Lectura 2: El tutorial de Python (páginas 19 a 25)

En este material de lectura veremos como definir funciones, veremos las distintas formas de pasar los argumentos y como definir funciones anónimas o expresiones lambda. Para cada uno de estos temas se mostrarán ejemplos.

Por otro lado se verán las cadenas de texto de documentación (o docstrings), las anotaciones de funciones y el estilo de codificación.

Cantidad de páginas: 7.

Definiendo funciones

Podemos crear una función que escriba la serie de Fibonacci hasta un límite determinado:

```
>>> def fib(n):  # escribe la serie de Fibonacci hasta n
...     """Escribe la serie de Fibonacci hasta n."""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Ahora llamamos a la funcion que acabamos de definir:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

La palabra reservada `def` se usa para *definir* funciones. Debe seguirle el nombre de la función y la lista de parámetros formales entre paréntesis. Las sentencias que forman el cuerpo de la función empiezan en la línea siguiente, y deben estar con sangría.

La primer sentencia del cuerpo de la función puede ser opcionalmente una cadena de texto literal; esta es la cadena de texto de documentación de la función, o *docstring*. (Podés encontrar más acerca de docstrings en la sección *Cadenas de texto de documentación*.)

Hay herramientas que usan las docstrings para producir automáticamente documentación en línea o imprimible, o para permitirle al usuario que navegue el código en forma interactiva; es una buena práctica incluir docstrings en el código que uno escribe, por lo que se debe hacer un hábito de esto.

La *ejecución* de una función introduce una nueva tabla de símbolos usada para las variables locales de la función. Más precisamente, todas las asignaciones de variables en la función almacenan el valor en la tabla de símbolos local; así mismo la referencia a variables primero mira la tabla de símbolos local, luego en la tabla de símbolos local de las funciones externas, luego la tabla de símbolos global, y finalmente la tabla de nombres predefinidos. Así, no se les puede asignar directamente un valor a las variables globales dentro de una función (a menos se las nombre en la sentencia `global`), aunque si pueden ser referenciadas.

Los parámetros reales (argumentos) de una función se introducen en la tabla de símbolos local de la función llamada cuando esta es ejecutada; así, los argumentos son pasados *por valor* (dónde el *valor* es siempre una *referencia* a un objeto, no el valor del objeto). [4] Cuando una función llama a otra función, una nueva tabla de símbolos local es creada para esa llamada.

La definición de una función introduce el nombre de la función en la tabla de símbolos actual. El valor del nombre de la función tiene un tipo que es reconocido por el interprete como una función definida por el usuario. Este valor puede ser

asignado a otro nombre que luego puede ser usado como una función. Esto sirve como un mecanismo general para renombrar:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Viniendo de otros lenguajes, podés objetar que `fib` no es una función, sino un procedimiento, porque no devuelve un valor. De hecho, técnicamente hablando, los procedimientos sí retornan un valor, aunque uno aburrido. Este valor se llama `None` (es un nombre predefinido). El intérprete por lo general no escribe el valor `None` si va a ser el único valor escrito. Si realmente se quiere, se puede verlo usando la función **`print()`**:

```
>>> fib(0)
>>> print(fib(0))
None
```

Es simple escribir una función que retorne una lista con los números de la serie de Fibonacci en lugar de imprimirlos:

```
>>> def fib2(n): # devuelve la serie de Fibonacci hasta n
...     """Devuelve una lista conteniendo la serie de Fibonacci hasta n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a)    # ver abajo
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100)    # llamarla
>>> f100                # escribir el resultado
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Este ejemplo, como es usual, demuestra algunas características más de Python:

- La sentencia **`return`** devuelve un valor en una función. **`return`** sin una expresión como argumento retorna `None`. Si se alcanza el final de una función, también se retorna `None`.
- La sentencia `result.append(a)` llama a un *método* del objeto lista `result`. Un método es una función que 'pertenece' a un objeto y se nombra `obj.methodname`, dónde `obj` es algún objeto (puede ser una expresión), y `methodname` es el nombre del método que está definido por el tipo del objeto. Distintos tipos definen distintos métodos. Métodos de diferentes tipos pueden tener el mismo nombre sin causar ambigüedad. (Es posible definir tipos de objetos propios, y métodos, usando *clases*, mirá *Clases*). El método **`append()`** mostrado en el ejemplo está definido para objetos lista; añade un nuevo elemento al final de la lista. En este ejemplo es equivalente a `result = result + [a]`, pero más eficiente.

Más sobre definición de funciones

También es posible definir funciones con un número variable de argumentos. Hay tres formas que pueden ser combinadas.

Argumentos con valores por omisión

La forma más útil es especificar un valor por omisión para uno o más argumentos. Esto crea una función que puede ser llamada con menos argumentos que los que permite. Por ejemplo:

```
def pedir_confirmacion(prompt, reintentos=4, recordatorio='Por favor, intente nuevamente!'):
    while True:
        ok = input(prompt)
        if ok in ('s', 'S', 'si', 'Si', 'SI'):
```

```
            return True
        if ok in ('n', 'no', 'No', 'NO'):
            return False
        reintentos = reintentos - 1
        if reintentos < 0:
            raise ValueError('respuesta de usuario inválida')
        print(recordatorio)
```

Esta función puede ser llamada de distintas maneras:

- pasando sólo el argumento obligatorio: `pedir_confirmacion('¿Realmente queres salir?')`
- pasando uno de los argumentos opcionales: `pedir_confirmacion('¿Sobreescribir archivo?', 2)`
- o pasando todos los argumentos: `pedir_confirmacion('¿Sobreescribir archivo?', 2, "Vamos, solo si o no!)`

Este ejemplo también introduce la palabra reservada `in`, la cual prueba si una secuencia contiene o no un determinado valor.

Los valores por omisión son evaluados en el momento de la definición de la función, en el ámbito de la *definición*, entonces:

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

...imprimirá `5`.

Advertencia importante: El valor por omisión es evaluado solo una vez. Existe una diferencia cuando el valor por omisión es un objeto mutable como una lista, diccionario, o instancia de la mayoría de las clases. Por ejemplo, la siguiente función acumula los argumentos que se le pasan en subsiguientes llamadas:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

Imprimirá:

```
[1]
[1, 2]
[1, 2, 3]
```

Si no se quiere que el valor por omisión sea compartido entre subsiguientes llamadas, se pueden escribir la función así:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

Palabras claves como argumentos

Las funciones también puede ser llamadas usando argumentos de palabras clave (o argumentos nombrados) de la forma `keyword = value`. Por ejemplo, la siguiente función:

```
def loro(tension, estado='muerto', accion='explotar', tipo='Azul Nordico'):
    print("-- Este loro no va a", accion, end=' ')
    print("si le aplicás", tension, "voltios.")
    print("-- Gran plumaje tiene el", tipo)
    print("-- Está", estado, "!")
```

...acepta un argumento obligatorio (`tension`) y tres argumentos opcionales (`estado`, `accion`, y `tipo`). Esta función puede llamarse de cualquiera de las siguientes maneras:

```
loro(1000)                                          # 1 argumento posicional
loro(tension=1000)                                  # 1 argumento nombrado
loro(tension=1000000, accion='VOOOOOM')             # 2 argumentos nombrados
loro(accion='VOOOOOM', tension=1000000)             # 2 argumentos nombrados
loro('un millón', 'despojado de vida', 'saltar')    # 3 args posicionales
loro('mil', estado='viendo crecer las flores desde abajo')  # uno y uno
```

...pero estas otras llamadas serían todas inválidas:

```
loro()                      # falta argumento obligatorio
loro(tension=5.0, 'muerto') # argumento posicional luego de uno nombrado
loro(110, tension=220)      # valor duplicado para el mismo argumento
loro(actor='Juan Garau')    # nombre del argumento desconocido
```

En una llamada a una función, los argumentos nombrados deben seguir a los argumentos posicionales. Cada uno de los argumentos nombrados pasados deben coincidir con un argumento aceptado por la función (por ejemplo, `actor` no es un argumento válido para la función `loro`), y el orden de los mismos no es importante. Esto también se aplica a los argumentos obligatorios (por ejemplo, `loro(tension=1000)` también es válido). Ningún argumento puede recibir más de un valor al mismo tiempo. Aquí hay un ejemplo que falla debido a esta restricción:

```
>>> def funcion(a):
...     pass
...
>>> funcion(0, a=0)
Traceback (most recent call last):
...
TypeError: funcion() got multiple values for keyword argument 'a'
```

Cuando un parámetro formal de la forma `**nombre` está presente al final, recibe un diccionario (ver *Tipos integrados*) conteniendo todos los argumentos nombrados excepto aquellos correspondientes a un parámetro formal. Esto puede ser combinado con un parámetro formal de la forma `*nombre` (descripto en la siguiente sección) que recibe una tupla conteniendo los argumentos posicionales además de la lista de parámetros formales. `*nombre` debe ocurrir antes de `**nombre`). Por ejemplo, si definimos una función así:

```
def ventadequeso(tipo, *argumentos, **palabrasclaves):
    print("-- ¿Tiene", tipo, "?")
    print("-- Lo siento, nos quedamos sin", tipo)
    for arg in argumentos:
        print(arg)
    print("-" * 40)
    for c in palabrasclaves:
        print(c, ":", palabrasclaves[c])
```

Puede ser llamada así:

```
ventadequeso("Limburger", "Es muy liquido, sr.",
             "Realmente es muy muy liquido, sr.",
             cliente="Juan Garau",
             vendedor="Miguel Paez",
             puesto="Venta de Queso Argentino")
```

...y por supuesto imprimirá:

```
.. code-block:: none
```

> -- ¿Tiene Limburger ? -- Lo siento, nos quedamos sin Limburger Es muy liquido, sr. Realmente es muy muy liquido, sr. ---------- cliente : Juan Garau vendedor : Miguel Paez puesto : Venta de Queso Argentino

Se debe notar que el orden en el cual los argumentos nombrados son impresos está garantizado para coincidir con el orden en el cual fueron provistos en la llamada a la función.

Listas de argumentos arbitrarios

Finalmente, la opción menos frecuentemente usada es especificar que una función puede ser llamada con un número arbitrario de argumentos. Estos argumentos serán organizados en una tupla (mirá *Tuplas y secuencias*). Antes del número variable de argumentos, cero o más argumentos normales pueden estar presentes.:

```
def muchos_items(archivo, separador, *args):
    archivo.write(separador.join(args))
```

Normalmente estos argumentos de cantidad variables son los últimos en la lista de parámetros formales, porque toman todo el remanente de argumentos que se pasan a la función. Cualquier parámetro que suceda luego de `*args` será 'sólo nombrado', o sea que sólo se pueden usar como nombrados y no posicionales.:

```
>>> def concatenar(*args, sep="/"):
...     return sep.join(args)
...
>>> concatenar("tierra", "marte", "venus")
'tierra/marte/venus'
>>> concatenar("tierra", "marte", "venus", sep=".")
'tierra.marte.venus'
```

Desempaquetando una lista de argumentos

La situación inversa ocurre cuando los argumentos ya están en una lista o tupla pero necesitan ser desempaquetados para llamar a una función que requiere argumentos posicionales separados. Por ejemplo, la función predefinida **`range()`** espera los argumentos *inicio* y *fin*. Si no están disponibles en forma separada, se puede escribir la llamada a la función con el operador para desempaquetar argumentos de una lista o una tupla `*`::

```
>>> list(range(3, 6))    # llamada normal con argumentos separados
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))  # llamada con argumentos desempaquetados de la lista
[3, 4, 5]
```

Del mismo modo, los diccionarios pueden entregar argumentos nombrados con el operador `**`::

```
>>> def loro(tension, estado='rostizado', accion='explotar'):
...     print("-- Este loro no va a", accion, end=' ')
...     print("si le aplicás", tension, "voltios.", end=' ')
...     print("Está", estado, "!")
...
>>> d = {"tension": "cinco mil", "estado": "demacrado",
...      "accion": "VOLAR"}
```

```
>>> loro(**d)
-- Este loro no va a VOLAR si le aplicás cinco mil voltios. Está demacrado !
```

Expresiones lambda

Pequeñas funciones anónimas pueden ser creadas con la palabra reservada **`lambda`**. Esta función retorna la suma de sus dos argumentos: `lambda a, b: a + b` . Las funciones Lambda pueden ser usadas en cualquier lugar donde sea requerido un objeto de tipo función. Están sintácticamente restringidas a una sola expresión. Semánticamente, son solo azúcar sintáctica para definiciones normales de funciones. Al igual que las funciones anidadas, las funciones lambda pueden hacer referencia a variables desde el ámbito que la contiene:

```
>>> def hacer_incrementador(n):
...     return lambda x: x + n
...
>>> f = hacer_incrementador(42)
>>> f(0)
42
>>> f(1)
43
```

Cadenas de texto de documentación

Acá hay algunas convenciones sobre el contenido y formato de las cadenas de texto de documentación.

La primer línea debe ser siempre un resumen corto y conciso del propósito del objeto. Para ser breve, no se debe mencionar explícitamente el nombre o tipo del objeto, ya que estos están disponibles de otros modos (excepto si el nombre es un verbo que describe el funcionamiento de la función). Esta línea debe empezar con una letra mayúscula y terminar con un punto.

Si hay más líneas en la cadena de texto de documentación, la segunda línea debe estar en blanco, separando visualmente el resumen del resto de la descripción. Las líneas siguientes deben ser uno o más párrafos describiendo las convenciones para llamar al objeto, efectos secundarios, etc.

El analizador de Python no quita el sangrado de las cadenas de texto literales multi-líneas, entonces las herramientas que procesan documentación tienen que quitarlo si así lo desean. Esto se hace mediante la siguiente convención. La primer línea que no está en blanco *siguiente* a la primer línea de la cadena determina la cantidad de sangría para toda la cadena de documentación. (No podemos usar la primer línea ya que generalmente es adyacente a las comillas de apertura de la cadena y el sangrado no se nota en la cadena de texto). Los espacios en blanco "equivalentes" a este sangrado son luego quitados del comienzo de cada línea en la cadena. No deberían haber líneas con una sangría menor, pero si las hay todos los espacios en blanco del comienzo deben ser quitados. La equivalencia de espacios en blanco debe ser verificada luego de la expansión de tabs (a 8 espacios, normalmente).

Este es un ejemplo de un docstring multi-línea:

```
>>> def mi_funcion():
...     """No hace mas que documentar la funcion.
...
...     No, de verdad. No hace nada.
...     """
...     pass
...
>>> print(mi_funcion.__doc__)
No hace mas que documentar la funcion.

No, de verdad. No hace nada.
```

Anotación de funciones

+:ref:*Function annotations* <*function*> are completely optional metadata +information about the types used by user-defined functions (see **PEP 484** +for more information).

Las anotaciones de funciones son información completamente opcional sobre los tipos usadas en funciones definidas por el usuario (ver **PEP 484** para más información).

Las anotaciones se almacenan en el atributo `__annotations__` de la función como un diccionario y no tienen efecto en ninguna otra parte de la función. Las anotaciones de los parámetros se definen luego de dos puntos después del nombre del parámetro, seguido de una expresión que evalúa al valor de la anotación. Las anotaciones de retorno son definidas por el literal `->`, seguidas de una expresión, entre la lista de parámetros y los dos puntos que marcan el final de la declaración `def`. El siguiente ejemplo tiene un argumento posicional, uno nombrado, y el valor de retorno anotado:

```
>>> def f(jamon: str, huevos: str = 'huevos') -> str:
...     print("Anotaciones:", f.__annotations__)
...     print("Argumentos:", jamon, huevos)
...     return jamon + ' y ' + huevos
...
>>> f('carne')
Anotaciones: {'jamon': <class 'str'>, 'huevos': <class 'str'>, 'return': <class 'str'>}
Argumentos: carne huevos
'carne y huevos'
>>>
```

Intermezzo: Estilo de codificación

Ahora que estás a punto de escribir piezas de Python más largas y complejas, es un buen momento para hablar sobre *estilo de codificación*. La mayoría de los lenguajes pueden ser escritos (o mejor dicho, *formateados*) con diferentes estilos; algunos son mas fáciles de leer que otros. Hacer que tu código sea más fácil de leer por otros es siempre una buena idea, y adoptar un buen estilo de codificación ayuda tremendamente a lograrlo.

Para Python, **PEP 8** se erigió como la guía de estilo a la que más proyectos adhirieron; promueve un estilo de codificación fácil de leer y visualmente agradable. Todos los desarrolladores Python deben leerlo en algún momento; aquí están extraídos los puntos más importantes:

- Usar sangrías de 4 espacios, no tabs.

  4 espacios son un buen compromiso entre una sangría pequeña (permite mayor nivel de sangrado)y una sangría grande (más fácil de leer). Los tabs introducen confusión y es mejor dejarlos de lado.

- Recortar las líneas para que no superen los 79 caracteres.

  Esto ayuda a los usuarios con pantallas pequeñas y hace posible tener varios archivos de código abiertos, uno al lado del otro, en pantallas grandes.

- Usar líneas en blanco para separar funciones y clases, y bloques grandes de código dentro de funciones.

- Cuando sea posible, poner comentarios en una sola línea.

- Usar docstrings.

- Usar espacios alrededor de operadores y luego de las comas, pero no directamente dentro de paréntesis: `a = f(1, 2) + g(3, 4)`.

- Nombrar las clases y funciones consistentemente; la convención es usar `NotacionCamello` para clases y `minusculas_con_guiones_bajos` para funciones y métodos. Siempre usá `self` como el nombre para el primer argumento en los métodos (mirá *Un primer vistazo a las clases* para más información sobre clases y métodos).

- No uses codificaciones estrafalarias si esperás usar el código en entornos internacionales. El default de Python, UTF-8, o incluso ASCII plano funcionan bien en la mayoría de los casos.

- De la misma manera, no uses caracteres no-ASCII en los identificadores si hay incluso una pequeñísima chance de que gente que hable otro idioma tenga que leer o mantener el código.

---

4 En realidad, *llamadas por referencia de objeto* sería una mejor descripción, ya que si se pasa un objeto mutable, quien realiza la llamada verá cualquier cambio que se realice sobre el mismo (por ejemplo ítems insertados en una lista).