Python VI: Funciones

Una función es una unidad operacional con una estructura y un propósito definidos, que nos permitirá empezar a sacar provecho de la **estructura modular** que tenemos en mente al programar en Python.

Ya nos hemos topado con muchas funciones hasta ahora; algunos ejemplos son `print`, `open`, `sum`, `sqrt` en el módulo `math`, `chdir` en el módulo `os`, etc. Todas ellas siguen una sintaxis en particular: una función `f` se ejecuta escribiendo el nombre de la función y entregándole una serie de argumentos (o uno o ninguno, en algunos casos) entre paréntesis:

```
f(...)
```

En esta clase aprenderemos a crear nuestras propias funciones y trabajar con ellas.

Definición

Empecemos por una operación que queremos ejecutar:

```
x = 4

doble = 2*x

doble

> 8
```

Podemos escribirlo como una función de la siguiente manera:

```
def doble():

    x = 4

    y = 2*x

    return y

doble()

> 8
```

Podemos hacer una función mucho más útil si pedimos x como *argumento*:

```
def doble(x):

    y = 2*x
```

```
    return y

doble(4)

> 8

doble(6)

> 12
```

o, de manera más compacta,

```
def doble(x):

    return 2*x
```

Es decir, una función toma argumentos de entrada y opera sobre ellos para producir valores de salida. Es importante que, tal como con las variables, el nombre de una función sea descriptivo de su objetivo. Por convención, las funciones tienen nombres con todas las letras en minúscula y, de ser necesario, las palabras deben separarse con guiones bajos (idealmente no más de 3 palabras).

Incluso, podemos pedir el prefactor como argumento también, para hacer una función genérica de multiplicación:

```
def mult(x, y):

    return x*y

mult(4, 3)

> 12
```

Las funciones pueden trabajar con cualquier tipo de objeto, no solamente numéricos:

```
def sufijo(raiz, suf):

    return f'{raiz}_{suf}'
```

Las funciones también pueden entregar más de un resultado. Un ejemplo de esto que ya habíamos visto es la función `str.split()`. Podemos lograr esto simplemente haciendo:

```
def doble_y_mitad(x):

    return 2*x, x/2
```

Técnicamente, el resultado es una tupla (arriba, los paréntesis son implícitos):

```
doble_y_mitad(3)
```

```
> (6, 1.5)
```

que también podemos extraer individualmente:

```
doble, mitad = doble_y_mitad(9)

doble

> 18

mitad

> 4.5
```

Noten que el primer valor es `int`, mientras que el segundo es `float`. **Los valores que entrega una función no necesitan ser del mismo tipo.** Ésta es otra de las cualidades que diferencian a Python de otros lenguajes, y es en realidad resultado de la flexibilidad de la tupla. También podríamos querer producir una lista en lugar de una tupla, por supuesto:

```
def doble_y_mitad(x):

     return [2*x, x/2]
```

Para que la función opere correctamente, debemos entregarle todos los argumentos pedidos, y ni uno más:

```
doble_y_mitad()

> TypeError: doble_y_mitad() missing 1 required positional argument: 'x'

doble_y_mitad(3, 4)

> TypeError: doble_y_mitad() takes 1 positional argument but 2 were given
```

Pero podemos permitir argumentos opcionales, definiéndolos con un valor predeterminado:

```
def potencia(a, b=2):

     return a**b

potencia(3)

> 9

potencia(2, 3)

> 8
```

Podemos comprobar que asignar el valor predeterminado a un argumento opcional es igual a no especificarlo:

```
potencia(3, 2) == potencia(3)

> True
```

Puede haber tantos argumentos obligatorios u opcionales como queramos, pero los argumentos obligatorios deben definirse todos siempre antes de los opcionales. Podemos elegir qué argumentos opcionales nos interesa especificar, usando sus nombres:

```
def info(a, b, c=0, d=0):
    return f'a={a}, b={b}, c={c}, d={d}'

info(1, 2)

> 'a=1, b=2, c=0, d=0'

info(1, 2, 3)

> 'a=1, b=2, c=3, d=0'

info(1, 2, c=3)

> 'a=1, b=2, c=3, d=0'

info(1, 2, d=3)

> 'a=1, b=2, c=0, d=3'
```

Por supuesto, podemos incorporar a una función todo lo que hemos aprendido hasta ahora. Por ejemplo, podemos implementar nuestra propia función “valor absoluto”:

```
def valor_absoluto(x):
    if x < 0:
        x_abs = -x
    else:
        x_abs = x
    return x_abs

valor_absoluto(-4)

> 4
```

o en versión compacta:

```
def valor_absoluto(x):
    return x if x >= 0 else -x
```

```
valor_absoluto(-3)

> 3
```

Si nuestra función es suficientemente simple (i.e., si puede ser definida en una línea), podemos usar funciones *lambda*:

```
potencia = lambda a, b=1: a**b

potencia(4, 2)

> 16
```

Documentación

Ya dijimos que el nombre de la función debe ser descriptivo, pero a veces necesitamos información en detalle sobre cómo ejecutar una función. Ya vimos que podemos acceder a esta información con el comando help:

```
help(abs)

> abs(x, /)

>    Return the absolute value of the argument.
```

Podemos agregar información como ésta muy fácilmente a nuestra función:

```
def valor_absoluto(x):
     """Entrega el valor absoluto del argumento"""
     if x < 0:
          x_abs = -x
     else:
          x_abs = x
     return x_abs
help(valor_absoluto)

> valor_absoluto(x)

>    Entrega el valor absoluto del argumento
```

Ahora que comenzamos a escribir texto u operaciones más largas y complejas, es bueno tener en cuenta las siguientes convenciones (ver PEP8):

- Una línea no debe superar los 79 caracteres
- Una línea que contiene comentarios no debe superar los 72 caracteres
- Usar espacios entre operadores y variables, aunque a veces pueden suprimirse para indicar "jerarquía" de operaciones (ej, `2 * (x+y)`)

Pero, se deben tener en cuenta las siguientes afirmaciones del Zen de Python:

> ***La legibilidad es importante***, y
> ***La practicalidad le gana a la pureza***

Podemos usar esta documentación para dar tanto detalle como sea necesario (revisemos por ejemplo la documentación de `os.listdir`):

```
def potencia(x, a, b=1):
     """Elevar a una potencia, con un prefactor

     Parametros
     ----------
     x : float
          base
     a : float
          exponente

     Parametros opcionales
     ---------------------
     b : float
          prefactor

     Entrega
     -------
     potencia : float
          resultado de b*x**a
     """
     return b * x**a
```

Cada uno puede encontrar una convención que le parezca clara; yo generalmente escribo la documentación de mis funciones más o menos así. Es útil especificar el tipo de variable que se espera, además de lo que significa cada una. Noten que el valor predeterminado puede leerse en la declaración misma por lo que no es realmente necesario especificarlo en la descripción - pero hacerlo no está mal tampoco. El objetivo de la documentación es que el usuario pueda saber en detalle

para qué sirve y cómo se llama una función, sin necesidad de revisar el código fuente.

Aunque en python las variables no se declaran antes de ser definidas, podemos usar “sugerencias de tipo” (“type hints”) en la definición de una función, tanto para los argumentos como para el resultado

```
def potencia(x: float, a: float, b: float=1) -> float:
    return a * x**b
```

Pero esto es sólo ayuda para la documentación (una sugerencia!); no es una condición para la ejecución de la función:

```
potencia(2, 1, 'a')
> 'aa'
```

Forzar consistencia

Para hacer esto, debemos agregar pruebas de consistencia a nuestra función. Dependiendo del tipo de argumento que requiere nuestra función, tenemos muchas alternativas para hacer esto.

- Si estamos seguros del tipo de variable que requiere nuestra función, podemos usar la función nativa isinstance:

  ```
  isinstance('string', str)
  > True
  ```

  pero para muchos tipos de variable esto puede resultar muy restrictivo. Por ejemplo,

  ```
  isinstance(3, float)
  > False
  isinstance(3.0, int)
  > False
  ```

  A veces puede ser que queramos este comportamiento, pero muchas otras veces no.

- En el caso de iterables, podemos usar

  ```
  hasattr([1, 2, 3], '__iter__')
  ```

```
> True
```

pero:

```
hasattr('string', '__iter__')

> True
```

(ya veremos qué significa esto).

- Para funciones,

```
callable(potencia)

> True

callable(lambda x: x**2)

> True
```

- Una declaración muy útil en este contexto es la declaración **`assert`** (“asegurar”):

```
assert isinstance(3, float)

> AssertionError:
```

El mensaje de error debemos entregarlo nosotros:

```
assert isinstance(3, float), 'Variable debe ser float'

> AssertionError: 'Variable debe ser float'
```

Si la condición se cumple, el programa sigue adelante sin más:

```
assert isinstance(3, int)

>
```

Noten que hay algo muy importante que no está escrito en este documento: el *rastro* de un error, que nos indica dónde hemos cometido el error:

```
assert isinstance(3, float), 'Variable debe ser float'

---------------------------------------------------------
-----------------

AssertionError

Traceback (most recent call last)
```

```
<ipython-input-66-cbc01b6577a7> in <module>

----> 1 assert isinstance(3, float), 'Variable debe ser
float'


AssertionError: Variable debe ser float
```

Queda claro dónde cometimos el error, y esto facilita mucho la tarea de corregirlo. En el caso de las declaraciones assert, incluso queda muy claro qué es lo que se busca! Pero en este documento los omitimos para hacerlo más breve.

Una función también puede recibir otra función:

```
def doble(func, x, y):
     return 2 * func(x, y)
doble(potencia, 2, 3)
> 16
```

y/o entregarla:

```
def funcion_por_nombre(nombre):
     if nombre == 'min':
          return min
     elif nombre == 'max':
          return max
type(funcion_por_nombre('min'))
> function
```

En casos como éstos, puede ser útil usar los constructores * y **, que sirven para expandir listas o tuplas y diccionarios, respectivamente:

```
x = [1, 2, 3]
print(x)
> [1, 2, 3]
print(*x)
> 1 2 3
```

```
print(1, 2, 3)

> 1 2 3

y = {'a': 1, 'b': 2, 'c': 3}

{**y}

> {'a': 1, 'b': 2, 'c': 3}

{y}

> TypeError: unhashable type: 'dict'
```

El motivo por el que estos constructores son convenientes en este contexto, es que se pueden usar para agrupar los argumentos obligatorios y opcionales de una función. Si quisiéramos construir una función que recibe y utiliza los argumentos de otra función, podríamos hacerlo así:

```
def potencia(a, b, n=1):
    return n * a**b

def doble(func, args, kwargs):
    return 2 * func(*args, **kwargs)
```

(Los nombres `args` y `kwargs` se usan por convención, para *arguments* y *keyword arguments*, pero pueden recibir cualquier nombre como cualquier otra variable.) La manera de llamar la función `doble` sería algo así:

```
doble(potencia, (3, 2), {'n': 4})

> 72
```

Sin embargo, podemos facilitarnos la vida definiendo:

```
def doble(func, *args, **kwargs):
    return 2 * func(*args, **kwargs)
```

de manera que la función doble espera una lista/tupla y un diccionario *desagregados*:

```
doble(potencia, 3, 2, n=3)

> 72
```

`args` sigue siendo una tupla, y `kwargs` un diccionario:

```
def doble(func, *args, **kwargs):
```

```
        print(args)
        print(kwargs)
        return 2 * func(*args, **kwargs)
doble(potencia, 3, 2, n=4)
> (3, 2)
> {'n': 4}
> 72
```