PROGRAMACIÓN ESTRUCTURADA Y MODULAR

FUNCIONES

Definición de funciones

Veamos un ejemplo de definición de función:

```
>>> def factorial(n):
...     """Calcula el factorial de un número"""
...     resultado = 1
...     for i in range(1,n+1):
...         resultado*=i
...     return resultado
```

Podemos obtener información de la función:

```
>>> help(factorial)
Help on function factorial in module __main__:
factorial(n)
    Calcula el factorial de un número
```

Y para utilizar la función:

```
>>> factorial(5)
120
```

Ámbito de variables. Sentencia global

Una variable local se declara en su ámbito de uso (en el programa principal y dentro de una función) y una global fuera de su ámbito para que se pueda utilizar en cualquier función que la declare como global.

```
>>> def operar(a,b):
...     global suma
...     suma = a + b
...     resta = a - b
...     print(suma,resta)
...
>>> operar(4,5)
9 -1
>>> resta
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'resta' is not defined
>>> suma
9
```

Podemos definir variables globales, que serán visibles en todo el módulo. Se recomienda declararlas en mayúsculas:

```
>>> PI = 3.1415
>>> def area(radio):
...     return PI*radio**2
...
>>> area(2)
12.566
```

Parámetros formales y reales

- Parámetros formales: Son las variables que recibe la función, se crean al definir la función. Su contenido lo recibe al realizar la llamada a la función de los parámetro reales. Los parámetros formales son variables locales dentro de la función.
- Parámetros reales: Son la expresiones que se utilizan en la llamada de la función, sus valores se copiarán en los parámetros formales.

Paso de parámetro por valor o por referencia

En Python el paso de parámetros es siempre por referencia. El lenguaje no trabaja con el concepto de variables sino objetos y referencias. Al realizar la asignación a = 1 no se dice que "a contiene el valor 1" sino que "a referencia a 1". Así, en comparación con otros lenguajes, podría decirse que en Python los parámetros siempre se pasan por referencia.

Evidentemente si se pasa un valor de un objeto inmutable, su valor no se podrá cambiar dentro de la función:

```
>>> def f(a):
...     a=5
>>> a=1
>>> f(a)
>>> a
1
```

Sin embargo si pasamos un objeto de un tipo mutable, si podremos cambiar su valor:

```
>>> def f(lista):
...   lista.append(5)
...
>>> l = [1,2]
>>> f(l)
>>> l
[1, 2, 5]
```

Llamadas a una función

Cuando se llama a una función se tienen que indicar los parámetros reales que se van a pasar. La llamada a una función se puede considerar una expresión cuyo valor y tipo es el retornado por la función. Si la función no tiene una instrucción `return` el tipo de la llamada sera `None`.

```
>>> def cuadrado(n):
...   return n*n

>>> a=cuadrado(2)
>>> cuadrado(3)+1
10
>>> cuadrado(cuadrado(4))
256
>>> type(cuadrado(2))
<class 'int'>
```

Cuando estamos definiendo una función estamos creando un objeto de tipo `function`.

```
>>> type(cuadrado)
<class 'function'>
```

Y por lo tanto puedo guardar el objeto función en otra variable:

```
>>> c=cuadrado
>>> c(4)
16
```

Tipos de argumentos: posicionales o keyword

Tenemos dos tipos de parámetros: los posiciónales donde el parámetro real debe coincidir en posición con el parámetro formal:

```
>>> def sumar(n1,n2):
...   return n1+n2
...
>>> sumar(5,7)
12
>>> sumar(4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sumar() missing 1 required positional argument: 'n2'
```

Además podemos tener parámetros con valores por defecto:

```
>>> def operar(n1,n2,operador='+',respuesta='El resultado es '):
...   if operador=="+":
...     return respuesta+str(n1+n2)
...   elif operador=="-":
...     return respuesta+str(n1-n2)
...   else:
...     return "Error"
...
>>> operar(5,7)
'El resultado es 12'
>>> operar(5,7,"-")
'El resultado es -2'
>>> operar(5,7,"-","La resta es ")
'La resta es -2'
```

Los parámetros keyword son aquellos donde se indican el nombre del parámetro formal y su valor, por lo tanto no es necesario que tengan la misma posición. Al definir una función o al llamarla, hay que indicar primero los argumentos posicionales y a continuación los argumentos con valor por defecto (keyword).

```
>>> operar(5,7)    # dos parámetros posicionales
>>> operar(n1=4,n2=6)    # dos parámetros keyword
>>> operar(4,6,respuesta="La suma es")    # dos parámetros posicionales y uno keyword
>>> operar(4,6,respuesta="La resta es",operador="-")    # dos parámetros posicionales y dos keyword
```

Parámetro *

Un parámetro * entre los parámetros formales de una función, nos obliga a indicar los parámetros reales posteriores como keyword:

```
>>> def sumar(n1,n2,*,op="+"):
...   if op=="+":
...     return n1+n2
...   elif op=="-":
...     return n1-n2
...   else:
...     return "error"
...
>>> sumar(2,3)
5
>>> sumar(2,3,"-")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sumar() takes 2 positional arguments but 3 were given
>>> sumar(2,3,op="-")
-1
```

Argumentos arbitrarios (*args y **kwargs)

Para indicar un número indefinido de argumentos posicionales al definir una función, utilizamos el símbolo *:

```
>>> def sumar(n,*args):
...   resultado=n
...   for i in args:
...     resultado+=i
...   return resultado
...
>>> sumar(2)
2
>>> sumar(2,3,4)
9
```

Para indicar un número indefinido de argumentos keyword al definir una función, utilizamos el símbolo **:

```
>>> def saludar(nombre="pepe",**kwargs):
...   cadena=nombre
...   for valor in kwargs.values():
...    cadena=cadena+" "+valor
...   return "Hola "+cadena
...
>>> saludar()
'Hola pepe'
>>> saludar("juan")
'Hola juan'
>>> saludar(nombre="juan",nombre2="pepe")
'Hola juan pepe'
>>> saludar(nombre="juan",nombre2="pepe",nombre3="maria")
'Hola juan maria pepe'
```

Por lo tanto podríamos tener definiciones de funciones del tipo:

```
>>> def f()
>>> def f(a,b=1)
>>> def f(a,*args,b=1)
>>> def f(*args,b=1)
>>> def f(*args,b=1,*kwargs)
>>> def f(*args,*kwargs)
>>> def f(*args)
>>> def f(*kwargs)
```

Desempaquetar argumentos: pasar listas y diccionarios

En caso contrario es cuando tenemos que pasar parámetros que lo tenemos guardados en una lista o en un diccionario.

Para pasar listas utilizamos el símbolo *:

```
>>> lista=[1,2,3]
>>> sumar(*lista)
6
>>> sumar(2,*lista)
8
>>> sumar(2,3,*lista)
11
```

Podemos tener parámetros keyword guardados en un diccionario, para enviar un diccionario utilizamos el símbolo **:

```
>>> datos={"nombre":"jose","nombre2":"pepe","nombre3":"maria"}
>>> saludar(**datos)
'Hola jose maria pepe'
```

Devolver múltiples resultados

La instrucción `return` puede devolver cualquier tipo de resultados, por lo tanto es fácil devolver múltiples datos guardados en una lista o en un diccionario. Veamos un ejemplo en que devolvemos los datos en una tupla:

```
>>> def operar(n1,n2):
...   return (n1+n2,n1-n2,n1*n2)

>>> suma,resta,producto = operar(5,2)
>>> suma
7
>>> resta
3
>>> producto
10
```

Funciones recursivas

Una función recursiva es aquella que al ejecutarse hace llamadas a ella misma. Por lo tento tenemos que tener "un caso base" que hace terminar el bucle de llamadas. Veamos un ejemplo:

```
>>> def factorial(numero):
...     if(numero == 0 or numero == 1):
...         return 1
...     else:
...         return numero * factorial(numero-1)
...
>>> factorial(5)
120
```

Funciones lambda

Las funciones lambda nos sirven para crear pequeñas funciones anónimas, de una sola línea sobre la marcha.

```
>>> cuadrado = lambda x: x**2
>>> cuadrado(2)
```

Como podemos notar las funciones lambda no tienen nombre. Pero gracias a que lambda crea una referencia a un objeto función, la podemos llamar.

```
>>> lambda x: x**2
<function <lambda> at 0xb74469cc>
>>>
>>> (lambda x: x**2)(3)
9
```

Otro ejemplo:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

Decoradores

Los decoradores son funciones que reciben como parámetros otras funciones y retornan como resultado otras funciones con el objetivo de alterar el funcionamiento original de la función que se pasa como parámetro. Hay funciones que tienen en común muchas funcionalidades, por ejemplo las de manejo de errores de conexión de recursos I/O (que se deben programar siempre que usemos estos recursos) o las de validación de permisos en las respuestas de peticiones de servidores, en vez de repetir el código de rutinas podemos abstraer, bien sea el manejo de error o la respuesta de peticiones, en una función decorador.

```
>>> def tablas(funcion):
...     def envoltura(tabla=1):
...         print('Tabla del %i:' %tabla)
...         print('-' * 15)
...         for numero in range(0, 11):
...             funcion(numero, tabla)
...         print('-' * 15)
...     return envoltura
...
>>> @tablas
... def suma(numero, tabla=1):
...     print('%2i + %2i = %3i' %(tabla, numero, tabla+numero))
...
>>> @tablas
... def multiplicar(numero, tabla=1):
...     print('%2i X %2i = %3i' %(tabla, numero, tabla*numero))

# Muestra la tabla de sumar del 1
suma()
# Muestra la tabla de sumar del 4
suma(4)
# Muestra la tabla de multiplicar del 1
multiplicar()
# Muestra la tabla de multiplicar del 10
multiplicar(10)
```

Funciones generadoras

Un generador es un tipo concreto de iterador. Es una función que permite obtener sus resultados paso a paso.

```
>>> def par(inicio,fin):
...   for i in range(inicio,fin):
...     if i % 2==0:
...       yield i

>>> datos = par(1,5)
>>> next(datos)
2
>>> next(datos)
4

>>> for i in par(20,30):
...   print(i,end=" ")
20 22 24 26 28

>>> lista_pares = list(par(1,10))
>>> lista_pares
[2, 4, 6, 8]
```

EJERCICIO PROPUESTO

1. Estructura un programa en Python que permita registrar alumnos almacenándolos en un diccionario. Queda a tu elección determinar los campos que se almacenarán en dicho diccionario.

   Realiza también una función de búsqueda que reciba un conjunto de parámetros keyword y devuelva los alumnos que coincidan con los criterios de búsqueda.

   No olvides controlar las excepciones correspondientes.

2. Realiza un ejercicio en Python que dada una lista de triángulos se almacene para cada elemento: coordenadas, longitud y área.
   Desarrolla las funciones necesarias para poder añadir triángulos a la lista (como mínimo tenemos que introducir las coordenadas), calcular el área de un triángulo dadas sus coordenadas, calcular la longitud de un triángulo dadas sus coordenadas.