Funciones

Contenido:

1. Definición:

- Se declaran **en cualquier lugar** del código, es decir, no hay lugar específico para su declaración (principio o final del código, por ejemplo)
- Sentencia:

```
def nombreFuncion (parametro1, parametro2,…):

        …….

        …….

        return valor
```

- Si la sentencia **return no existe, devolverá none**.

Ejemplos funciones:

```
    def basica():
        print "Sin argumentos ni return"

    print "Llamada a basica"
    print "retorno:",basica()
    #print basica ("Hola")<-- Sentencia errónea. Esta función no tiene
argumentos

    def basica_return():
        print "Sin argumentos y return"
        return "Hola"

    print "Llamada a basica_return"
    print "retorno:",basica_return()

    def basica_argumentos(arg1,arg2):
            print "Con argumentos y return"
            arg1=arg1+arg2
            return arg1

    print "Llamada a basica_argumentos"
    print "retorno:",basica_argumentos(3,6)
```

```
 print "retorno:",basica_argumentos(3,6)
#print "retorno:",basica_argumentos(3)<-- Sentencia errónea. Esta función
tiene 2 argumentos
```

2. Funciones con argumentos inicializados

Posibilidad de llamar a una función con **menos parámetros de los que tiene definidos**, siempre y **cuando** éstos **en la definición** de la función **tengan un valor por defecto**.

def nombrefuncion (arg1, arg2,arg3=valor)

Estas funciones a la hora de ser llamadas, podrán sufrir reasignación de valores de sus argumentos.

Los **argumentos** que lleven valores por defecto **inicializados**, deberán ir en la declaración de la función **al final**.

Sentencia correcta:

```
def función (arg1, arg2,arg3=valor)
```

Sentencia errónea:

```
def función (arg1, arg2=valor, arg3)
```

Ejemplos:

```
def argumentos (arg1,arg2,arg3=23,arg4=(4,True)):
        print arg1,arg2,arg3,arg4



print "Llamada a argumentos"
#argumentos(11)<-- Sentencia errónea. Esta función tiene al menos 2
argumentos que deben ser asignados
argumentos(11,33) # Solo paso 2 argumentos porque los otros están
inicializados

print "Llamada a argumentos con reasignación en uno de los valores"
argumentos(11, 33, 44)
argumentos(11,33,arg4=55)
#argumentos (11,33,arg4=55,65)<-- Sentencia errónea.
argumentos(11,33,arg4=55,arg3=44)

#argumentos (arg3=44,11,arg4=55,33)<-- Sentencia errónea.
argumentos(11,33,arg4=(False,45),arg3=(22,1))
```

3. Funciones con parámetros en una tupla o en un diccionario

Puede haber funciones que reciban sus argumentos en forma de tupla o diccionario. Si es una **tupla en la declaración** llevará un asterisco delante para indicarlo,*****, y si es un **diccionario** llevará 2 asteriscos,******.

Ejemplo tupla:

```
def funciontupla(*tupla):
  for i in tupla: print i

#Main
print "1-Definicion de la tupla"
Tvariable=(34,"Hola",12.4)
print "2- Llamada a funciontupla"
funciontupla(Tvariable)
print "1- Llamada con la tupla en los argumentos"
funciontupla(11,11,111,"Adios","Quizás")
funciontupla("Solo1")
```

Ejemplo diccionario:

```
def funciondiccionario(**diccionario):
    for i in diccionario: print i
    #Otra opción de impresión, usando funciones de los diccionarios,
  por ejemplo: print diccionario.items()


#Main
 print "1- Llamada a funciondiccionario"
funciondiccionario(clave1="valor uno", clave2="valor dos")
```

Ejemplo función con parámetros de diferentes tipos:

```
def recorrer_parametros_arbitrarios(parametro_fijo, *tupla, **dic):

      print parametro_fijo
      for argumento in tupla:
            print argumento
       # Los argumentos arbitrarios tipo clave, se recorren como los
diccionarios
      for clave in dic:
            print "El valor de", clave, "es", dic[clave]

#Main
print "1- Llamada a con parámetros mezclados"
recorrer_parametros_arbitrarios("FIJO", "tupla 1", "tupla2", clave1="valor
uno", clave2="valor dos")
#recorrer_parametros_arbitrarios ("FIJO", clave1="valor uno", clave2="valor
dos","arbitrario 1", "arbitrario 2", "arbitrario 3")<-- Sentencia errónea
```

4. Variables locales y globales

La asignación de las variables y su ámbito funciona igual que en otros lenguajes. Lo único que hay que tener en cuenta es que las **listas** y los **diccionarios** si **mantienen** el **cambio** si han sido **reasignados dentro de la función.**

Ejemplos:

```
    def función ():
        var1="valor1local"
        var2[0]="Nuevo valor"
        print "Dentro función"
        print "var1",var1
        print "var2",var2

   #Main
    print "Variables locales y globales"
    var1="valor1global"
    var2=["valor1globalLista","valor2globalLista"]
    función()

    print "Fuera de la función"
    print "SOLO las listas y los diccionarios mantienen el cambio realizado"
    print "var1:",var1
    print "var2",var2
```

Resultado:

```
Variables locales y globales
Dentro función
var1 valor1local
var2 ['Nuevo valor', 'valor2globalLista']

Fuera de la función
SOLO las listas y los diccionarios mantienen el cambio realizado
var1: valor1global
var2 ['Nuevo valor', 'valor2globalLista']
```

```
    print "Segunda función"
    var1="valor1global"
    var2=["valor1globalLista","valor2globalLista"]

    def funcion2(var1,var2):
        var1="valor1local"
        var2[0]="Nuevo valor"
        print "Dentro función"
        print "var1",var1
        print "var2",var2

    funcion2(var1,var2)
    print "Fuera de la función"
    print "SOLO las listas y los diccionarios mantienen el cambio realizado
dentro de la función"
    print "var1:",var1
    print "var2",var2
```

Resultado:

```
Segunda función
Dentro función
var1 valor1local
var2 ['Nuevo valor', 'valor2globalLista']
Fuera de la función
SOLO las listas y los diccionarios mantienen el cambio realizado
dentro de la función
var1: valor1global
var2 ['Nuevo valor', 'valor12globalLista']
```

En el caso de querer utilizar una variable global dentro de una función (en el ejemplo var1), en la propia función habría que indicar que se va a a utilizar declarándola al inicio de la función y precedida de la variable global

Ejemplo:

```
    def función ():
        global var1 #Así indicamos que la variable var que utilizaremos de
aquí en adelante no es local, sino la global
        var1="valor1local"
```

5. Funciones a Listas

- map (función, lista)
  Aplica una función en concreto a todos los elementos de la lista
  Ejemplo:

```
print "map-- aplica una función a cada elemento de la lista"
def num_mas2(arg1):
      arg1=arg1+2
      return arg1

lista_num=[2,3,4,5,6]
print "Lista original",lista_num
print map(num_mas2,lista_num)
```

  Resultado:

```
map-- aplica una función a cada elemento de la lista
Lista original [2, 3, 4, 5, 6]
[4, 5, 6, 7, 8]
```

- filter (función, lista)
  Verifica si cumplen una condición los elementos de la lista. Dicha condición se especificará en el return de la función
  Ejemplo:

```
print "filter-- filtra resultado de la función"
def num_mas2(arg1):
    arg1=arg1*2
    return (arg1>=10)
lista_num=[2,3,4,5,6]
print "Lista original",lista_num
print "El doble del número es mayor de 9 en el caso de:",

filter(num_mas2,lista_num)
```

  Resultado:

```
filter-- filtra resultado de la función
Lista original [2, 3, 4, 5, 6]
El doble del número es mayor de 9 en el caso de: [5, 6]
```

- reduce (función, lista)
  Opera de dos a dos los elementos de la lista. El resultado de la operación anterior se pasa como parámetro en la siguiente.
  Ejemplos así en abstracto:
  - dada una lista de precios, obtener el promedio
  - data una lista de números obtener el máximo
  - dada una lista de personas obtener la mayor (de mayor edad)
  - dada una lista de números obtener la suma de ellos.

  Para esto aplica una función reductora de dos parámetros en forma progresiva de izquierda a derecha. La función deberá recibir dos parámetros y retornar un único valor. Ese valor se utilizará para invocar la función al siguiente elemento.

  Ejemplos:

```
    Lista original [2, 3, 4, 5, 6]
    print "reduce-- opera los elementos de la lista de 2 en 2. "
    def num_mas2(arg1,arg2):
        return (arg1*arg2)
    print reduce(num_mas2, lista_num)," "
```

Resultado:

```
      720
```

Este resultado es la multiplicación de todos los elementos de la lista: 2*3*4*5*6

Otro ejemplo con visualización paso a paso:

```
lista_num=[2,3,4,5,6]
def num_mas3(arg1,arg2):
  print "Dentro función:- arg1:",arg1,"-arg2:",arg2
  print "return:",arg1*arg2
  return (arg1*arg2)
resultado=reduce(num_mas3, lista_num)
print "Reduce total",resultado
```

Resultado:

```
      Dentro función:- arg1: 2 -arg2: 3
      return: 6
      Dentro función:- arg1: 6 -arg2: 4
      return: 24
      Dentro función:- arg1: 24 -arg2: 5
      return: 120
      Dentro función:- arg1: 120 -arg2: 6
      return: 720
      Reduce total 720
```

6. Funciones lambda

Python admite una interesante sintaxis que permite definir funciones mínimas, de una línea, sobre la marcha. Son funciones que no se definen previamente (no tienen nombre, también reciben el apodo de anónimas) y que pueden ser llamadas pasándoles un/varios parámetro/s. No hace falta definir el return, porque retorna automáticamente

Sentencia: `lambda nombreParametros (,otroPar,otro,...):cuerpoDeLaFuncion`

EJEMPLO: Presentación de las funciones lambda

```
>>> def f(x):
...     return x*2
...
>>> f(3)
6
>>> g = lambda x: x*2   ❶
>>> g(3)
6
>>> (lambda x: x*2)(3)  ❷
6
```

❶ Esta es una función `lambda` que consigue el mismo efecto que la función anterior. Aquí la sintaxis abreviada: la lista de argumentos no está entre paréntesis, y falta la palabra reservada `return` (está implícita, ya que la función entera debe ser una única expresión). Igualmente, la función no tiene nombre, pero puede ser llamada mediante la variable a que se ha asignado.

❷ Se puede utilizar una función `lambda` incluso sin asignarla a una variable. No es lo más útil del mundo, pero sirve para mostrar que una `lambda` es sólo una función en línea.

En general, una función lambda es una función que toma cualquier número de argumentos y devuelve el valor de una expresión simple. Las funciones lambda no pueden contener órdenes, y no pueden contener tampoco más de una expresión. Si se necesita algo más complejo, hay que definir en su lugar una función normal y hacerla tan larga como se requiera.

Ejemplos:

1. Ejem1

```
g = lambda x: x**2
print g(3)

print lambda x:x**2 (3)
```

Ambas llamadas `g(3)` y `lambda x:x**2 (3)` NO darán el mismo resultado. La primera devuelve el valor 9 y la segunda la construcción de la función sin resolver: `<function <lambda> at 0x010DC1F0>`

2. Ejem2

```
def make_incrementor (n): return lambda x: x + n
print """Estas sentencias colocan el valor n y devuelven la
construcción de la función lambda por ejemplo: x+2 ó x+6"""
f = make_incrementor(2)
g = make_incrementor(6)
print f
print g
print "Una vez asignada el valor de n, al llamar ahora el nuevo valor
se coloca en la x. Por ejemplo 42+2 ó 42+6"
print f(42), g(42)
print "Esta sentencia llama a la función y le pasa dos valores
separados entre corchetes. Uno es el parámetro de la función y el otro
el de la función lambda"
print make_incrementor(22)(33)
```

Resultado:

```
Estas sentencias colocan el valor n y devuelven la construcción de la
función lambda
    por ejemplo: x+2 ó x+6

<function <lambda> at 0x010DC230>
<function <lambda> at 0x010DC070>

Una vez asignada el valor de n, al llamar ahora el nuevo valor se
coloca en la x. Por ejemplo 42+2 ó 42+6

44 48

Esta sentencia llama a la función y le pasa dos valores separados
entre corchetes. Uno es el parámetro de la función y el otro el de la
función lambda
55
```

3. Ejem3:

```
a = lambda x, f, y: x**2
a(4,5,6)
```

Resultado: `16`

4. Ejemplo con listas y más parámetros:

```
print "Otra vuelta de tuerca. Con listas y funciones a listas"

foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]
print "Filtra la función lambda con los valores de la lista"
print filter(lambda x:x%3==0, foo)

print "A todos los valores de la lista les realiza la función"
print map(lambda x:x*2 + 10, foo)

print "Opera de 2 en 2 los elementos de la lista"
print reduce(lambda x,y:x+y, foo)
```

Resultado:

```
Otra vuelta de tuerca. Con listas y funciones a listas
Filtra la función lambda con los valores de la lista
[18, 9, 24, 12, 27]

A todos los valores de la lista les realiza la función
[14, 46, 28, 54, 44, 58, 26, 34, 64]

Opera de 2 en 2 los elementos de la lista
139
```

5. Otros ejemplos:

```
print map(lambda w: len(w),'El cielo esta enladrillado'.split())

print "Listas de acciones"
lista = [lambda x: x*2,lambda x:x*3]
numero = 4
for accion in lista:
    print accion(numero)

print "Generar nueva lista desde una original"
li = [1, 2, 3]
new_li = map(lambda x: x+2, li)
for item in new_li: print(item)
```

Resultado:

```
[2, 5, 4, 12]
Listas de acciones
8
12
Generar nueva lista desde una original
3
4
5
```

Fuente:

- http://es.diveintopython.net/apihelper_lambda.html
- https://jolthgs.wordpress.com/2012/02/12/funciones-lambda-en-python/
- http://recursospython.com/guias-y-manuales/funciones-lambda/