variable ya que esto está prohibido.
 - Pero no es un problema ya que un método que recoge un valor, el tipo de dicho valor puede ser cualquiera, por lo que se puede determinar qué acción realizar en función del tipo con un único método.
 - En el caso de que sea imprescindible el uso de parámetros variables con un único método podemos usar el operador asterisco.

➤ Ejemplo:


#SOBRESCRITURA DE MÉTODOS

```
class Mesa2(Mueble):  
    def __init__(self, tipo, color):  
        Mueble.__init__(self, tipo)  
        self.color = color  
    def getColor(self):  
        return self.color  
    def setColor(self, color):  
        self.color = color  
    def getTipo(self): #sobrescribimos el padre  
        return self.tipo + 1
```

```
ms2 = Mesa2(5, 7)  
print "Mesa2(5,7)"  
print "El tipo es:", ms2.getTipo()  
print "El color es:", ms2.getColor()
```

Actividad 5.24

- Crea un **módulo** de nombre **Actividad24** que incluya la clase Mueble de los apuntes y las líneas con el código del ejemplo (en color verde).
- Añade al módulo Actividad24 el código de la clase Mesa y las líneas de código para probar su funcionamiento (en color azul).
- Añade al módulo Actividad24 el código de la clase Mesa2 y las líneas de código para probar su funcionamiento (en color rojo).

- 
- El último elemento clave de la programación orientada a objetos es la **encapsulación de datos y protección de acceso.**
 - Este punto bajo Python es muy simple, no existe ningún tipo de protección, **todo es público y accesible.**
 - En principio, Python considera propiedades y métodos **privados** a aquellos que **comienzan por dos guiones bajos (__)**, pero el mecanismo interior del intérprete es renombrar dicho componente anteponiendo el nombre de la clase seguido del objeto. De esta manera, la propiedad "supuestamente" privada **__Privada** del ejemplo daría una excepción al intentar acceder directamente, pero si utilizamos la notación **NombreClase__Privada** no hay ningún problema.

➤ Ejemplo:

NO EXISTE NADA PRIVADO

class UnaPrivada:

def __init__(self):

self.__Privada = 1 # Supuesta variable privada

self.Publica = 2

pr = UnaPrivada()

print "Pública:", pr.Publica

print "Privada:", pr.__Privada #Lanza una excepción

print "Privada:", pr._UnaPrivada__Privada # Se puede acceder, se ha renombrado

Actividad 5.25

Crea un proyecto con el código siguiente. Ejecútalo.

```
# -*- coding: utf-8 -*-
# NO EXISTE NADA PRIVADO
class UnaPrivada:
    def __init__(self):
        self.__Privada = 1 # Suguesta variable privada
        self.Publica = 2


pr = UnaPrivada()
print "Pública:", pr.Publica
# print "Privada:", pr.__Privada #lanza una excepción
print "Privada:", pr._UnaPrivada__Privada # Se puede acceder, se ha renombrado
```

Console

<terminated> C:\Users\Web\workspace\Clases\src\Actividad25__init__.py

Pública: 2

Privada: 1

- 
- La implementación de propiedades a través de métodos **setter** y **getter** también está presente.
 - El mecanismo es sencillo, en el constructor definimos una propiedad privada, creamos dos métodos, uno para establecer (*set*) y otro para recoger (*get*) en los que usamos la propiedad "privada" y a continuación definimos la propiedad pública con un nombre distinto mediante la sentencia **NombrePropiedad=property(método_get,método_set)**.
 - Se usa cuando queramos permitir el acceso a algún atributo de nuestro objeto, pero que este se produzca de forma controlada.

➤ Ejemplo:

#PROPIEDADES

class Propiedades:

```
def __init__(self, dia):
    self.__d = dia; #propiedad __d que almacenará el valor privada
def __getDia(self): #llamado al establecer dia, no se puede usar directamente
    return self.__d
def __setDia(self, dia): #llamado al recoger dia, no se puede usar directamente
    self.__d = dia
dia = property(__getDia,__setDia) #Propiedad dia creada
```

```
d = Propiedades(3)
print "Día antes:", d.dia
d.dia = 7
#print "Día después:",d.__getDia() ##Excepción
print "Día después:", d.dia #nueva propiedad
```


Actividad 5.26

Crea un proyecto con el código siguiente. Ejecútalo.

```
Actividad26 ✖
# -*- coding: utf-8 -*-
#PROPIEDADES
class Propiedades:
    def __init__(self, dia):
        self.__d = dia; #propiedad __d que almacenará el valor privada
    def __getDia(self): #llamado al establecer dia, no se puede usar directamente
        return self.__d
    def __setDia(self, dia): #llamado al recoger dia, no se puede usar directamente
        self.__d = dia
        dia = property(__getDia, __setDia) #Propiedad dia creada

d = Propiedades(3)
print "Día antes:", d.dia
d.dia = 7
#print "Día después:", d.__getDia() ##Excepción
print "Día después:", d.dia #nueva propiedad

Console ✖
<terminated> C:\Users\Web\workspace\Clases\src\Actividad26\__init__.py
Día antes: 3
Día después: 7
```

- 
- Las clases permiten un conjunto de métodos con nombres especiales que la dotarán de funcionalidad añadida.
 - Si queremos comparar dos clases mediante los operadores pertinentes implementaremos el método **__cmp__**, si necesitamos imprimir la clase con la función **print** se usará el método **__str__**, etc.
 - Los métodos más importantes a conocer son:
 - **__init__(self.args)**. Constructor.
 - **__del__(self)**. Destructor.
 - **__delitem__**, **__getitem__**, **__setitem__** se implementarán si queremos dotar a la clase de la misma funcionalidad que un diccionario.

- **__str__(self)**, se llama cuando se necesite una representación en cadena de la clase.
 - **__cmp__(self,otro)** se utiliza en las comparaciones, pudiendo usarse la clase en las sentencias **if** sin ningún problema. Este método devolverá menos uno si el primer elemento es menor, cero si son iguales y uno si el segundo elemento es menor.
 - **__len__(self)** se usa con la sentencia **len(ObjetoClase)** determinando la longitud de la clase.
- Para terminar el recorrido por las características más importantes de la POO en Python vamos a comentar dos operadores que podemos usar para determinar la igualdad de los objetos, los operadores **is** e **igualdad (==)**. El primero (**is**) determinará si dos variables referencian al mismo objeto en memoria. El segundo establecerá cuando dos objetos tienen el mismo valor (llamando al método **__cmp__** implementado en la clase).

Actividad 5.27

Crea un proyecto con el código siguiente. Ejecútalo. A la derecha puedes ver lo que tiene que salir.

```
#!/usr/bin/env python3
# coding: utf-8 -*-
# comparación de clases
class Mueble: # el primer parámetro siempre self en todas
    def __init__(self, tipo): # Constructor el destructor es __del__(self)
        self.tipo = tipo
    def getTipo(self): # método getter
        return self.tipo
    def setTipo(self, tipo): # método setter
        self.tipo = tipo

# ver http://docs.python.org/2/reference/datamodel.html para todas las opciones

# Personalización de las clases
    def __cmp__(self, other):
        if self.getTipo() < other.getTipo(): return -1
        elif self.getTipo() == other.getTipo(): return 0
        else: return 1
    def __str__(self):
        return "El objeto vale: " + str(self.tipo)
    def __len__(self):
        return self.tipo.__sizeof__()

muVal1 = Mueble(3)
muVal2 = Mueble(2)
muVal3 = Mueble(3)

if muVal1 == muVal2:
    print "Iguales"
else:
    print "distintos"

if muVal1 == muVal1:
    print "Iguales"
else:
    print "distintos"

if muVal1 == muVal3: # ¿contienen el mismo valor?
    print "Iguales"
else:
    print "distintos"

if muVal1 is muVal3: # ¿es el mismo objeto?
    print "Iguales"
else:
    print "distintos"

print muVal1
print len(muVal1)
```

```
Console
<terminated> C:\Users\Web\workspace\Clases\src\Actividad27\_init_.py
distintos
Iguales
Iguales
distintos
El objeto vale: 3
12
```