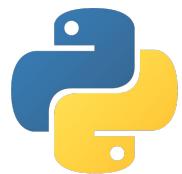


Programación Funcional

$f(x)$



Funciones

Las funciones son uno de los principales métodos de organización y reuso de código. Como regla general, cuando se necesita repetir una parte de código más de una vez. Es aconsejable escribir una función. Las funciones también ayudan a que el código sea más legible.

Funciones

La forma más sencilla de **definir** una función es de la siguiente manera:

```
def function():
    print('Hola Mundo!')
```

Colocamos la palabra reservada **def**, seguida del **nombre de la función**, dos **paréntesis** y dos **puntos**. A continuación se coloca lo que va a realizar la función en otro bloque de código.

Funciones

Para **llamar (ejecutar)** a una función se debe colocar el **nombre** de esta seguida de **dos paréntesis**.

```
>>> def funcion():
...     print('Hola Mundo!')
...
>>> funcion()
Hola Mundo!
>>>
```

Funciones

Las funciones también pueden retornar valores, esto se hace por medio de la palabra reservada `return`.

```
def suma():
    a = 4
    b = 3
    return a+b
```

Funciones

El valor que regresa una variables por lo general se **almacena** en **otra variable**.

```
>>> def suma():
...     a = 4
...     b = 3
...     return a+b
...
>>> c = suma()
>>> c
```

Funciones

Cuando se alcanza un `return`, las funciones terminan.

```
>>> def func():
...     a = 4
...     b = 3
...     return a+b
...     print('Hola')
...
>>> func()
7
```

Parámetros

Las funciones pueden recibir **parámetros**, estos van a definir las **acciones** o **resultados** que se van a obtener.

En este caso los parámetros de la función son tres números a, b y c.

De esta manera, es necesario llamar a la función con los tres números, en caso contrario, habrá un error.

```
>>> multiplicar(2,3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: multiplicar() missing 1 required positional argument: 'c'
```

```
>>> def multiplicar(a,b,c):
...     return a*b*c
...
>>> multiplicar(2,3,2)
12
>>> multiplicar(1,1,1)
1
>>> multiplicar(4,4,1)
16
```

Argumentos

En el caso de esta función, los **parámetros** serían a y b, al llamarla, los números 3 y 2 se les conoce como **argumentos**.

Estos son los **valores** que le **pasamos a la función** para que pueda operar con ellos. Los parámetros son las variables que ponemos en la definición. El orden en el que ponemos los argumentos afecta el resultado.

```
>>> def operacion(a,b):  
...     return(a-b)  
...  
>>> operacion(3,2)  
1  
>>> operacion(2,3)  
-1  
>>>
```

Argumentos

Se puede pasar los argumentos de una función en **desorden** si se **especifican** a qué **parámetro** van dirigidos. A estos argumentos se les denomina **argumentos por llave** o '**keyword arguments**'.

```
>>> def saludar(nombre1,nombre2):
...     print('Hola ' + nombre1 + ' y ' + nombre2)
...
>>> saludar('Ana','Jorge')
Hola Ana y Jorge
>>> saludar(nombre2 = 'Ana', nombre1= 'Jorge')
Hola Jorge y Ana
>>>
```

Argumentos

Es importante que si vamos a **especificar el valor** de los argumentos, estos se deben colocar después de los **posicionales**.

```
>>> def funcion(a,b,c):
...     return a*b*c
...
>>> funcion(1,b=2,c=3)
6
>>> funcion(a=1,3,4)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
>>>
```

Argumentos por default

Al definir una función, podemos especificar ciertos **parámetros** que van a tener un **valor por default** en caso de que el usuario no especifique un valor, estos se colocan **después de todos los parámetros posicionales**.

```
>>> def funcion(a,b,c=4):
...     return a+b+c
...
>>> funcion(2,3)
9
>>> funcion(2,3,1)
6
>>>
```

Argumentos por default

Si tratamos de definir una función con un **argumento por default** antes de un **parámetro posicional**, habría un **error**.

```
>>> def funcion(a=1,b,c):  
...     return a+b+c  
...  
File "<stdin>", line 1  
SyntaxError: non-default argument follows default argument  
>>>
```

args

Las funciones pueden recibir un número variable de argumentos, para hacer esto, debemos colocar un **parámetro al final** con un ***** antes de su nombre.

```
>>> def func(a,b,*c):
```

args

Con el ejemplo siguiente, se puede ver como los **argumentos posicionales que sobran** son almacenados en una tupla, que después puede ser utilizada en la función.

```
>>> def func(a,*c):
...     print(a)
...     print(c)
...     print(c*a)
...
>>> func(4,6,2,3,9)
4
(6, 2, 3, 9)
(6, 2, 3, 9, 6, 2, 3, 9, 6, 2, 3, 9)
>>>
```

args

El parámetro que va a recibir todos los argumentos posicionales que sobren y los va a guardar en una tupla, en general, se acostumbra nombrarlos como `args`.

```
>>> def func(a,b,*args):  
...     return args + (a,b)  
...  
>>> func(2,3,6,7,5,6,7)  
(6, 7, 5, 6, 7, 2, 3)  
>>>
```

kwargs

De igual forma, podemos pasar un **número variable** de **argumentos por llave desconocidos**, esto funciona de manera similar a cuando proporcionamos un número variable de argumentos posicionales, pero hay dos diferencias, antes de la variable, en lugar de colocar uno, se colocan **dos asteriscos**, y en vez de ser una tupla, va a ser un **diccionario**. Por lo general a esta variable se le nombra **kwargs**.

```
>>> def func(a,b,**kwargs):
```

kwargs

```
>>> def func(a,b,**kwargs):
...     print(a)
...     print(b)
...     print(kwargs)
...
>>> func(2,4,u=4,v=2,w=1)
2
4
{'u': 4, 'v': 2, 'w': 1}
>>>
```

Recursividad

Una función se puede llamar a sí misma, a esto se le denomina **recursión**. El ejemplo más sencillo es el siguiente:

Evitar correr el siguiente código

```
>>> def func():
...     func()
```

Recursividad

En la programación, la recursividad es una manera de solucionar un problema que puede llegar a ser muy eficiente, sin embargo, poder prever el comportamiento que va a tener es de un grado de dificultad alto, es por este motivo que de ser posible, en Python se recomienda tomar medidas alternas a la recursividad.

Recursividad

Un ejemplo de la aplicación de la recursividad es con un código que genere la serie de Fibonacci.

```
def fib(n):
    if n == 0 or n == 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Generadores

Los generadores son muy similares a las funciones, de hecho, la definición de ambos es casi idéntica. Lo que diferencia a los generadores de las funciones es que, las funciones regresan los valores en un único return y terminan, mientras que los generadores regresan los valores hasta que se les pide. Los generadores usan la palabra reservada yield en lugar de return.

Generadores

Las funciones retornan todos los valores al mismo tiempo. Esta retorna una lista con valores completa.

```
>>> def func():
...     lista = []
...     for i in range(4):
...         lista.append(i)
...     return lista
...
>>> list = func()
>>> list
[0, 1, 2, 3]
```

Generadores

Los **generadores** nos permiten obtener los **valores hasta que se los pidamos**, en primer instancia, los generadores en vez de **return** tienen **yield**. Y para pedir los valores usamos la función **next**.

En el siguiente ejemplo, con **yield** especificamos que queremos que los valores 0, 1, 2 y 3 sean retornados cuando se soliciten, después con la función **next** los vamos solicitando.

```
>>> def gen():
...     yield 0
...     yield 1
...     yield 2
...     yield 3
...
>>> generador = gen()
>>> next(generador)
0
>>> next(generador)
1
>>> next(generador)
2
>>> next(generador)
3
```

Generadores

Los generadores **retienen la ejecución** de su código hasta el punto donde está el primer **yield**, cuando se le vuelve a pedir el siguiente valor, su ejecución continúa hasta el siguiente **yield**.

```
>>> def saludar(*nombres):
...     for nom in nombres:
...         yield('Hola ' + nom)
...         print('Siguiente..')
...
>>> gen = saludar('Jorge', 'Ana', 'Hugo', 'Jessica')
>>> next(gen)
'Hola Jorge'
>>> next(gen)
Siguiente..
'Hola Ana'
>>> next(gen)
Siguiente..
'Hola Hugo'
```

Generadores

Los **generadores** pueden colocarse como los objetos **iterables** en un **ciclo for**.

```
>>> def gen():
...     yield 'a'
...     yield 1
...     yield 4 + 5j
...     yield (2,3,4)
...
>>> for valor in gen():
...     print(valor)
...
a
1
(4+5j)
(2, 3, 4)
>>> ■
```

Built-ins

Python cuenta con muchas **funciones ya establecidas** que ayudan a solucionar ciertos problemas de manera sencilla, sin embargo existen algunas que se utilizan particularmente en la **programación funcional**. Estas son **zip**, **enumerate**, **map** y **filter**.

zip

Nos permite combinar los elementos de dos o más arreglos en un grupo de tuplas, donde las tuplas se conforman de los elementos que comparten el mismo índice. (el primer elemento del primer arreglo con el primer elemento del segundo arreglo).

```
>>> lista1 = [4,5,6,7]
>>> lista2 = ['d','e','f','g']
>>> for tupla in zip(lista1,lista2):
...     print(tupla)
...
(4, 'd')
(5, 'e')
(6, 'f')
(7, 'g')
>>>
```

enumerate

Con enumerate podemos realizar una **cuenta de los elementos de un objeto** sobre el que estemos iterando. Particularmente, en un ciclo for.

```
>>> tupla = ('a','c','r','q','s','a')
>>> for num,elem in enumerate(tupla):
...     print(num, ' -> ', elem)
...
0  ->  a
1  ->  c
2  ->  r
3  ->  q
4  ->  s
5  ->  a
>>>
```

map

Con `map` podemos utilizar los **elementos** de un **arreglo** como **parámetros** de una función.

```
>>> def func(x):
...     return x**2 + 2
...
>>> lista = [2,4,6,8]
>>> for i in map(func,lista):
...     print(i)
...
6
18
38
66
>>>
```

filter

Nos permite filtrar los valores de un arreglo, para hacer uso de este debemos definir una función que reciba como parámetro los valores del arreglo y retorne true o false, dependiendo de una condición. Cuando la función retorne false, el elemento usado será filtrado.

```
>>> def func(x):
...     if x > 2:
...         return True
...     else:
...         return False
...
>>> for i in filter(func,[0,1,2,3,4,5]):
...     print(i)
...
3
4
5
>>>
```

Lista por comprensión

Muchas veces necesitamos **llenar** el contenido de una **lista** y para este propósito utilizamos un **ciclo for**.

```
>>> lista = []
>>> for num in range(4):
...     lista.append(num**2)
...
>>> lista
[0, 1, 4, 9]
>>>
```

Puesto que esta operación es muy común, se creó una manera de realizarlo con **menos líneas** de código y de manera **más eficiente**. Esto es por medio de una **lista por comprensión**.

Lista por comprensión

```
>>> lista = [num**2 for num in range(4)]  
>>> lista  
[0, 1, 4, 9]  
>>>
```

Se puede decir que la **sintaxis** de una lista por comprensión es **similar a un ciclo for dentro de una lista**, donde el elemento a la izquierda (`num**2`) es que va a irse agregando a la lista cada ciclo. Los valores que va a adquirir `num` son 0,1,2 y 3, los cuadrados de estos van a ser agregados a la lista.

Lista por comprensión

Opcionalmente podemos colocar un **filtro** a los **elementos** que vamos a colocar en la **lista**. Esto se realiza de la siguiente manera.

```
>>> lista = [num for num in range(6) if num%2 == 0]
>>> lista
[0, 2, 4]
>>>
```

Equivalente a:

```
>>> lista = []
>>> for num in range(6):
...     if num%2 == 0:
...         lista.append(num)
...
>>> lista
[0, 2, 4]
>>>
```

Lambdas

Son una forma especial de crear pequeñas funciones de una sola línea. También se les denomina funciones anónimas puesto que se definen de tal forma que no tienen un nombre explícito.

Lambdas

La sintaxis de una función lambda es la siguiente:

```
>>> lambda x,y : x+y+3
```

Primero se coloca la palabra reservada **lambda**, seguido de los **parámetros** (**x,y**), **dos puntos**, y finalmente lo que va a **retornar** (**x+y+3**).

Lambdas

```
>>> (lambda x,y : x+y+3)(4,5)  
12
```

En este ejemplo se puede ver una de las características principales de las funciones lambda, y es que no necesitamos asignarle ningún nombre para poder utilizarla, es por esto que se les conoce como funciones anónimas. De igual forma, si quisiéramos asignar un nombre, podríamos hacerlo.

```
>>> func = lambda x,y : x+y+3  
>>> func(4,5)  
12
```