Funciones

March 30, 2022

Ejecuta el siguiente bloque de código siempre antes de ejecutar el resto del notebook.

[ ]:
```
from IPython.core.magic import  Magics, magics_class, cell_magic, line_magic

@magics_class
class Helper(Magics):

  def __init__(self, shell=None,  **kwargs):
    super().__init__(shell=shell, **kwargs)

  @cell_magic
  def debug_cell_with_pytutor(self, line, cell):
    import urllib.parse
    url_src = urllib.parse.quote(cell)
    str_begin = '<iframe width="1000" height="500" frameborder="0" src="https://
↪pythontutor.com/iframe-embed.html#code='
    str_end   = '&cumulative=false&py=3&curInstr=0"></iframe>'
    import IPython
    from google.colab import output
    display(IPython.display.HTML(str_begin+url_src+str_end))

get_ipython().register_magics(Helper)
```

1 Funciones

Hasta ahora, todas las instrucciones que hemos estudiado nos permiten resolver cualquier problema, pero debemos ejecutar el programa entero para obtener un resultado; si queremos llamar bloques de código dentro de otro programa, debemos usar funciones. Las funciones de Python son un bloque de sentencias relacionadas, diseñadas para realizar una tarea computacional, lógica o evaluativa. La idea es juntar algunas tareas que se hacen común o repetidamente y construir una función para "invocarla" cada vez que se requiera sin tener que recurrir a la reescritura de código en aquellas partes del programa donde se vuelva a necesitar.

Existen funciones incorporadas en Python y en sus librerías (ya hemos usado varias dentro del curso, por ejemplo, `len`) y también existen las funciones definidas por el usuario. Todas ellas ayudan a que el programa sea conciso, no repetitivo y organizado. Una estructura más o menos general de una función en Python es así:

```
def funcion(parámetros):
    """documentación"""
    instrucciones
    return resultado
```

1.1 Creación de una función

Podemos crear una función de Python utilizando la sentencia `def`, seguido de un nombre que identifique la función y un par de paréntesis `()` rodeando los parámetros que pudiera tener (a continuación definiremos qué es un parámetro, estos son opcionales, una función se puede definir perfectamente sin recurrir a estos) y finalmente, puesto que la función también representa un bloque de código, agregamos al final dos puntos `:`. Después de crear una función podemos invocarla utilizando el nombre de la función seguido de paréntesis `()` que, dado el caso, contendrá los parámetros de esa función en particular. Es importante resaltar que toda función se debe definir antes de llamarla, de lo contrario Python mostrará un mensaje de error.

[ ]:
```
%%debug_cell_with_pytutor
def saludar():
    print('hola mundo')

saludar()
```

1.2 Argumentos y parámetros de una función

Los argumentos son los valores que se pasan dentro del paréntesis de la función. Estos argumentos se corresponden uno a uno con los parámetros de la función, que son las variables que usará la función para capturar los valores del usuario y luego operar para obtener el resultado requerido. Una función puede tener cualquier número de parámetros separados por una coma.

[ ]:
```
#función para definir si un número es par o impar

def paridad(x): #recibimos el número del usuario en x
    if x%2 == 0: #evaluamos la paridad de x con el módulo
        print(f'{x} es par')
    else:
        print(f'{x} es impar')
```

[ ]:
```
paridad(2) #aquí x=2
paridad(5) #aquí x=5
paridad(84) #aquí x=84
paridad(33) #aquí x=33
```

```
2 es par
5 es impar
84 es par
33 es impar
```

###Tipos de parámetros Python soporta varios tipos de parámetros que pueden ser pasados en el momento de la llamada a la función.

Parámetros por defecto Es un parámetro que asume un valor por defecto si no se proporciona un valor en la llamada a la función. El siguiente ejemplo ilustra los parámetros por defecto.

[ ]:
```
%%debug_cell_with_pytutor
#función para multiplicar números

def multiplicar(x,y=2):
    print(f'x = {x}, y = {y}')
    print('x*y=',x*y)


multiplicar(5)
multiplicar(5,5)
```

Como vimos, si no se le entrega un argumento al parámetro que tiene un valor por defecto, la función tomará este valor, de lo contrario, usará el valor que le entregamos. Podemos poner la cantidad de parámetros con valor por defecto que queramos, esto estará más ceñido al diseño de la aplicación que estemos implementando.

1.3 Parámetros con palabras clave (keyword)

La idea es permitir que la persona que llama la función especifique el nombre del argumento con valores a ingresar, esto es útil, ya que nos ahorra el tener que recordar el orden de los parámetros, si el usuario ingresa en desorden los argumentos, pero los pasa por palabra clave, el programa usará correctamente cada valor.

[ ]:
```
# Programa para demostrar el paso por palabras clave.
def estudiante(nombre, apellido):
    print(nombre, apellido)


# argumentos por palabra clave
estudiante(nombre='Pepe', apellido='Mujica')
estudiante(apellido='Tapias', nombre='Armando')
```

```
Pepe Mujica
Armando Tapias
```

Pero ¿cómo podemos garantizar que el usuario sepa qué valores debe ingresar como argumentos en la función?

1.4 Docstring

La primera cadena después de la función se denomina cadena de documento o Docstring en forma abreviada. Se utiliza para describir la funcionalidad de la función. El uso del docstring en las funciones es **opcional** pero se considera una **buena práctica**. Este string suele ir siempre rodeado de 3 comillas, sean sencillas ' o dobles ", ya que esto permite realizar la documentación en varias líneas, mejorando su legibilidad. Esta documentación debería ser breve, describir los parámetros, explicando el tipo de dato que se espera recibir y también cuál será la salida de nuestra función.

La siguiente sintaxis se puede utilizar para imprimir el docstring de una función:

```
print(nombre_funcion.__doc__)
```

Regresemos al ejemplo de los números pares, añadamos un docstring e imprimámoslo.

[ ]:
```
#función para definir si un número es par o impar

def paridad(x): #recibimos el número del usuario en x
    """Función para comprobar si el número es par o impar.
    Entrada:
        x -> int: número a verificar
    Salida
        None. En la función se imprime el resultado
    """
    if x%2 == 0: #evaluamos la paridad de x con el módulo
        print(f'{x} es par')
    else:
        print(f'{x} es impar')
```

[ ]:
```
print(paridad.__doc__)
```

```
Función para comprobar si el número es par o impar.
    Entrada:
        x -> int: número a verificar
    Salida
        None. En la función se imprime el resultado
```

1.5 La sentencia `return`

La sentencia `return` de una función se utiliza para salir de la misma, volver a la línea donde se llama la función y devolverle el valor o dato especificado.

```
return salida1,salida2,salida3
```

La sentencia return puede consistir en una variable, una expresión o una tupla (como habíamos mencionado antes) que se retorna al final de la ejecución de la función. Si no se añade ninguna de estas a la sentencia return, ésta devuelve por defecto un objeto `None`.

La palabra reservada `None` se utiliza para definir un valor nulo, o literalmente "ningún valor".

`None` no es lo mismo que `0`, `False`, o una cadena vacía. `None` es un tipo de dato propio (`NoneType`) y sólo `None` puede ser igual a `None`.

[ ]:
```
# Creemon una función que calcule potencias o el cuadrado de un número por␣
 ↪defecto y lo retorne
def potencias(a,b=2):
    """
    Función para calcular la potencia de un número (a^b)
    Entradas:
```

```
        a -> int o float: base de la potencia
        b -> int o float: exponente, valor por defecto: 2
    Salidas:
        p -> int o float: a**b
    """
    return a**b

print(potencias(2))
print(potencias(3,5))
x = potencias(2,8) # podemos almacenar los resultados en variables
print('x:',x)
```

```
4
243
x: 256
```

[ ]:
```
# función para separar nombre y apellido a partir de un solo string separado␣
↪por espacios

def separador(texto):
    """
    Función para separar nombre y apellido a partir de un solo string separado␣
↪por espacios
    Entrada:
        texto -> string: cadena con nombre y apellido
    Salida:
        nombre,apellido -> string: nombre y apellido separados
    """
    aux = texto.split(" ") #la función split permite separar en una lista␣
↪strings
                           #según el argumento que se le pase.
    nombre = aux[0]        #esperamos solo un nombre y apellido, el nombre␣
↪estará
    apellido = aux[1]      #en la primera posición de aux, el apellido en la␣
↪segunda

    return nombre,apellido
```

[ ]:
```
print(separador("Fulanito DeTal"))
```

```
('Fulanito', 'DeTal')
```

[ ]:
```
print(type(separador("Fulanito DeTal")))
```

```
<class 'tuple'>
```

```
[ ]: nombre,apellido = separador("Fulanito DeTal") #podemos guardar las salidas en␣
     ↪distintas variables
                                                    #separando cada una con comas

     print(nombre)
     print(apellido)
```

```
Fulanito
DeTal
```

1.6 Alcance y vida útil de las variables

El alcance (***scope***) de una variable es la parte de un programa donde se reconoce la variable. Los parámetros y variables definidos dentro de una función no son visibles desde fuera de la misma. Por lo tanto, tienen un alcance local.

El tiempo de vida de una variable es el periodo durante el cual la variable existe en la memoria. El tiempo de vida de las variables dentro de una función es tan largo como la función que se ejecuta. Se destruyen una vez que regresamos de la función. Por lo tanto, una función no recuerda el valor de una variable de sus llamadas anteriores.

Este es un ejemplo para ilustrar el alcance de una variable dentro de una función.

```
[ ]: def func():
         x = 10
         print("Valor dentro de la función:",x)

     x = 20
     func()
     print("Valor fuera de la función",x)
```

```
Valor dentro de la función: 10
Valor fuera de la función 20
```

Veamos este manejo de memoria dentro de Python Tutor:

```
[ ]: %%debug_cell_with_pytutor
     def func():
         x = 10
         print("Valor dentro de la función:",x)

     x = 20
     func()
     print("Valor fuera de la función",x)
```

Aquí, podemos ver que el valor de x es 20 inicialmente. Aunque la función `func()` cambió el valor de x a 10, no afectó al valor fuera de la función.

Esto se debe a que la variable x dentro de la función es diferente (local a la función) de la que está fuera. Aunque tengan los mismos nombres, son dos variables diferentes con alcances distintos.

Por otro lado, las variables fuera de la función son visibles desde dentro. Tienen un ámbito global.

Podemos leer estos valores desde dentro de la función, pero no podemos modificarlos (sobrescribirlos). Para modificar el valor de las variables definidas fuera de la función, estas se deben definir como variables globales escribiendo la palabra reservada `global` antes del nombre de la variable, sin embargo, esto suele ser una muy mala práctica.