2-Uso_avanzado_de_funciones

March 13, 2020

1 Programación para la ciencia de datos

1.1 Unidad 2: Uso avanzado de funciones en Python

1.1.1 Instrucciones de uso

Este documento es un notebook interactivo que intercala explicaciones más bien teóricas de conceptos de programación con fragmentos de código ejecutables. Para aprovechar las ventajas que aporta este formato, se recomienda, en primer lugar, leer las explicaciones y el código que os proporcionamos. De esta manera tendréis un primer contacto con los conceptos que se exponen. Ahora bien, **¡la lectura es solo el principio!** Una vez que hayáis leído el contenido proporcionado, no olvidéis ejecutar el código proporcionado y modificarlo para crear variantes, que os permitan comprobar que habéis entendido su funcionalidad y explorar los detalles de la implementación. Por último, se recomienda también consultar la documentación enlazada para explorar con más profundidad las funcionalidades de los módulos presentados.

[1]:
```
%load_ext pycodestyle_magic
```

[2]:
```
# Activamos las alertas de estilo
%pycodestyle_on
```

1.1.2 Introducción

En esta unidad se repasan muy rápidamente los conceptos básicos de funciones y se exponen algunos conceptos más avanzados sobre el uso de funciones en Python.

En primer lugar, se explica qué es el ámbito (el *scope*) de una variable, un concepto que seguro conocéis de manera informal, pero que no se ha trabajado formalmente todavía.

En segundo lugar, se repasa como devolver valores desde funciones, se comentan los posibles valores que puede devolver una función y se presentant algunas situaciones que pueden parecer especiales, tales como las funciones que devuelven funciones.

A continuación, se expone cómo funcionan los parámetros de las funciones en Python. Se habla de cómo se pasan estos parámetros y qué pasa cuando los modificamos, de la definición de funciones con argumentos opcionales y con un número de argumentos indeterminado, y también de cómo pasar funciones como argumentos de otras funciones.

Seguidamente, se hace una pequeña introducción a la programación funcional en Python, presentando las tres funciones básicas de este paradigma de programación (`map`, `filter` y `reduce`).

Después, se presentan las funciones anónimas, y se ponen ejemplos de uso de este tipo de funciones cuando se utiliza `map`, `filter` y `reduce`.

Finalmente, se explica que es el *docstring*, como podemos consultarlo y qué convenciones se utilizan a la hora de definir el *docstring* de las funciones.

A continuación se incluye la tabla de contenidos, que podéis utilizar para navegar por el documento:

---

2 1.- Funciones

Como ya sabemos, una función es una manera de encapsular código, que nos permite reaprovechar-lo para diversas tareas. Una función es un fragmento de código que tiene un nombre y que realiza una tarea específica.

En Python definimos una función con la palabra reservada `def`, seguida del nombre que damos a la función y los parámetros que recibe entre paréntesis. Una función puede devolver un valor, que se indica con la palabra `return`.

[3]:
```
# Definimos la función de nombre 'sum_two_values' con dos argumentos:
# 'x' y 'y':
def sum_two_values(x, y):
    """Return the value of the sum."""
    return x + y
```

Para ejecutar una función, la llamamos utilizando su nombre, y especificando los argumentos:

[4]:
```
r = sum_two_values(3, 5)
print(r)
```

```
8
```

2.1 1.1.- Ámbito de visibilidad

Como ya hemos visto, la asignación de una variable en cualquier celda de un notebook permite utilizar esta variable en todo el notebook. Así pues, por ejemplo, podemos mostrar el contenido de la variable `r` asignada en la celda anterior en la siguiente celda:

[5]:
```
print(r)
```

```
8
```

Si intentamos utilizar una variable que no hayamos asignado anteriormente, Python lanza la excepción `NameError`:

[6]:
```
try:
    print(undef_var)
except NameError as e:
    print(e)
```

```
name 'undef_var' is not defined
```

Este error aparece ya que la variable `undef_var` no se ha asignado con anterioridad a la ejecución del `print` sobre la variable.

Recordad que las diferentes celdas de una notebook pueden ejecutarse de manera independiente, y que el orden de ejecución de las celdas (y no el orden en el que se encuentran dentro del notebook) determinará el flujo de ejecución. Cuando programamos utilizando notebooks **es muy recomendable que estos se puedan ejecutar linealmente sin errores**, es decir, que el resultado esperado del notebook sea el que se obtiene al pulsar el botón del Menú `Kernel`,`Restart and Run All`.

A modo de ejemplo y para que podáis comprobar la posible no linealidad de ejecución de un notebook, en la celda siguiente definiremos la variable `undef_var` y os pedimos que ejecutéis de nuevo la celda anterior (donde se hacía un `print` de esta variable). A diferencia de la primera vez que ejecutamos la celda, esta vez no debería dar ningún error, ya que ya habremos definido la variable. Este tipo de acciones, sin embargo, son las que hay que evitar, ya que dificultan la lectura, claridad y portabilidad del código que implementamos.

[7]:
```
# Definimos la variable undef_var
undef_var = "Now it has a value"

#################################################################
# IMPORTANTE: ejecute ahora la celda de código anterior, que contiene
# El print de la variable undef_var, para comprobar que la ejecución
# Ahora no genera ningún error.
#################################################################
```

Decimos que una variable es **global** cuando ésta se asigna fuera de una función (como en el caso tanto de la variable `r` como de la variable`undef_var`). Por el contrario, diremos que una variable es **local** cuando la definimos dentro de una función. Para referirnos al área desde donde puede utilizarse una determinada variable utilizamos la palabra **ámbito** (en inglés, hablamos de *scope*).

Una variable local no puede ser utilizada fuera del cuerpo de la función donde está definida. En cambio, una variable global puede ser utilizada tanto desde fuera de cualquier función como desde el cuerpo de las funciones:

[8]:
```
# Definimos una variable global
global_var = "This is a global variable"


def fun_1():
    # Definimos una variable local dentro de fun_1
    local_var_1 = "local_var_1 is local to fun_1"
    # Mostramos el valor de la variable global y de la local
    print(global_var)
    print(local_var_1)


def fun_2():
    # Mostramos la variable global
    print(global_var)
```

```
    # Intentamos acceder a la variable local local_var_1,
    # cosa que generará un error ya que no está definida
    # dentro de la función fun_2
    print(local_var_1)
```

[9]:
```
# Mostramos la variable global
print(global_var)
```

```
This is a global variable
```

[10]:
```
# Comprobamos como no podemos acceder a la variable local local_var_1
# ya que ésta está definida dentro del ámbito de la función fun_1
try:
    print(local_var_1)
except NameError:
    print("Error: Variable no definida")
```

```
Error: Variable no definida
```

[11]:
```
# Ejecutamos fun_1, que muestra correctamente el valor de la variable
# global y la local
fun_1()
```

```
This is a global variable
local_var_1 is local to fun_1
```

[12]:
```
# Ejecutamos fun_2, que muestra correctamente el valor de la variable
# global pero lanza una excepción al acceder a la variable local
# ya que ésta está definida en fun_1
try:
    fun_2()
except NameError:
    print("Error: Variable no definida")
```

```
This is a global variable
Error: Variable no definida
```

Hasta ahora hemos visto el comportamiento de las variables globales y locales dentro y fuera de funciones. £Qué pasa, sin embargo, si definimos una variable local (dentro de una función) que tiene el mismo nombre que una variable global definida fuera de la función?

[13]:
```
def fun_3():
    # Asignamos la variable global_var
    global_var = "Now this is a local var!"
    # Mostramos el valor de global_var
    print(global_var)
```

```
# Mostramos el valor de global_var antes de ejecutar la función fun_3
print(global_var)
# Ejecutamos la función fun_3
fun_3()
# Mostramos el valor de global_var después de ejecutar la función fun_3
print(global_var)
```

```
This is a global variable
Now this is a local var!
This is a global variable
```

Por un lado, tened en cuenta que el valor de la variable global `global_var` no cambia con la ejecución de la función: el `print` de antes de la llamada a `fun_3` y el `print` de después de la llamada devuelven exactamente el mismo valor. Por otra parte, tened en cuenta que dentro de la función `fun_3`, la variable `global_var` toma un nuevo valor. En realidad, lo que hemos hecho al asignar `global_var` dentro de la función es crear una nueva variable con el mismo nombre, dentro del ámbito de la función. Este comportamiento difiere de lo que pasaba al llamar las funciones `fun_1` y `fun_2`, donde el nombre `global_var` hacía referencia a la variable global. La diferencia se debe a que en las funciones 1 y 2 simplemente se mostraba el contenido de la variable, mientras que en la función 3 la variable se asigna.

Así pues, £como podemos reasignar el valor de una variable global desde dentro de una función? Para ello, habrá que hacer uso de la palabra reservada `global`, que permitirá indicar, en el contexto de una función, que queremos utilizar una variable global:

[14]:
```
def fun_4():
    # Identificamos la variable global_var como global
    global global_var
    # Asignamos un nuevo valor a la variable global global_var
    global_var = "Changing the value of the global var"
    # Mostramos el valor de global_var
    print(global_var)


# Mostramos el valor de global_var antes de ejecutar la función fun_4
print(global_var)
# Ejecutamos la función fun_4
fun_4()
# Mostramos el valor de global_var después de ejecutar la función fun_4
print(global_var)
```

```
This is a global variable
Changing the value of the global var
Changing the value of the global var
```

Notad cómo, al llamar a `fun_4`, se modifica el valor de la variable global `global_var` (el valor de la variable en el último print es el valor asignado dentro de la función).

2.1.1 1.1.1.- Funciones dentro de funciones

Python permite definir funciones dentro de otras funciones. En esta situación, el comportamiento de las variables definidas en estas funciones es similar al que hemos descrito anteriormente.

En el ejemplo siguiente hay una variable global `y` y una función `outer_fun` definida también en el ámbito global. Dentro de la función `outer_fun`, se define una variable local `x` con valor `3`, y cuatro funciones internas (`inner_fun_i` con `i` de 1 a 4). Cada una de las cuatro funciones internas muestra los posibles comportamientos con relación a la variable `x` definida en la función externa: * `inner_fun_1` muestra el valor de la variable `x` definida en `outer_fun`. * `inner_fun_2` asigna a una variable `x` el valor 5. Esto crea una nueva variable llamada `x`, local a `inner_fun_2`. Por lo tanto, el valor de la variable `x` de `outer_fun` no se modifica. * `inner_fun_3` utiliza la palabra clave `nonlocal` para indicar que quiere acceder a la variable `x` del ámbito de `outer_fun`, y reasigna el valor de `x` a 7. Por lo tanto, cuando se vuelve a mostrar el valor de `x` desde la función `outer_fun`, el valor ha sido modificado. El `nonlocal` tiene una funcionalidad similar al `global` que habíamos visto en la función `fun_4`, pero sirve para referirse a una variable definida en un ámbito superior no global. * `inner_fun_4` intenta acceder a la variable `my_var_if1`, que es local a la función `inner_fun_1`, generando, en consecuencia, una excepción.

[15]:
```
# Definimos la función outer_fun al ámbito global
def outer_fun():

    # Definimos la función inner_fun_1 dentro de outer_fun
    def inner_fun_1():
        # Definimos la variable my_var_if1 dentro inner_fun_1
        my_var_if1 = "This is defined in inner_fun_1"
        # Mostramos el valor de la variable de outer_fun x y la variable global␣
↪y
        print("Inner fun 1 x:\t{}".format(x))
        print("Inner fun 1 y:\t{}".format(y))

    # Definimos la función inner_fun_2 dentro de outer_fun
    def inner_fun_2():
        # Definimos y mostramos una nueva variable x local a inner_fun_2 con␣
↪valor 5
        x = 5
        print("Inner fun 2 x:\t{}".format(x))

    # Definimos la función inner_fun_3 dentro de outer_fun
    def inner_fun_3():
        # Indicamos que utilizaremos la variable x no local
        # (definida en outer_fun)
        nonlocal x
        # Modificamos y mostramos el valor de x
        x = 7
        print("Inner fun 3 x:\t{}".format(x))

    # Definimos la función inner_fun_4 dentro de outer_fun
    def inner_fun_4():
```

```
        try:
            # Intentamos acceder a la variable my_var_if1 definida dentro
            # de inner_fun_1 (lo que generará una excepción)
            print(my_var_if1)
        except NameError:
            print("Error: undefined variable")

    # Mostramos el valor de la variable global y
    print("Outer fun 1 y:\t{}".format(y))

    # Definimos una variable local a outer_fun de nombre x y valor 3
    x = 3
    # Vamos mostrando el valor de x y ejecutando las funciones internas, para␣
 ↪ver
    # el efecto que tienen sobre x
    print("Outer fun x:\t{}".format(x))
    inner_fun_1()
    print("Outer fun x:\t{}".format(x))
    inner_fun_2()
    print("Outer fun x:\t{}".format(x))
    inner_fun_3()
    print("Outer fun x:\t{}".format(x))
    inner_fun_4()


# Definimos una variable global y
y = 1
# Llamamos la función outer_fun
outer_fun()
```

```
10:80: E501 line too long (81 > 79 characters)
16:80: E501 line too long (84 > 79 characters)
43:80: E501 line too long (81 > 79 characters)

Outer fun 1 y:  1
Outer fun x:    3
Inner fun 1 x:  3
Inner fun 1 y:  1
Outer fun x:    3
Inner fun 2 x:  5
Outer fun x:    3
Inner fun 3 x:  7
Outer fun x:    7
Error: undefined variable
```

Definir funciones dentro de funciones permite **encapsular** código, es decir, ocultar ciertas partes del código, de forma que sólo puedan ser llamadas desde ciertas funciones. Siguiendo

con el ejemplo anterior, las funciones `inner_fun_i` sólo son visibles desde dentro de`outer_fun`, y no pueden ser llamadas desde el ámbito global:

[16]:
```
try:
    # Intentamos ejecutar una de las funciones internas, lo que generará
    # una excepción ya que no están definidas en el ámbito global
    inner_fun_1()
except NameError:
    print("Error: undefined 'inner_fun_1'")
```

```
Error: undefined 'inner_fun_1'
```

£Por qué podemos querer encapsular el código? Supongamos, por ejemplo, que estamos analizando un conjunto de datos con las localizaciones actuales de un conjunto de personas, así como la localización del domicilio y del lugar de trabajo de éstas, y que queremos saber si estas personas se encuentran cerca tanto del puesto de trabajo como del domicilio. Para ello, implementamos una función `is_close` que nos devuelve un booleano que indica si están cerca o no de ambas localizaciones. Esta función necesitará calcular la distancia entre dos puntos dos veces (una para calcular la distancia entre la localización actual y el domicilio, y una segunda vez para saber la distancia entre la localización actual y el lugar de trabajo). Por lo tanto, para no repetir código, definiremos otra función `dist` que calcule la distancia entre dos puntos. Ahora bien, para el análisis que queremos hacer, no utilizaremos la distancia euclídea, sino que utilizaremos la distancia de Manhattan. En el resto de nuestro código, pero, nunca utilizaremos esta definición de distancia, y queremos evitar que algún otro programador del equipo, por error, llame a nuestra función distancia `dist` (pensando, quizás, que calcula la distancia más habitual, la Euclidiana). Una posible manera de hacerlo es definir la función `dist` como una función local a `is_close`. De este modo, podemos utilizar una función y no tendremos que repetir código al calcular las distancias y, al mismo tiempo, evitaremos que se llame esta función desde fuera de la función `is_close`:

[17]:
```
def is_close(x, y, l1_x, l1_y, l2_x, l2_y):
    """Computes whether a location is nearby two other locations.

    Computes if location (x,y) is close to two other locations,
    (l1_x, l1_y) and (l2_x, l2_y)."""

    def dist(x1, y1, x2, y2):
        """Computes the Manhattan distance between (x1, y1) and (x2, y2)."""
        return abs(x1 - x2) + abs(y1 - y2)

    # Consideramos que un punto está cerca de otro si la distancia entre ellos
    # es inferior a lim
    lim = 10
    # Retornamos si (x, y) está cerca tanto de (l1_x, l1_y) como de (l2_x,␣
 ↪l2_y)
    return (dist(x, y, l1_x, l1_y) < lim) and (dist(x, y, l2_x, l2_y) < lim)
```

```
14:80: E501 line too long (80 > 79 characters)
```

[18]:
```
# Calculamos si la dirección actual (0, 0) está cerca de (1, 4) y (-7, 1)
r1 = is_close(0, 0, 1, 4, -7, 1)
print("(0, 0) is close to (1, 4) and (-7, 1)?: {}".format(r1))

# Calculamos si la dirección actual (0, 0) está cerca de (1, 4) y (15, 1)
r2 = is_close(0, 0, 1, 4, 15, 1)
print("(0, 0) is close to (1, 4) and (15, 1)?: {}".format(r2))
```

```
(0, 0) is close to (1, 4) and (-7, 1)?: True
(0, 0) is close to (1, 4) and (15, 1)?: False
```

Antes de terminar esta sección, hacemos un apunte sobre la implementación de la función `is_close`. Fijaos que la función debe devolver `True` si se cumple una determinada condición (si el usuario se encuentra cerca del trabajo y de su vivienda) y `False` de lo contrario. Una posible manera de implementar este comportamiento es con una instrucción `if`:

```
if (dist(x, y, l1_x, l1_y) < lim) and (dist (x, y, l2_x, l2_y) < lim):
    return True
else:
    return False
```

Este código utiliza una expresión (que incluye el cálculo de distancias) para diferenciar si la función debe devolver `True` o `False`. Es importante notar que la propia expresión ya devuelve un booleano y que, por lo tanto, puede utilizarse directamente como valor de retorno:

```
return (dist(x, y, l1_x, l1_y) < lim) and (dist(x, y, l2_x, l2_y) < lim)
```

de modo que obtenemos un código más conciso, que se comporta exactamente igual, y que evita redundancia.

2.2 1.2.- Valor de retorno

Una función puede devolver un valor, que se indica con la palabra `return`. Si la función no tiene `return` o bien tiene un `return` vacío, la ejecución de la función devolverá el valor `None`:

[19]:
```
# Definimos una función sin return
def print_sum_v0(x, y):
    print("Result is: {}".format(x+y))


# Definimos una función con retorno vacío
def print_sum_v1(x, y):
    print("Result is: {}".format(x+y))
    return


# El valor de retorno de las dos funciones es None
r = print_sum_v0(3, 5)
print(r)
```

```
r = print_sum_v1(3, 5)
print(r)
```

```
Result is: 8
None
Result is: 8
None
```

Una función devuelve un único **objeto**. Así, si es necesario que una función devuelva más de un valor, podemos hacer que la función devuelva una tupla con los diversos valores.

[20]:
```
# Definimos la función de nombre 'sum_two_values' con dos argumentos: 'x'
# e 'y':
def sum_two_values(x, y):
    """Return the value of the sum."""
    return x + y


# Definimos la función de nombre 'sum_and_mult_two_values' con dos argumentos:
# 'x' e 'y':
def sum_and_mult_two_values(x, y):
    """Return a tuple with the value of the sum and the product."""
    return x + y, x * y


r1 = sum_two_values(5, 11)
r2 = sum_and_mult_two_values(5, 11)
print("The result of sum_two_values(5, 11) is {} ({})".
      format(r1, type(r1)))
print("The result of sum_and_mult_two_values(5, 11) is {} ({})".
      format(r2, type(r2)))
```

```
The result of sum_two_values(5, 11) is 16 (<class 'int'>)
The result of sum_and_mult_two_values(5, 11) is (16, 55) (<class 'tuple'>)
```

Tened en cuenta que al retornar la tupla, se está utilizando la sintaxis de definición de tupla sin paréntesis, que es más rápida de escribir ya que contiene menos caracteres, pero puede llevar a confusión al programador principiante. Así pues, notad que el return de la segunda función sería equivalente a:

```
return (x + y, x * y)
```

2.2.1 1.2.1.- Funciones como valores de retorno

Del mismo modo que una función puede devolver, por ejemplo, un entero, una cadena de caracteres o una tupla, una función puede devolver también otra función. Esto nos será útil en ciertas construcciones que veremos en este módulo.

Por ejemplo, la función `min_or_max` definida en la siguiente celda devuelve o bien la función `min` o bien la función `max`, dependiendo de si el valor que recibe como parámetro, `sel`, es par o impar:

[21]:
```
def min_or_max(sel):
    """Return either the min or the max function."""
    if sel % 2:
        # sel es impar
        f = max
    else:
        # sel es par
        f = min
    return f
```

[22]:
```
# Definimos una lista
a_list = [1, 1, 2, 3, 5, 8, 13, 21]
# Llamamos a min_or_max con el último elemento de la lista como parámetro
f = min_or_max(a_list[-1])
# Como el último elemento de la lista es impar, f contendrá la función max
print("The type of f is: {}".format(type(f)))
print("f is: {}".format(f))
# Aplicamos la función f (es decir, max) a la lista a_list
r = f(a_list)
print("The result of applying f to a_list is: {}".format(r))
```

```
The type of f is: <class 'builtin_function_or_method'>
f is: <built-in function max>
The result of applying f to a_list is: 21
```

Combinando la funcionalidad de definir funciones dentro de funciones y la de devolver funciones, se pueden crear construcciones complejas que permiten resolver elegantemente algunos problemas. Si estáis interesados en este tipo de construcciones, podéis consultar el enlace(lectura opcional, para ampliar conocimientos).

2.3 1.3.- Parámetros

Las funciones pueden tener parámetros, que sirven para proveerlas de datos que necesitan para su ejecución. Formalmente, distinguimos entre parámetros y argumentos. Los parámetros son parte de la firma de la función, mientras que los argumentos son los valores que recibe la función en el momento de ejecutarla. Por ejemplo, en la celda de código donde definíamos la función `sum_and_mult_two_values`, `x` e `y` son los parámetros de la función `sum_and_mult_two_values`, y cuando hacemos la llamada `sum_and_mult_two_values(5, 11)`, `5` y `11` son los argumentos.

Informalmente, sin embargo, a menudo se utilizan ambos términos indistintamente.

2.3.1 1.3.1.- Paso por referencia de objeto

Hasta ahora hemos visto como las funciones reciben unos argumentos y devuelven unos valores. £Qué pasa, sin embargo, cuando modificamos los valores de las variables que recibimos como argumentos en una llamada de una función? ąObservémoslo con algunos ejemplos!

[23]:
```
# Definimos una función que recibe un parámetro 'i' y
# asigna el valor 5 a este
def reassign_num(i):
    i = 5
    print("Value of i in the function:\t{}".format(i))


# Definimos una función que recibe un parámetro 'li' y
# asigna el valor [1, 2] a este
def reassign_list(li):
    li = [1, 2]
    print("Value of li in the function:\t{}".format(li))


# Definimos una función que recibe un parámetro 'li' y
# le añade un entero 1 con append
def append_val(li):
    li.append(1)
    print("Value of li in the function:\t{}".format(li))
```

Hay varios detalles a tener en cuenta en la definición de estas tres funciones. En primer lugar, las funciones no tienen ninguna instrucción `return`, por lo que su llamada devolverá siempre `None`. En segundo lugar, las dos primeras funciones (`reassign_num` y `reassign_list`) pueden recibir un valor de cualquier tipo como parámetro. En cambio, la tercera función (`append_val`) debe recibir un parámetro de un tipo que implemente el método `append`, como una lista. Por último, es interesante notar qué hacen estas funciones: las dos primeras (`reassign_num` y `reassign_list`) **asignan** un nuevo valor a la variable que reciben como parámetro (un entero en el caso de `reassign_num`, y una lista para `reassign_list`). Contrariamente, la tercera función (`append_val`) lo que hace es **modificar** la variable que recibe como parámetro, en este caso, añadiendo un elemento a la lista.

Fijémonos, a continuación, con el efecto que tiene la ejecución de estas funciones sobre las variables que se reciben.

[24]:
```
an_integer = 42
print("Value of an_integer:\t\t{}".format(an_integer))
ret_val = reassign_num(an_integer)
print("Function returns:\t\t{}".format(ret_val))
print("Value of an_integer:\t\t{}".format(an_integer))
```

```
Value of an_integer:            42
Value of i in the function:     5
Function returns:               None
Value of an_integer:            42
```

[25]:
```
a_list = [42]
print("Value of a_list:\t\t{}".format(a_list))
ret_val = reassign_list(a_list)
print("Function returns:\t\t{}".format(ret_val))
```

```
print("Value of a_list:\t\t{}".format(a_list))
```

```
Value of a_list:                [42]
Value of li in the function:    [1, 2]
Function returns:               None
Value of a_list:                [42]
```

En estos dos primeros casos, la variable que se pasa como parámetro en las llamadas a las funciones `reassign_num` y `reassign_list` no se ve afectada por la reasignación que se realiza dentro de las funciones: después de ejecutar `reassign_num`, la variable `an_integer` sigue conteniendo el valor42; y después de ejecutar la función `reassign_list`, la variable `a_list` sigue valiendo `[42]`. Fijémonos ahora en el comportamiento de la variable `a_list` cuando llamamos la función `append_val`:

[26]:
```
a_list = [42]
print("Value of a_list:\t\t{}".format(a_list))
ret_val = append_val(a_list)
print("Function returns:\t\t{}".format(ret_val))
print("Value of a_list:\t\t{}".format(a_list))
```

```
Value of a_list:                [42]
Value of li in the function:    [42, 1]
Function returns:               None
Value of a_list:                [42, 1]
```

En este caso, y a diferencia de las dos llamadas anteriores, el valor de la variable `a_list` se modifica (aunque la función no ha devuelto ningún valor). Esto sucede por la implementación interna que hace Python del paso de parámetros en las llamadas a las funciones. Este comportamiento se conoce como paso por referencia de objeto (en inglés, *pass-by-object-reference*), y no es igual en todos los lenguajes de programación. El lector interesado puede consultar este post para profundizar en el funcionamiento del paso de parámetros en Python (lectura opcional, para ampliar conocimientos). Es importante tener en cuenta este comportamiento de las funciones en Python a la hora de programar ya que, en caso de no hacerlo, podemos obtener resultados inesperados en las llamadas a funciones.

2.3.2 1.3.2.- Argumentos opcionales

La manera más directa de especificar argumentos opcionales en la definición de una función es darles un valor predeterminado. Entonces, si en el momento de hacer la llamada el argumento no se especifica, este tomará el valor por defecto que se haya especificado en la definición de la función:

[27]:
```
# Definimos una función con dos parámetros obligatorios y uno
# de opcional. El parámetro opcional z será 0 si no se especifica.
def sum_two_or_three_values(x, y, z=0):
    print("Values are: x={}, y={}, z={}".format(x, y, z))
    return x + y + z
```

[28]:
```
# Llamamos a la función con sólo los parámetros obligatorios, x e y
print("(1, 3): {}".format(sum_two_or_three_values(1, 3)))
```

```
# Llamamos a la función con los parámetros obligatorios y el opcional
print("(1, 3, 4): {}".format(sum_two_or_three_values(1, 3, 4)))
```

```
Values are: x=1, y=3, z=0
(1, 3): 4
Values are: x=1, y=3, z=4
(1, 3, 4): 8
```

[29]:
```
# Si llamamos a la función con 1 único parámetro se genera un error
# ya que hay dos parámetros obligatorios
try:
    print("(1): {}".format(sum_two_or_three_values(1)))
except TypeError as e:
    print("TypeError:", e)
```

```
TypeError: sum_two_or_three_values() missing 1 required positional argument: 'y'
```

Las llamadas anteriores especifican los parámetros por **posición**. En este caso, diremos que utilizamos argumentos posicionales. También podemos hacer llamadas a las funciones especificando los argumentos por su **nombre** (en inglés, los llamaremos *keyword arguments*), independientemente de si éstos son obligatorios u opcionales:

[30]:
```
# Especificamos todos los parámetros por nombre
print("(x=1, y=3, z=4): {}".format(
    sum_two_or_three_values(x=1, y=3, z=4)))

# Especificamos dos parámetros por posición y uno por nombre
print("(1, 3, z=4): {}".format(
    sum_two_or_three_values(1, 3, z=4)))

# Especificamos todos los parámetros por nombre, cambiando el orden
# los parámetros
print("(y=3, z=4, x=1): {}".format(
    sum_two_or_three_values(y=3, z=4, x=1)))
```

```
Values are: x=1, y=3, z=4
(x=1, y=3, z=4): 8
Values are: x=1, y=3, z=4
(1, 3, z=4): 8
Values are: x=1, y=3, z=4
(y=3, z=4, x=1): 8
```

Notad cómo, cuando especificamos los parámetros por su nombre, podemos incluirlos en el orden que queramos. En cambio, si especificamos los parámetros por posición, la posición del parámetro determinará su asignación.

2.3.3 1.3.3.- Número indeterminado de argumentos

Python también permite definir funciones que acepten un número arbitrario (indefinido en el momento de la definición de la función) de argumentos. En este caso, se define un parámetro especial anteponiendo * o ** al nombre del parámetro, y este parámetro recibirá todos los argumentos que no coincidan con ninguno de los parámetros definidos explícitamente de la función:

- Si se antepone * al nombre del parámetro, este será una tupla con todos los valores de los argumentos no definidos explícitamente.
- Si se antepone ** al nombre del parámetro, éste recibirá un diccionario con los pares de nombre y valor de los argumentos no definidos explícitamente (que se deberán especificar por nombre en el momento de hacer la llamada a la función).

[31]:
```
# Definimos una función con dos parámetros obligatorios y
# un número indeterminado de parámetros posicionales
def sum_two_or_three_values_p(x, y, *extra_arguments):
    print("Compulsory arguments are: x={}, y={}".format(x, y))
    print("Additional arguments are: {}".format(extra_arguments))
    print("extra_arguments type is: {}".format(type(extra_arguments)))
    return x + y + sum(extra_arguments)
```

[32]:
```
# Llamamos a la función sólo con los argumentos obligatorios
sum_two_or_three_values_p(1, 2)
```

```
Compulsory arguments are: x=1, y=2
Additional arguments are: ()
extra_arguments type is: <class 'tuple'>
```

[32]:
```
3
```

[33]:
```
# Llamamos a la función con 5 argumentos posicionales
sum_two_or_three_values_p(1, 2, 3, 4, 5)
```

```
Compulsory arguments are: x=1, y=2
Additional arguments are: (3, 4, 5)
extra_arguments type is: <class 'tuple'>
```

[33]:
```
15
```

[34]:
```
# Definimos una función con dos parámetros obligatorios y
# un número indeterminado de parámetros con nombre
def sum_two_or_three_values_n(x, y, **extra_arguments):
    print("Compulsory arguments are: x={}, y={}".format(x, y))
    print("Additional arguments are: ")
    for k in extra_arguments:
        print("\t{}={}".format(k, extra_arguments[k]))
    print("extra_arguments type is: {}".format(type(extra_arguments)))
    return x + y + sum(extra_arguments.values())
```

[35]:
```
# Llamamos la función sólo con los argumentos obligatorios
sum_two_or_three_values_n(1, 2)
```

```
Compulsory arguments are: x=1, y=2
Additional arguments are:
extra_arguments type is: <class 'dict'>
```

[35]: 3

[36]:
```
# Llamamos la función con los argumentos obligatorios y un argumento opcional
sum_two_or_three_values_n(1, 2, z=3)
```

```
Compulsory arguments are: x=1, y=2
Additional arguments are:
        z=3
extra_arguments type is: <class 'dict'>
```

[36]: 6

[37]:
```
# Llamamos la función con los argumentos obligatorios y tres argumentos
# opcionales
sum_two_or_three_values_n(1, 2, z=3, a=10, b=12)
```

```
Compulsory arguments are: x=1, y=2
Additional arguments are:
        z=3
        a=10
        b=12
extra_arguments type is: <class 'dict'>
```

[37]: 28

Es interesante notar que si hemos especificado que los argumentos opcionales serán posicionales (usando *), entonces no podemos a llamar la función asignando nombres a los argumentos. De manera análoga, si hemos especificado argumentos opcionales con nombre (usando **), no podremos llamar a la función con argumentos opcionales posicionales:

[38]:
```
try:
    # Especificamos un argumento opcional por nombre en una función
    # definida con *
    sum_two_or_three_values_p(1, 2, z=3)
except TypeError as e:
    print("TypeError:", e)
```

```
TypeError: sum_two_or_three_values_p() got an unexpected keyword argument 'z'
```

[39]:
```
try:
    # Especificamos un argumento opcional por posición en una función
```

```
    # definida con **
    sum_two_or_three_values_n(1, 2, 3)
except TypeError as e:
    print("TypeError:", e)
```

```
TypeError: sum_two_or_three_values_n() takes 2 positional arguments but 3 were
given
```

2.3.4 1.3.4.- Funciones como parámetros

Del mismo modo que una función puede recibir como parámetros, por ejemplo, un entero o una cadena de caracteres, una función puede recibir como parámetro otra función. Esto nos será útil en ciertas construcciones que veremos en este módulo.

Por ejemplo, la función `eval_and_print`, definida en la celda siguiente, recibe como parámetros una función y una lista, y lo que hace es evaluar la función sobre la lista, mostrar el resultado por pantalla, y devolverlo:

[40]:
```
# La función eval_and_print recibe otra función como parámetro (la función f)
def eval_and_print(f, x):
    # Mostramos f y el tipo de f
    print("f is {} and its type is {}".format(f, type(f)))
    # Aplicamos la función f sobre el valor x
    f_x = f(x)
    # Mostramos el valor por pantalla y lo retornamos
    print("The result of f(x) is: {}".format(f_x))
    return f_x


# Llamamos a eval_and_print con funciones diferentes
r1 = eval_and_print(max, [81, 75, 5, 10])
r2 = eval_and_print(min, [81, 75, 5, 10])
r3 = eval_and_print(sum, [81, 75, 5, 10])
```

```
f is <built-in function max> and its type is <class
'builtin_function_or_method'>
The result of f(x) is: 81
f is <built-in function min> and its type is <class
'builtin_function_or_method'>
The result of f(x) is: 5
f is <built-in function sum> and its type is <class
'builtin_function_or_method'>
The result of f(x) is: 171
```

2.4 1.4.- Introducción a la programación funcional

La programación funcional es un paradigma de programación basado en la evaluación de funciones matemáticas, en el que la salida de una función depende únicamente de su entrada, y no del estado del programa.

En esta sección, presentaremos tres funciones de Python que son útiles a la hora de programar con el paradigma de programación funcional: `map`, `filter` y `reduce`.

2.4.1 1.4.1.- Map

La función `map` recibe como parámetros una función y un iterable, y devuelve un iterador que aplica la función proporcionada a cada uno de los elementos del iterable.

[41]:
```
def sum_2(x):
    """Add 2 to the value received by parameter."""
    return x + 2


# Aplicamos la función sum_2 a cada uno de los elementos de la lista a_list
a_list = [1, 2, 3, 4]
r = map(sum_2, a_list)
print(list(r))
```

```
[3, 4, 5, 6]
```

El resultado de aplicar `map` sobre la lista es un iterable que contiene el resultado de llamar a la función `sum_2` sobre cada uno de los elementos de la lista original, `a_list`. Esto sería equivalente a ejecutar la *list comprehension* siguiente:

[42]:
```
[sum_2(x) for x in a_list]
```

[42]:
```
[3, 4, 5, 6]
```

[43]:
```
# Definimos una función que convierte cadenas de caracteres que expresan
# valores numéricos (posiblemente decimales) en enteros
def convert_to_int(x):
    """Convert strings to int."""
    return int(float(x))


# Aplicamos la función convert_to_int a cada uno de los elementos de la
# lista a_list
a_list = ["42.45", "13.4", "12000"]
list(map(convert_to_int, a_list))
```

[43]:
```
[42, 13, 12000]
```

En los ejemplos anteriores, la función que se pasa como argumento a `map` (`sum_2` y `convert_to_int`) tiene un único parámetro, por lo que `map` la puede aplicar directamente sobre cada uno de los elementos de la lista (lista que recibe como segundo argumento). Ahora bien, la función que se pasa como parámetro a `map` puede tener más de un parámetro. En este caso, la función `map` recibirá tantas listas como parámetros necesite la función que recibe.

En el ejemplo siguiente, definimos una función que calcula el precio de un piso a partir de los metros cuadrados que tiene (`sqm`), el estado de conservación (`status`) y el vecindario en que se encuentra (`neigh`):

[44]:
```python
def compute_price(sqm, status, neigh):
    """Compute the price of a flat."""

    price_per_sqm = 1000
    nice_neigh = ["A", "B"]
    nice_neigh_factor = 1.25
    new_factor = 2

    # Precio base es metros cuadrados por precio por metro cuadrado
    price = sqm * price_per_sqm

    # Si el piso se encuentra en un barrio considerado bueno, se aplica
    # un factor multiplicativo al precio del piso
    if neigh in nice_neigh:
        price *= nice_neigh_factor

    # Si el piso es nuevo, se aplica un factor multiplicativo al
    # precio del piso
    if status == "New":
        price *= new_factor

    return price
```

[45]:
```python
# Calculamos el precio de un piso nuevo de 100m^2 que se encuentra en un barrio
# catalogado como "A"
compute_price(100, "New", "A")
```

[45]:
```
250000.0
```

Una inmobiliaria tiene 5 pisos a su disposición, y quiere utilizar la función `compute_price` para calcular el precio que debería tener cada uno de los pisos. Podemos utilizar la función `map` para calcularlo, partiendo de tres listas que indiquen los metros cuadrados, los estados y los barrios de los pisos:

[46]:
```python
# Definimos tres listas con los datos de los pisos
sqms = [100, 120, 125, 190, 200]
statuses = ["New", "New", "Used", "Unknown", "Used"]
neigs = ["A", "B", "B", "D", "A"]

# Aplicamos la función compute_price sobre cada uno de los pisos
list(map(compute_price, sqms, statuses, neigs))
```

[46]:
```
[250000.0, 300000.0, 156250.0, 190000, 250000.0]
```

2.4.2 1.4.2.- Filter

La función `filter` recibe también como parámetros una función y un iterable, y devuelve un iterador que recorre los elementos del iterable tales que la evaluación de la función es True.

[47]:
```
# Definimos la función is_numeric, que devuelve True si recibe un entero o un
# flotante como argumento, y False en caso contrario
def is_numeric(x):
    if type(x) == float or type(x) == int:
        return True
    return False
```

[48]:
```
# Definimos una lista que contiene enteros, reales, cadenas, listas y
# un valor None
a_list_with_str_and_nums = [1, 2, "three", "four", 5, 5.5, 6, [0, 0, 1], None]

# Aplicamos filter de la lista con la función is_numeric
r = list(filter(is_numeric, a_list_with_str_and_nums))
print("The filtered list is {}".format(r))
```

```
The filtered list is [1, 2, 5, 5.5, 6]
```

2.4.3 1.4.3.- Reduce

La función `reduce` recibe, de nuevo, una función y un iterable. La función aplica, de manera acumulativa, la función que recibe como argumento a los elementos del iterable. La función que recibe como argumento debe ser una función que reciba dos parámetros y devuelva un único valor. Entonces, `reduce` aplica la función a los dos primeros valores del iterable, después al tercer valor y al resultado de la operación anterior, etc. Por ejemplo, sea 'f' la función a aplicar y [1, 2, 3, 4] el iterable, `reduce` calcularía:

```
f(f(f(1, 2), 3), 4)
```

Veamos algunos ejemplos:

[49]:
```
# Importamos reduce del módulo functools
from functools import reduce

# Calculamos la suma de los valores de una lista
a_list = [1, 2, 3, 4]
reduce(sum_two_values, a_list)
```

[49]:
```
10
```

Recordad que la función `sum_two_values` recibía dos parámetros y devolvía un solo valor, correspondientes a la suma de los parámetros. Así, al aplicarla con `reduce` sobre una lista, lo que obtenemos es la suma de los valores de la lista: se aplica la función `sum_two_values` los dos primeros valores, 1 y 2, lo que da como resultado 3; se vuelve a aplicar la función con 3 y 3 como argumentos (el resultado de la primera llamada a `sum_two_values` y el tercer valor de la lista), resultando con 6; y finalmente se llama a la función con 6 y 4 (el resultado de la última llamada a la función y el valor del último elemento de la lista); el resultado final es pues 10.

```
sum_two_values(sum_two_values(sum_two_values(1, 2), 3), 4) =
= Sum_two_values(sum_two_values(3, 3), 4)
= Sum_two_values(6, 4)
= 10
```

De manera análoga, podemos utilizar `reduce` para calcular el valor máximo de una lista, definiendo primero una función que devuelva el máximo de dos valores, y aplicándola sobre una lista con `reduce`.

[50]:
```
def max_two(x, y):
    """Return the max of two values, max(x, y)."""
    if x > y:
        return x
    else:
        return y


# Aplicamos reduce con max_two sobre una lista para calcular el máximo
a_list = [10, 1, 15, 19, 30]
reduce(max_two, a_list)
```

[50]:
```
30
```

En este caso, la ejecución del `reduce` sería la siguiente:

```
max_two(max_two(max_two(max_two(10, 1), 15), 19), 30) =
= max_two(max_two(max_two(10, 15), 19), 30) =
= max_two(max_two(15, 19), 30) =
= max_two(19, 30) =
= 30
```

Veamos un tercer ejemplo de uso de `reduce` que nos permite 'aplanar' listas, es decir, dada una lista de listas, obtener una lista que contenga los elementos de las listas interiores:

[51]:
```
def join_lists(x, y):
    """Concatenate two lists."""
    return x + y


# Llamamos join_lists con dos listas, para ver su efecto
join_lists([1, 2], [3, 4])
```

[51]:
```
[1, 2, 3, 4]
```

[52]:
```
# Usamos join_lists con reduce para aplanar una lista
list_to_flatten = [[0], [1, 2, 3], [5]]
reduce(join_lists, list_to_flatten)
```

[52]:
```
[0, 1, 2, 3, 5]
```

Opcionalmente, la función `reduce` también puede recibir un tercer parámetro, que corresponde al valor inicial. Si está presente, entonces la primera llamada a la función se hace con el valor inicial y el primer valor del iterable, en vez de usar los dos primeros valores del iterable.

[53]:
```
# Usamos reduce con sum_two_values para sumar una lista,
# indicando 0 como valor inicial para asegurar la corrección del resultado
a_list = [1, 2, 3, 4]
reduce(sum_two_values, a_list, 0)
```

[53]:
```
10
```

[54]:
```
# Usamos el valor inicial de reduce para añadir 15 al sumatorio de una lista
reduce(sum_two_values, a_list, 15)
```

[54]:
```
25
```

[55]:
```
# Usamos reduce con max_two para calcular el máximo de una lista
# indicando 0 como valor inicial, por lo que obtendremos el máximo de la
# lista si hay algún valor positivo o 0 si todos los números de
# la lista son negativos
print(reduce(max_two, [10, 1, 15, 19, 30], 0))
print(reduce(max_two, [-5, -3, -2], 0))
```

```
30
0
```

Para terminar esta sección, veremos un ejemplo que combina las tres funciones que acabamos de presentar (`reduce`, `map`, y `filter`). Supongamos que queremos definir una función que reciba una lista y devuelva el producto de los cuadrados de todos los elementos impares de la lista. Combinando las tres funciones que hemos visto, podemos hacer este cálculo en una sola línea:

[56]:
```
# Definimos una función que devuelve el producto de dos valores
def prod_two_values(x, y):
    return x*y


# Definimos una función que devuelve el cuadrado de un valor
def sq(x):
    return x**2


# Definimos una función que devuelve True si un valor es impar
# y False en caso contrario
def is_odd(x):
    return x % 2


# Definimos la función que devuelve el producto de los cuadrados de los impares
# de una lista
def prod_sq_if_odd(l):
    return reduce(prod_two_values, map(sq, filter(is_odd, l)))
```

[57]:
```
# Ejecutamos la función prod_sq_if_odd para una lista y comprobamos
# el resultado 'manualmente'
a_list = [5, 9, 2, 12, 15]
print(prod_sq_if_odd(a_list))
print(5**2 * 9**2 * 15**2)
```

```
455625
455625
```

2.5 1.5.- Funciones anónimas

Las funciones anónimas son un tipo especial de funciones que no tienen nombre y que se encuentran limitadas a una sola expresión. Este tipo de funciones son útiles cuando necesitamos una función normalmente simple que no querremos reutilizar (o bien la reutilizaremos muy poco) en nuestro código, y se utilizan a menudo en combinación con las funciones de programación funcional que hemos visto en la sección anterior.

Las funciones anónimas sólo pueden tener una única expresión. Esta expresión es evaluada con los argumentos cuando se llama a la función, y el resultado de esta evaluación es el valor que devuelve la función.

Para definir una función anónima en Python se utiliza la palabra clave `lambda`, seguida de los parámetros de la función, unos dos puntos, y la expresión de retorno de la función. Por este motivo, también utilizamos la expresión función lambda para nombrar a las funciones anónimas.

La celda siguiente muestra una función anónima que suma dos valores:

[58]:
```
# Definimos una función lambda que suma dos valores
lambda x, y: x+y
```

[58]:
```
<function __main__.<lambda>(x, y)>
```

Podemos llamar a una función lambda tanto directamente (rodeándola de paréntesis y pasándole los argumentos), o bien asignándole un nombre:

[59]:
```
# Definimos una función lambda que suma dos valores y la llamamos
# con los valores 15 y 19
(lambda x, y: x+y)(15, 19)
```

[59]:
```
34
```

[60]:
```
# Definimos una función lambda y le asignamos el nombre suma,
# dara poderla llamar posteriormente
suma = lambda x, y: x+y
suma(15, 19)
```

[60]:
```
34
```

```
3:1: E731 do not assign a lambda expression, use a def
```

Normalmente, sin embargo, encontraremos las funciones lambda dentro de otras expresiones. De hecho, el ejemplo anterior se considera mala praxis, y se incluye únicamente para ayudar en la comprensión de este tipo de funciones.

Un ejemplo de una situación en la que utilizaríamos funciones anónimas es el último ejemplo de la sección anterior, donde hemos tenido que definir tres funciones muy simples, que seguramente no volveremos a utilizar en ninguna otra parte del código, a fin de poder hacer el cálculo de `prod_sq_if_odd`. Utilizando funciones anónimas, podríamos reducir la definición de la función `prod_sq_if_odd` a una sola expresión, sin necesidad de definir con anterioridad las tres funciones auxiliares como hemos hecho antes:

[61]:
```
def prod_sq_if_odd_with_lamb(l):
    return reduce(lambda x, y: x*y,
                  map(lambda x: x**2,
                      filter(lambda x: x % 2, l)))
```

En este caso, en vez de definir las funciones `prod_two_values`, `sq` y `is_odd`, hemos utilizado funciones anónimas equivalentes, por lo que la definición de la función`prod_sq_if_odd_with_lamb` es mucho más compacta.

A continuación, veremos cómo reescribir algunos de los otros ejemplos de programación funcional que hemos visto en este notebook usando funciones anónimas:

[62]:
```
# Sumamos 2 a cada uno de los elementos de la lista a_list
# utilizando una función anónima
a_list = [1, 2, 3, 4]
r = map(lambda x: x + 2, a_list)
print(list(r))
```

```
[3, 4, 5, 6]
```

[63]:
```
# Convertimos las cadenas de caracteres que expresan valores numéricos
# (posiblemente decimales) de una lista en enteros utilizando
# una función anónima
a_list = ["42.45", "13.4", "12000"]
list(map(lambda x: int(float(x)), a_list))
```

[63]:
```
[42, 13, 12000]
```

[64]:
```
# Filtramos de una lista los valores numéricos
a_list_with_str_and_nums = [1, 2, "three", "four", 5, 5.5, 6, [0, 0, 1], None]
r = list(filter(
    lambda x: True if type(x) == float or type(x) == int else False,
    a_list_with_str_and_nums))
print("The filtered list is {}".format(r))
```

```
The filtered list is [1, 2, 5, 5.5, 6]
```

[65]:
```
# Calculamos el máximo de una lista con reduce y una función anónima
# que devuelve el máximo de dos valores
a_list = [10, 1, 15, 19, 30]
reduce(lambda x, y: x if x > y else y, a_list)
```

[65]:
```
30
```

2.6 1.6.- Docstring

Los *docstrings* son cadenas de texto que contienen la documentación de las funciones, entre otros objetos, en Python. Ya hemos ido utilizando *docstring* para documentar algunas de las funciones de este notebook. A continuación repasaremos algunos detalles sobre su funcionamiento y sobre las convenciones que se utilizan a la hora de escribirlos.

Podemos acceder al *docstring* a través de la función `help` o, directamente, a través del atributo `__doc__`.

[66]:
```
# Accedemos al docstring de la función built-in max con help
help(max)
```

```
Help on built-in function max in module builtins:

max(...)
    max(iterable, *[, default=obj, key=func]) -> value
    max(arg1, arg2, *args, *[, key=func]) -> value

    With a single iterable argument, return its biggest item. The
    default keyword-only argument specifies an object to return if
    the provided iterable is empty.
    With two or more arguments, return the largest argument.
```

[67]:
```
# Accedemos al docstring de la función built-in max a través del atributo doc
max.__doc__
```

[67]:
```
'max(iterable, *[, default=obj, key=func]) -> value\nmax(arg1, arg2, *args, *[,
key=func]) -> value\n\nWith a single iterable argument, return its biggest item.
The\ndefault keyword-only argument specifies an object to return if\nthe
provided iterable is empty.\nWith two or more arguments, return the largest
argument.'
```

Las convenciones sobre el uso de *docstring* en Python se encuentran descritas en PEP-257. Los *docstrings* se definen como primera sentencia dentro de la definición de la función y distinguimos entre *docstrings* de una única línea y *docstrings* multilínea.

La convención en la definición de *docstrings* de una sola línea es usar triples comillas dobles para indicar el comentario (aunque no sea necesario ya que el comentario ocupe únicamente una línea), escribir las comillas de inicio y cierre en la misma línea del texto, comenzar la frase del comentario en mayúsculas y terminar en un punto, y no incluir salto de línea ni antes ni después del comentario.

Así, por ejemplo, este sería un *docstring* de una sola línea que sigue las convenciones:

[68]:
```
def sum_two_values(x, y):
    """Return the value of the sum of the parameters."""
    return x + y
```

[69]:
```
help(sum_two_values)
```

```
Help on function sum_two_values in module __main__:

sum_two_values(x, y)
    Return the value of the sum of the parameters.
```

De manera similar, los *docstrings* multilínea se delimitan también con triples comillas dobles. En este caso, consisten en una línea de resumen, una línea en blanco, y luego una descripción más completa, que se puede extender varias líneas.

El ***docstring* de una función debería resumir su comportamiento**, y listar sus argumentos, valor de retorno, las excepciones que puede lanzar, los efectos colaterales que puede tener su llamada, y las restricciones a tener en cuenta cuando la llamamos.

[70]:
```
def sum_two_values(x, y=0):
    """Return the value of the sum of the parameters.

    This function accepts up to two numbers, and returns
    the sum of them.

    Positional arguments:
    x -- the first number
    y -- the second number (optional, default 0)

    Returns:
    number: Sum of the values
    """
    return x + y
```

[71]:
```
help(sum_two_values)
```

```
Help on function sum_two_values in module __main__:

sum_two_values(x, y=0)
    Return the value of the sum of the parameters.

    This function accepts up to two numbers, and returns
    the sum of them.

    Positional arguments:
    x -- the first number
    y -- the second number (optional, default 0)

    Returns:
    number: Sum of the values
```

Hay diferentes formatos para escribir el *docstring*, que detallan cómo se deben especificar los argumentos, los valores de retorno, las excepciones, etc. Recomendamos leer el post de stackoverflow sobre los formatos de dosctring the Python para ver algunos ejemplos (lectura opcional, para ampliar conocimientos).

Por último, es importante notar que no sólo las funciones hacen uso de *docstrings* para documentar su comportamiento. Los módulos, las clases, y los métodos también se documentan con *docstring*.

[72]:
```
# Accedemos al docstring de la clase int
help(int)
```

```
Help on class int in module builtins:

class int(object)
 |  int(x=0) -> integer
```

```
|  int(x, base=10) -> integer
|
|  Convert a number or string to an integer, or return 0 if no arguments
|  are given.  If x is a number, return x.__int__().  For floating point
|  numbers, this truncates towards zero.
|
|  If x is not a number or if base is given, then x must be a string,
|  bytes, or bytearray instance representing an integer literal in the
|  given base.  The literal can be preceded by '+' or '-' and be surrounded
|  by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
|  Base 0 means to interpret the base from the string as an integer literal.
|  >>> int('0b100', base=0)
|  4
|
|  Methods defined here:
|
|  __abs__(self, /)
|      abs(self)
|
|  __add__(self, value, /)
|      Return self+value.
|
|  __and__(self, value, /)
|      Return self&value.
|
|  __bool__(self, /)
|      self != 0
|
|  __ceil__(...)
|      Ceiling of an Integral returns itself.
|
|  __divmod__(self, value, /)
|      Return divmod(self, value).
|
|  __eq__(self, value, /)
|      Return self==value.
|
|  __float__(self, /)
|      float(self)
|
|  __floor__(...)
|      Flooring an Integral returns itself.
|
|  __floordiv__(self, value, /)
|      Return self//value.
|
|  __format__(...)
|      default object formatter
```

```
 |
 |  __ge__(self, value, /)
 |      Return self>=value.
 |
 |  __getattribute__(self, name, /)
 |      Return getattr(self, name).
 |
 |  __getnewargs__(...)
 |
 |  __gt__(self, value, /)
 |      Return self>value.
 |
 |  __hash__(self, /)
 |      Return hash(self).
 |
 |  __index__(self, /)
 |      Return self converted to an integer, if self is suitable for use as an
index into a list.
 |
 |  __int__(self, /)
 |      int(self)
 |
 |  __invert__(self, /)
 |      ~self
 |
 |  __le__(self, value, /)
 |      Return self<=value.
 |
 |  __lshift__(self, value, /)
 |      Return self<<value.
 |
 |  __lt__(self, value, /)
 |      Return self<value.
 |
 |  __mod__(self, value, /)
 |      Return self%value.
 |
 |  __mul__(self, value, /)
 |      Return self*value.
 |
 |  __ne__(self, value, /)
 |      Return self!=value.
 |
 |  __neg__(self, /)
 |      -self
 |
 |  __new__(*args, **kwargs) from builtins.type
 |      Create and return a new object.  See help(type) for accurate signature.
```

```
|
|  __or__(self, value, /)
|      Return self|value.
|
|  __pos__(self, /)
|      +self
|
|  __pow__(self, value, mod=None, /)
|      Return pow(self, value, mod).
|
|  __radd__(self, value, /)
|      Return value+self.
|
|  __rand__(self, value, /)
|      Return value&self.
|
|  __rdivmod__(self, value, /)
|      Return divmod(value, self).
|
|  __repr__(self, /)
|      Return repr(self).
|
|  __rfloordiv__(self, value, /)
|      Return value//self.
|
|  __rlshift__(self, value, /)
|      Return value<<self.
|
|  __rmod__(self, value, /)
|      Return value%self.
|
|  __rmul__(self, value, /)
|      Return value*self.
|
|  __ror__(self, value, /)
|      Return value|self.
|
|  __round__(...)
|      Rounding an Integral returns itself.
|      Rounding with an ndigits argument also returns an integer.
|
|  __rpow__(self, value, mod=None, /)
|      Return pow(value, self, mod).
|
|  __rrshift__(self, value, /)
|      Return value>>self.
|
|  __rshift__(self, value, /)
```

```
 |      Return self>>value.
 |
 |  __rsub__(self, value, /)
 |      Return value-self.
 |
 |  __rtruediv__(self, value, /)
 |      Return value/self.
 |
 |  __rxor__(self, value, /)
 |      Return value^self.
 |
 |  __sizeof__(...)
 |      Returns size in memory, in bytes
 |
 |  __str__(self, /)
 |      Return str(self).
 |
 |  __sub__(self, value, /)
 |      Return self-value.
 |
 |  __truediv__(self, value, /)
 |      Return self/value.
 |
 |  __trunc__(...)
 |      Truncating an Integral returns itself.
 |
 |  __xor__(self, value, /)
 |      Return self^value.
 |
 |  bit_length(...)
 |      int.bit_length() -> int
 |
 |      Number of bits necessary to represent self in binary.
 |      >>> bin(37)
 |      '0b100101'
 |      >>> (37).bit_length()
 |      6
 |
 |  conjugate(...)
 |      Returns self, the complex conjugate of any int.
 |
 |  from_bytes(...) from builtins.type
 |      int.from_bytes(bytes, byteorder, *, signed=False) -> int
 |
 |      Return the integer represented by the given array of bytes.
 |
 |      The bytes argument must be a bytes-like object (e.g. bytes or
bytearray).
```

```
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order
value.
 |
 |      The signed keyword-only argument indicates whether two's complement is
 |      used to represent the integer.
 |
 |  to_bytes(...)
 |      int.to_bytes(length, byteorder, *, signed=False) -> bytes
 |
 |      Return an array of bytes representing an integer.
 |
 |      The integer is represented using length bytes.  An OverflowError is
 |      raised if the integer is not representable with the given number of
 |      bytes.
 |
 |      The byteorder argument determines the byte order used to represent the
 |      integer.  If byteorder is 'big', the most significant byte is at the
 |      beginning of the byte array.  If byteorder is 'little', the most
 |      significant byte is at the end of the byte array.  To request the native
 |      byte order of the host system, use `sys.byteorder' as the byte order
value.
 |
 |      The signed keyword-only argument determines whether two's complement is
 |      used to represent the integer.  If signed is False and a negative
integer
 |      is given, an OverflowError is raised.
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  denominator
 |      the denominator of a rational number in lowest terms
 |
 |  imag
 |      the imaginary part of a complex number
 |
 |  numerator
 |      the numerator of a rational number in lowest terms
 |
 |  real
 |      the real part of a complex number
```

3 2.- Ejercicios para practicar

A continuación encontraréis un conjunto de problemas que pueden servir para practicar los conceptos explicados en esta primera unidad, así como para refrescar los conceptos básicos de programación. Os recomendamos que intentéis realizar estos problemas vosotros mismos y que, una vez realizados, comparéis la solución que proponemos con vuestra solución. No dudéis en dirigir todas las dudas que surjan de la resolución de estos ejercicios o bien de las soluciones propuestas al foro del aula.

1. Definid una función que reciba como parámetros dos valores (`x` e `y`) que serán dos vectores de enteros, y devuelva la distancia euclídea entre los puntos representados por los vectores. Es necesario que el cuerpo de la función contenga una única expresión, que calcule y devuelva el resultado. Los vectores pueden tener un tamaño arbitrario, pero ambos vectores tendrán el mismo número de elementos. Sólo se pueden utilizar funciones de la librería estándar de Python.

[73]:
```
# Respuesta
```

2. Definid una función que reciba como parámetros dos valores (`x` e `y`) que serán dos vectores de enteros, y devuelva la distancia de Manhattan entre los puntos representados por los vectores. Es necesario que el cuerpo de la función contenga una única expresión, que calcule y devuelva el resultado. Los vectores pueden tener un tamaño arbitrario, pero ambos vectores tendrán el mismo número de elementos. Sólo se pueden utilizar funciones de la librería estándar de Python.

[74]:
```
# Respuesta
```

3. Definid una función `compute_all_distances` que reciba como parámetros dos valores (`x` e `y`) que serán dos vectores de enteros, y devuelva una tupla de dos elementos, con las distancias euclidiana y de Manhattan entre los puntos representados por los vectores. Los vectores pueden tener un tamaño arbitrario, pero ambos vectores tendrán el mismo número de elementos. Sólo se pueden utilizar funciones de la librería estándar de Python.

Para ello, encapsulad el código de las funciones de las actividades anteriores dentro de la función `compute_all_distances`.

[75]:
```
# Respuesta
```

4. En la frutería del barrio tienen un problema que requiere de nuestra ayuda. Reiteradamente, se les rompen las estanterías donde ponen las naranjas, y quieren evitar que esto vuelva a pasar. Han calculado que los estantes de madera soportan sin problemas un peso de 50 kilos, y los de plástico 30 kilos, pero siempre dudan de si pueden añadir algún piso de naranjas más (ya que esto siempre luce más delante de los clientes).

Las naranjas se encuentran apiladas en una pirámide de base cuadrada. Así pues, en lo alto hay una sola naranja, en el segundo piso hay 4, en el tercer piso hay 9, etc. Los pisos siempre están completos.

Definid una función que reciba como parámetros el número de pisos de naranjas que quieren hacer, el peso medio de cada naranja, y el tipo de material del estante, y devuelva un booleano indicando si el estante aguantará el peso o no. La función siempre recibirá el número de pisos de naranjas, pero los parámetros de peso medio y material son opcionales, y tomarán un valor por defecto de 0.2 y madera (`"Wood"`), respectivamente.

[76]:
```
# Respuesta
```

5. Definid una función que pueda recibir un número de parámetros cualquiera, superior a 1, y que devuelva el resultado de sumar el resultado de aplicar la función que recibe como primer parámetro a cada uno de los otros parámetros.

Por ejemplo, si la función recibe como primer argumento una función que calcula cuadrados, como segundo argumento un 5, y como tercer argumento un 10, debería devolver $ 5 ^ 2 + 10 ^ 2 = 125 $.

Llamad a la función definida con los valores del ejemplo mencionado en el enunciado, y comprobad que se obtiene el resultado correcto. Compruebad también que la función devuelve los resultados esperados para una llamada con 2 argumentos y con 5 argumentos.

[77]:
```
# Respuesta
```

6. Definid una función que reciba dos parámetros, una lista de enteros y un entero, y devuelva una lista con los mismos elementos que la lista original eliminando todas las apariciones del entero especificado.

6.1. Implementad una función que modifique la lista original. Haced una llamada a la función definida y mostrad que, efectivamente, la lista original se modifica.

[78]:
```
# Respuesta
```

6.2. Implementad una función que **no** modifique la lista original. Haced una llamada a la función definida y mostrad que, efectivamente, no se modifica la lista original.

[79]:
```
# Respuesta
```

3.1 2.1.- Soluciones a los ejercicios para practicar

1. Definid una función que reciba como parámetros dos valores (x e y) que serán dos vectores de enteros, y devuelva la distancia euclídea entre los puntos representados por los vectores. Es necesario que el cuerpo de la función contenga una única expresión, que calcule y devuelva el resultado. Los vectores pueden tener un tamaño arbitrario, pero ambos vectores tendrán el mismo número de elementos. Sólo se pueden utilizar funciones de la librería estándar de Python.

[80]:
```
# Respuesta

# Importamos math, que está en la librería estándar y permite
# calcular raíces cuadradas
import math
```

```
def eucl_dist(x, y):
    # Definimos la función, que calcula la distancia utilizando una list
    # comprehension uniendo los valores de los vectores x e y con zip
    return math.sqrt(sum([(coord[0]-coord[1])**2 for coord in zip(x, y)]))
```

2. Definid una función que reciba como parámetros dos valores (x e y) que serán dos vectores de enteros, y devuelva la distancia de Manhattan entre los puntos representados por los vectores. Es necesario que el cuerpo de la función contenga una única expresión, que calcule y devuelva el resultado. Los vectores pueden tener un tamaño arbitrario, pero ambos vectores tendrán el mismo número de elementos. Sólo se pueden utilizar funciones de la librería estándar de Python.

[81]:
```
# Respuesta


def manh_dist(x, y):
    return sum([abs(coord[0]-coord[1]) for coord in zip(x, y)])
```

3. Definid una función `compute_all_distances` que reciba como parámetros dos valores (x e y) que serán dos vectores de enteros, y devuelva una tupla de dos elementos, con las distancias euclidiana y de Manhattan entre los puntos representados por los vectores. Los vectores pueden tener un tamaño arbitrario, pero ambos vectores tendrán el mismo número de elementos. Sólo se pueden utilizar funciones de la librería estándar de Python.

Para ello, encapsulad el código de las funciones de las actividades anteriores dentro de la función `compute_all_distances`.

[82]:
```
# Respuesta


def compute_all_distances(x, y):

    def eucl_dist(x, y):
        return math.sqrt(sum([(coord[0]-coord[1])**2 for coord in zip(x, y)]))

    def manh_dist(x, y):
        return sum([abs(coord[0]-coord[1]) for coord in zip(x, y)])

    return (eucl_dist(x, y), manh_dist(x, y))
```

4. En la frutería del barrio tienen un problema que requiere de nuestra ayuda. Reiteradamente, se les rompen las estanterías donde ponen las naranjas, y quieren evitar que esto vuelva a pasar. Han calculado que los estantes de madera soportan sin problemas un peso de 50 kilos, y los de plástico 30 kilos, pero siempre dudan de si pueden añadir algún piso de naranjas más (ya que esto siempre luce más delante de los clientes).

Las naranjas se encuentran apiladas en una pirámide de base cuadrada. Así pues, en lo alto hay una sola naranja, en el segundo piso hay 4, en el tercer piso hay 9, etc. Los pisos siempre están completos.

Definid una función que reciba como parámetros el número de pisos de naranjas que quieren hacer, el peso medio de cada naranja, y el tipo de material del estante, y devuelva un booleano indicando si el estante aguantará el peso o no. La función siempre recibirá el número de pisos de naranjas, pero los parámetros de peso medio y material son opcionales, y tomarán un valor por defecto de 0.2 y madera ("Wood"), respectivamente.

[83]:
```
# Respuesta


def will_it_hold(num_floors, avg_weight=0.2, mat="Wood"):
    max_weights = {"Wood": 50, "Plastic": 30}
    num_org = sum([i**2 for i in range(1, num_floors+1)])
    return num_org * avg_weight <= max_weights[mat]
```

[84]:
```
will_it_hold(7, mat="Wood")
```

[84]:
```
True
```

5. Definid una función que pueda recibir un número de parámetros cualquiera, superior a 1, y que devuelva el resultado de sumar el resultado de aplicar la función que recibe como primer parámetro a cada uno de los otros parámetros.

Por ejemplo, si la función recibe como primer argumento una función que calcula cuadrados, como segundo argumento un 5, y como tercer argumento un 10, debería devolver $ 5 ˆ 2 + 10 ˆ 2 = 125 $.

Llamad a la función definida con los valores del ejemplo mencionado en el enunciado, y comprobad que se obtiene el resultado correcto. Compruebad también que la función devuelve los resultados esperados para una llamada con 2 argumentos y con 5 argumentos.

[85]:
```
# Respuesta


def apply_and_sum(f, *others):
    return sum((map(f, others)))
```

[86]:
```
apply_and_sum(lambda x: x ** 2, 5, 10)
```

[86]:
```
125
```

[87]:
```
apply_and_sum(lambda x: x ** 2, 1)
```

[87]:
```
1
```

[88]:
```
apply_and_sum(lambda x: x ** 2, 1, 2, 3, 4)
```

[88]:
```
30
```

6. Definid una función que reciba dos parámetros, una lista de enteros y un entero, y devuelva una lista con los mismos elementos que la lista original eliminando todas las apariciones del entero especificado.

6.1. Implementad una función que modifique la lista original. Haced una llamada a la función definida y mostrad que, efectivamente, la lista original se modifica.

[89]:
```
# Respuesta


def delete_ints(l, i):
    while i in l:
        l.remove(i)
    return l


original_list = [101, 1001, 10, 55, 10, 1]
print("List before call: {}".format(original_list))
new_list = delete_ints(original_list, 10)
print("List after call: {}".format(original_list))
print("Return result: {}".format(new_list))
```

```
List before call: [101, 1001, 10, 55, 10, 1]
List after call: [101, 1001, 55, 1]
Return result: [101, 1001, 55, 1]
```

6.2. Implementad una función que **no** modifique la lista original. Haced una llamada a la función definida y mostrad que, efectivamente, no se modifica la lista original.

[90]:
```
# Respuesta


def delete_ints(l, i):
    new_list = [e for e in l if e != i]
    return new_list


original_list = [101, 1001, 10, 55, 10, 1]
print("List before call: {}".format(original_list))
new_list = delete_ints(original_list, 10)
print("List after call: {}".format(original_list))
print("Return result: {}".format(new_list))
```

```
List before call: [101, 1001, 10, 55, 10, 1]
List after call: [101, 1001, 10, 55, 10, 1]
Return result: [101, 1001, 55, 1]
```

4 3.- Bibliografía

4.1 3.1.- Bibliografía básica

Os recomendamos revisar la documentación oficial de las funciones y clases descritas en esta unidad, que encontraréis enlazadas en cada uno de los apartados que las describen.

4.2 3.2.- Bibliografía adicional - Ampliación de conocimientos

Para profundizar en el funcionamiento del paso de parámetros en funciones en Python, se recomienda consultar este post de medium

Al notebook hemos visto cómo definir funciones dentro de funciones y las consecuencias que esto tiene sobre la visibilidad de las variables. También hemos comentado una de las utilidades de esta definición, que es la de encapsular código. Otra de las utilidades es la definición de *closures*. Si desea conocer esta construcción, se recomienda la lectura del artículo Python nested functions.

Como hemos comentado, hay diferentes formatos para escribir el *docstring*. Recomendamos leer el post de stackoverflow sobre los formatos de dosctring the Python para ver algunos ejemplos .