2-?s avan?at de funcions

March 13, 2020

1 Programació per a la ciència de dades

1.1 Unitat 2: Ús avançat de funcions en Python

1.1.1 Instruccions d'ús

Aquest document és un notebook interactiu que intercala explicacions més aviat teòriques de conceptes de programació amb fragments de codi executables. Per aprofitar els avantatges que aporta aquest format, us recomanem que, en primer lloc, llegiu les explicacions i el codi que us proporcionem. D'aquesta manera tindreu un primer contacte amb els conceptes que hi exposem. Ara bé, **la lectura és només el principi!** Una vegada hagueu llegit el contingut proporcionat, no oblideu executar el codi proporcionat i modificar-lo per crear-ne variants, que us permetin comprovar que heu entès la seva funcionalitat i explorar-ne els detalls d'implementació. Per últim, us recomanem també consultar la documentació enllaçada per explorar amb més profunditat les funcionalitats dels mòduls presentats.

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

[2]:
```
# Activem les alertes d'estil
%pycodestyle_on
```

1.1.2 Introducció

En aquesta unitat es repassen molt ràpidament els conceptes bàsics de funcions i s'exposen alguns conceptes més avançats sobre l'ús de funcions en Python.

En primer lloc, s'explica què és l'àmbit (l'*scope*) d'una variable, un concepte que de ben segur coneixeu de manera informal, però que no s'ha treballat formalment encara.

En segon lloc, es repassa com retornar valors des de funcions, es comenten els possibles valors que pot retornar una funció i es presenten algunes situacions que poden semblar especials, com ara les funcions que retornen funcions.

A continuació, s'exposa com funcionen els paràmetres de les funcions en Python. Es parla de com es passen aquests paràmetres i què passa quan els modifiquem, de la definició de funcions amb arguments opcionals i amb un número d'arguments indeterminat, i també de com passar funcions com a arguments d'altres funcions.

Seguidament, es fa una petita introducció a la programació funcional en Python, presentant les tres funcions bàsiques d'aquest paradigma de programació (`map`, `filter` i `reduce`).

Després, es presenten les funcions anònimes, i es posen exemples d'ús d'aquest tipus de funcions quan es fa servir `map`, `filter` i `reduce`.

Finalment, s'explica que és el *docstring*, com hi podem accedir i quines convencions s'utilitzen a l'hora de definir el *docstring* de les funcions.

A continuació s'inclou la taula de continguts, que podeu fer servir per a navegar pel document:

---

2 1.- Funcions

Com ja sabem, una funció és una manera d'encapsular codi, que ens permet reaprofitar-lo per a diverses tasques. Una funció és un fragment de codi que té un nom i que realitza una tasca específica.

En Python definim una funció amb la paraula reservada `def`, seguida del nom que donem a la funció i dels paràmetres que rep entre parèntesi. Una funció pot retornar un valor, que s'indica amb la paraula `return`.

[3]:
```
# Definim la funció de nom 'sum_two_values' amb dos arguments: 'x' i 'y':
def sum_two_values(x, y):
    """Return the value of the sum."""
    return x + y
```

Per tal d'executar una funció, la cridem fent servir el seu nom, i especificant els arguments:

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

```
8
```

2.1 1.1.- Àmbit de visibilitat

Com ja hem vist, l'assignació d'una variable en qualsevol celúla d'un notebook permet fer servir aquesta variable en tot el notebook. Així doncs, per exemple, podem mostrar el contingut de la variable `r` assignada a la celúla anterior en la següent celúla:

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

```
8
```

Si intentem fer servir una variable que no haguem assignat anteriorment, Python llança l'excepció `NameError`:

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

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

Aquest error apareix ja que la variable `undef_var` no s'ha assignat amb anterioritat a l'execució del `print` sobre la variable.

Recordeu que les diferents celúles d'un notebook poden executar-se de manera independent, i que l'ordre d'execució de les celúles (i no pas l'ordre en què es troben dins del notebook) en determinarà el flux d'execució. Quan programem fent servir notebooks **és molt recomanable que aquests es puguin executar linealment sense errors**, és a dir, que el resultat esperat del notebook sigui el que s'obté al prémer el botó del Menú `Kernel`, `Restart and Run All`.

A tall d'exemple i perquè pugueu comprovar la possible no linealitat d'execució d'un notebook, a la celúla següent definirem la variable `undef_var` i us demanem que executeu de nou la celúla anterior (on es feia un `print` d'aquesta variable). A diferència de la primera vegada en què s'executa la celúla, aquest cop no ens hauria de donar cap error, ja que ja haurem definit la variable. Aquest tipus d'accions, però, són les que cal evitar, ja que dificulten la lectura, claredat i portabilitat del codi que implementem.

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

#####################################################################
# IMPORTANT: executeu ara la celúla de codi anterior, que conté
# el print de la variable undef_var, per comprovar que l'execució
# ara no genera cap error.
#####################################################################
```

Diem que una variable és **global** quan aquesta és assignada fora d'una funció (com en el cas tant de la variable `r` com de la variable `undef_var`). Per contra, direm que una variable és **local** quan la definim dins d'una funció. Per referir-nos a l'àrea des d'on pot fer-se servir una determinada variable fem servir la paraula **àmbit** (en anglès, parlem d'*scope*).

Una variable local no pot ser utilitzada fora del cos de la funció on està definida. En canvi, una variable global pot ser utilitzada tant des de fora de qualsevol funció com des del cos de les funcions:

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


def fun_1():
    # Definim una variable local dins de fun_1
    local_var_1 = "local_var_1 is local to fun_1"
    # Mostrem el valor de la variable global i de la local
    print(global_var)
    print(local_var_1)


def fun_2():
    # Mostrem la variable global
    print(global_var)
    # Intentem accedir a la variable local local_var_1,
    # cosa que generarà un error ja que no està definida
    # dins de la funció fun_2
```

```
    print(local_var_1)
```

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

```
This is a global variable
```

[10]:
```
# Comprovem com no podem accedir a la variable local local_var_1
# ja que aquesta està definida dins de l'àmbit de la funció fun_1
try:
    print(local_var_1)
except NameError:
    print("Error: Variable no definida")
```

```
Error: Variable no definida
```

[11]:
```
# Executem fun_1, que mostra correctament el valor de la variable
# global i de la local
fun_1()
```

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

[12]:
```
# Executem fun_2, que mostra correctament el valor de la variable
# global però llança una excepció a l'accedir a la variable local
# ja que aquesta està definida a fun_1
try:
    fun_2()
except NameError:
    print("Error: Variable no definida")
```

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

Fins ara hem vist el comportament de les variables globals i locals dins i fora de funcions. Què passa, però, si definim una variable local (dins d'una funció) que té el mateix nom que una variable global definida fora de la funció?

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


# Mostrem el valor de global_var abans d'executar la funció fun_3
print(global_var)
```

```
# Executem la funció fun_3
fun_3()
# Mostrem el valor de global_var després d'executar la funció fun_3
print(global_var)
```

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

D'una banda, fixeu-vos que el valor de la variable global `global_var` no canvia amb l'execució de la funció: el `print` d'abans de la crida a `fun_3` i el `print` de després de la crida retornen exactament el mateix valor. D'altra banda, fixeu-vos que dins de la funció `fun_3`, la variable `global_var` pren un nou valor. En realitat, el que hem fet a l'assignar `global_var` dins de la funció és crear una nova variable amb el mateix nom, dins de l'àmbit de la funció. Aquest comportament difereix del que passava al cridar les funcions `fun_1` i `fun_2`, on el nom `global_var` feia referència a la variable global. La diferència és deguda al fet que en les funcions 1 i 2 simplement es mostrava el contingut de la variable, mentre que en la funció 3 la variable s'assigna.

Així doncs, com podem reassignar el valor d'una variable global des de dins d'una funció? Per fer-ho, caldrà fer ús de la paraula reservada `global`, que permetrà indicar, en el context d'una funció, que volem fer servir una variable global:

[14]:
```
def fun_4():
    # Identifiquem la variable global_var com a global
    global global_var
    # Assignem un nou valor a la variable global global_var
    global_var = "Changing the value of the global var"
    # Mostrem el valor de global_var
    print(global_var)


# Mostrem el valor de global_var abans d'executar la funció fun_4
print(global_var)
# Executem la funció fun_4
fun_4()
# Mostrem el valor de global_var després d'executar la funció fun_4
print(global_var)
```

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

Noteu com, al cridar `fun_4`, es modifica el valor de la variable global `global_var` (el valor de la variable en l'últim print és el valor assignat dins de la funció).

2.1.1 1.1.1.- Funcions dins de funcions

Python permet definir funcions dins d'altres funcions. Aleshores, el comportament de les variables definides en aquestes funcions és similar al que hem descrit anteriorment.

A l'exemple següent hi ha una variable global `y` i una funció `outer_fun` definida també en l'àmbit global. Dins de la funció `outer_fun`, es defineix una variable local `x` amb valor 3, i quatre funcions internes (`inner_fun_i` amb `id`'1 a 4). Cadascuna de les quatre funcions internes mostra els possibles comportaments amb relació a la variable `x` definida a la funció externa: * `inner_fun_1` mostra el valor de la variable `x` definida a `outer_fun`. * `inner_fun_2` assigna a una variable `x` el valor 5. Això crea una nova variable anomenada `x`, local a `inner_fun_2`. Per tant, el valor de la variable `x` d'`outer_fun` no es modifica. * `inner_fun_3` fa servir la paraula clau `nonlocal` per indicar que vol accedir a la variable `x` de l'àmbit d'`outer_fun`, i reassigna el valor d'`x` a 7. Per tant, quan es torna a mostrar el valor d'`x` des de la funció `outer_fun`, el valor ha estat modificat. El `nonlocal` té una funcionalitat similar al `global` que havíem vist en la funció `fun_4`, però serveix per referir-se a una variable definida en un àmbit superior no global. * `inner_fun_4` intenta accedir a la variable `my_var_if1`, que és local a la funció `inner_fun_1`, generant, en conseqüència, una excepció.

[15]:
```
# Definim la funció outer_fun a l'àmbit global
def outer_fun():

    # Definim la funció inner_fun_1 dins d'outer_fun
    def inner_fun_1():
        # Definim la variable my_var_if1 dins de inner_fun_1
        my_var_if1 = "This is defined in inner_fun_1"
        # Mostrem el valor de la variable d'outer_fun x i la variable global y
        print("Inner fun 1 x:\t{}".format(x))
        print("Inner fun 1 y:\t{}".format(y))

    # Definim la funció inner_fun_2 dins d'outer_fun
    def inner_fun_2():
        # Definim i mostrem una nova variable x local a inner_fun_2 amb valor 5
        x = 5
        print("Inner fun 2 x:\t{}".format(x))

    # Definim la funció inner_fun_3 dins d'outer_fun
    def inner_fun_3():
        # Indiquem que farem servir la variable x no local
        # (definida a outer_fun)
        nonlocal x
        # Modifiquem i mostrem el valor d'x
        x = 7
        print("Inner fun 3 x:\t{}".format(x))

    # Definim la funció inner_fun_4 dins d'outer_fun
    def inner_fun_4():
        try:
            # Intentem accedir a la variable my_var_if1 definida dins
            # d'inner_fun_1 (cosa que generarà una excepció)
            print(my_var_if1)
        except NameError:
            print("Error: undefined variable")
```

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

    # Definim una variable local a outer_fun de nom x i valor 3
    x = 3
    # Anem mostrant el valor d'x i executant les funcions internes, per veure
    # l'efecte que tenen 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()


# Definim una variable global y
y = 1
# Cridem la funció outer_fun
outer_fun()
```

```
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 funcions dins de funcions permet **encapsular** codi, és a dir, amagar certes parts del codi, de manera que només puguin ser cridades des de certes funcions. Seguint amb l'exemple anterior, les funcions `inner_fun_i` només són visibles des de dins d'`outer_fun`, i no poden ser cridades des de l'àmbit global:

[16]:
```
try:
    # Intentem executar una de les funcions internes, cosa que generarà
    # una excepció ja que no estan definides en l'àmbit global
    inner_fun_1()
except NameError:
    print("Error: undefined 'inner_fun_1'")
```

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

Per què podem voler encapsular el codi? Suposem, per exemple, que estem analitzant un conjunt de dades amb les localitzacions actuals d'un conjunt de persones, així com la localització del domicili i del lloc de treball d'aquestes, i que volem saber si aquestes persones es troben a prop tant del lloc de treball com del domicili. Per fer-ho, implementem una funció `is_close` que ens retorna un booleà que indica si estan a prop o no d'ambdues localitzacions. Aquesta funció necessitarà calcular la distància entre dos punts dues vegades (una per calcular la distància entre la localització actual i el domicili, i una segona vegada per saber la distància entre la localització actual i el lloc de treball). Per tant, per no repetir codi, definirem una altra funció `dist` que calculi la distància entre dos punts. Ara bé, per a l'anàlisi que volem fer, no farem servir la distància euclidiana, sinó que farem servir la distància de Manhattan. A la resta del nostre codi, però, mai farem servir aquesta definició de distància, i volem evitar que algun altre programador de l'equip, per error, cridi a la nostra funció distància `dist` (pensant, potser, que calcula la distància més habitual, l'Euclidiana). Una possible manera de fer-ho és definir la funció `dist` com a una funció local a `is_close`. D'aquesta manera, podem fer servir una funció i no haurem de repetir codi al calcular les distàncies i, alhora, evitarem que es cridi aquesta funció des de fora de la funció `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)

    # Considerem que un punt és a prop d'un altre si la distància entre ells
    # és inferior a lim
    lim = 10
    # Retornem si (x, y) és a prop tant de (l1_x, l1_y) com de (l2_x, l2_y)
    return (dist(x, y, l1_x, l1_y) < lim) and (dist(x, y, l2_x, l2_y) < lim)
```

[18]:
```
# Calculem si la localització actual (0, 0) és a prop de (1, 4) i (-7, 1)
r1 = is_close(0, 0, 1, 4, -7, 1)
print("(0, 0) is close to (1, 4) and (-7, 1)?: {}".format(r1))

# Calculem si la localització actual (0, 0) és a prop de (1, 4) i (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
```

Abans d'acabar aquesta secció, fem un apunt sobre la implementació de la funció `is_close`. Fixeu-vos que la funció ha de retornar `True` si es compleix una determinada condició (si l'usuari es troba a prop de la feina i del seu habitatge) i `False` en cas contrari. Una possible manera d'implementar aquest comportament és amb una instrucció `if`:

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

```
    return True
else:
    return False
```

Aquest codi fa servir una expressió (que inclou el càlcul de distàncies) per diferenciar si la funció ha de retornar `True` o `False`. És important notar que la pròpia expressió ja retorna un booleà i que, per tant, pot fer-se servir directament com a valor de retorn:

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

de manera que obtenim un codi més concís, que es comporta exactament igual, i que evita redundància.

2.2 1.2.- Valor de retorn

Una funció pot retornar un valor, que s'indica amb la paraula `return`. Si la funció no té `return` o bé té un `return` buit, l'execució de la funció retornarà el valor `None`:

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


# Definim una funció amb retorn buit
def print_sum_v1(x, y):
    print("Result is: {}".format(x+y))
    return


# El valor de retorn de les dues funcions és 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 funció retorna un únic **objecte**. Així, si cal que una funció retorni més d'un valor, podem fer que la funció retorni una tupla amb els diversos valors.

[20]:
```
# Definim la funció de nom 'sum_two_values' amb dos arguments: 'x' i 'y':
def sum_two_values(x, y):
    """Return the value of the sum."""
    return x + y
```

```
# Definim la funció de nom 'sum_and_mult_two_values' amb dos arguments:
# 'x' i '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'>)
```

Fixeu-vos que al retornar la tupla, s'està fent servir la sintaxi de definició de tupla sense parèntesis, que és més ràpida d'escriure ja que conté menys caràcters, però pot portar a confusió al programador novell. Així doncs, noteu que el return de la segona funció seria equivalent a:

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

2.2.1 1.2.1.- Funcions com a valors de retorn

De la mateixa manera que una funció pot retornar, per exemple, un enter, una cadena de caràcters o una tupla, una funció pot retornar també una altra funció. Això ens serà útil en certes construccions que veurem en aquest mòdul.

Per exemple, la funció `min_or_max` definida a la celůla següent retorna o bé la funció `min` o bé la funció `max`, depenent de si el valor que rep com a paràmetre, `sel`, és parell o senar:

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

[22]:
```
# Definim una llista
a_list = [1, 1, 2, 3, 5, 8, 13, 21]
# Cridem a min_or_max amb l'últim element de la llista com a paràmetre
f = min_or_max(a_list[-1])
# Com que l'últim element de la llista és senar, f contindrà la funció max
print("The type of f is: {}".format(type(f)))
print("f is: {}".format(f))
# Apliquem la funció f (és a dir, max) a la llista 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
```

Combinant la funcionalitat de definir funcions dins de funcions i la de retornar funcions, es poden crear construccions complexes que permeten resoldre elegantment alguns problemes. Si esteu interessats en aquest tipus de construccions, podeu consultar l'enllaç següent (lectura opcional, per a ampliar coneixements).

2.3 1.3.- Paràmetres

Les funcions poden tenir paràmetres, que serveixen per proveir-les de dades que necessiten per a la seva execució. Formalment, distingim entre paràmetres i arguments. Els paràmetres són part de la signatura de la funció, mentre que els arguments són els valors que rep la funció en el moment d'executar-la. Per exemple, a la celúla de codi on definíem la funció `sum_and_mult_two_values`, `x` i `y` són els paràmetres de la funció `sum_and_mult_two_values`, i quan fem la crida `sum_and_mult_two_values(5, 11)`, `5` i `11` són els arguments.

Informalment, però, sovint es fan servir els dos termes indistintament.

2.3.1 1.3.1.- Pas per referència d'objecte

Fins ara hem vist com les funcions reben uns arguments i retornen uns valors. Què passa, però, quan modifiquem els valors de les variables que rebem com a arguments en una crida d'una funció? Observem-ho amb alguns exemples!

[23]:
```
# Definim una funció que rep un paràmetre 'i' i
# assigna el valor 5 a aquest
def reassign_num(i):
    i = 5
    print("Value of i in the function:\t{}".format(i))


# Definim una funció que rep un paràmetre 'li' i
# assigna el valor [1., 2] a aquest
def reassign_list(li):
    li = [1, 2]
    print("Value of li in the function:\t{}".format(li))


# Definim una funció que rep un paràmetre 'li' i
# li afegeix un enter 1 amb append
def append_val(li):
    li.append(1)
    print("Value of li in the function:\t{}".format(li))
```

Hi ha diversos detalls a tenir en compte en la definició d'aquestes tres funcions. En primer lloc, les funcions no tenen cap instrucció `return`, de manera que la seva crida retornarà sempre `None`. En segon lloc, les dues primeres funcions (`reassign_num` i `reassign_list`) poden rebre un valor de qualsevol tipus com a paràmetre. En canvi, la tercera funció (`append_val`) ha de rebre un paràmetre d'un tipus que implementi el mètode `append`, com ara una llista. Per últim, és interessant notar què fan aquestes funcions: les dues primeres (`reassign_num` i `reassign_list`) **assignen** un nou valor a la variable que reben com a paràmetre (un enter en el cas de `reassign_num`, i una llista per a `reassign_list`). Contràriament, la tercera funció (`append_val`) el que fa és **modificar** la variable que rep com a paràmetre, en aquest cas, afegint un element a la llista.

Fixem-nos, a continuació, amb l'efecte que té l'execució d'aquestes funcions sobre les variables que es reben.

[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 aquests dos primers casos, la variable que es passa com a paràmetre en les crides a les funcions `reassign_num` i `reassign_list` no es veu afectada per la reassignació que es realitza a dins de les funcions: després d'executar `reassign_num`, la variable `an_integer` segueix contenint el valor `42`; i després d'executar la funció `reassign_list`, la variable `a_list` segueix valent `[42]`. Fixem-nos ara en el comportament de la variable `a_list` quan cridem la funció `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 aquest cas, i a diferència de les dues crides anteriors, el valor de la variable `a_list` es modifica (tot i que la funció no ha retornat cap valor). Això succeeix per la implementació interna que fa Python del pas de paràmetres en les crides a les funcions. Aquest comportament es coneix com a pas per referència d'objecte (en anglès, *pass-by-object-reference*), i no és igual en tots els llenguatges de programació. El lector interessat pot consultar aquest post per aprofundir en el funcionament del pas de paràmetres en Python (lectura opcional, per a ampliar coneixements). És important tenir en compte aquest comportament de les funcions en Python a l'hora de programar ja que, en cas de no fer-ho, podem obtenir resultats inesperats en les crides a funcions.

2.3.2 1.3.2.- Arguments opcionals

La manera més directa d'especificar arguments opcionals en la definició d'una funció és donar-los un valor per defecte. Aleshores, si en el moment de fer la crida l'argument no s'especifica, aquest prendrà el valor per defecte que s'hagi especificat en la definició de la funció:

[27]:
```
# Definim una funció amb dos paràmetres obligatoris i un
# d'opcional. El paràmetre opcional z serà 0 si no s'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]:
```
# Cridem la funció amb només els paràmetres obligatoris, x i y
print("(1, 3): {}".format(sum_two_or_three_values(1, 3)))

# Cridem la funció amb els paràmetres obligatoris i l'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 cridem la funció amb 1 únic paràmetre es genera un error
# ja que hi ha dos paràmetres obligatoris
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'
```

Les crides anteriors especifiquen els paràmetres per **posició**. En aquest cas, direm que fem servir arguments posicionals. També podem fer crides a les funcions especificant els arguments pel seu **nom** (en anglès, en direm *keyword arguments*), independentment de si aquests són obligatoris o opcionals:

[30]:
```
# Especifiquem tots els paràmetres per nom
print("(x=1, y=3, z=4): {}".format(
    sum_two_or_three_values(x=1, y=3, z=4)))
```

```
# Especifiquem dos paràmetres per posició i un per nom
print("(1, 3, z=4): {}".format(
    sum_two_or_three_values(1, 3, z=4)))

# Especifiquem tots els paràmetres per nom, canviant l'ordre
# dels paràmetres
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
```

Noteu com, quan especifiquem els paràmetres pel seu nom, podem incloure'ls en l'ordre què vulguem. En canvi, si especifiquem els paràmetres per posició, la posició del paràmetre en determinarà la seva assignació.

2.3.3 1.3.3.- Número indeterminat d'arguments

Python també permet definir funcions que acceptin un número arbitrari (indefinit en el moment de la definició de la funció) d'arguments. En aquest cas, es defineix un paràmetre especial anteposant * o bé ** al nom del paràmetre, i aquest paràmetre rebrà tots els arguments que no coincideixin amb cap dels paràmetres definits explícitament de la funció:

- Si s'anteposa * al nom del paràmetre, aquest serà una tupla amb tots els valors dels arguments no definits explícitament.
- Si s'anteposa ** al nom del paràmetre, aquest rebrà un diccionari amb els parells de nom i valor dels arguments no definits explícitament (que s'hauran d'especificar per nom en el moment de fer la crida a la funció).

[31]:
```
# Definim una funció amb dos paràmetres obligatoris i
# un número indeterminat de paràmetres posicionals
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]:
```
# Cridem la funció només amb els arguments obligatoris
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]:
```
# Cridem la funció amb 5 arguments posicionals
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]:
```
# Definim una funció amb dos paràmetres obligatoris i
# un número indeterminat de paràmetres amb nom
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]:
```
# Cridem la funció només amb els arguments obligatoris
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]:
```
# Cridem la funció amb els arguments obligatoris i un argument 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]:
```
# Cridem la funció amb els arguments obligatoris i tres arguments opcionals
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
```

És interessant notar que si hem especificat que els arguments opcionals seran posicionals (fent servir *), aleshores no podem cridar la funció assignant noms als arguments. De manera anàloga, si hem especificat arguments opcionals amb nom (fent servir **), no podrem cridar la funció amb arguments opcionals posicionals:

[38]:
```
try:
    # Especifiquem un argument opcional per nom en una funció
    # definida amb *
    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:
    # Especifiquem un argument opcional per posició en una funció
    # definida amb **
    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.- Funcions com a paràmetres

De la mateixa manera que una funció pot rebre com a paràmetres, per exemple, un enter o una cadena de caràcters, una funció pot rebre com a paràmetre una altra funció. Això ens serà útil en certes construccions que veurem en aquest mòdul.

Per exemple, la funció `eval_and_print`, definida a la propera celůla, rep com a paràmetres una funció i una llista, i el que fa és avaluar la funció sobre la llista, mostrar el resultat per pantalla, i retornar-lo:

[40]:
```
# La funció eval_and_print rep una altra funció com a paràmetre (la funció f)
def eval_and_print(f, x):
    # Mostrem f i el tipus de f
    print("f is {} and its type is {}".format(f, type(f)))
    # Apliquem la funció f sobre el valor x
    f_x = f(x)
    # Mostrem el valor per pantalla i el retornem
    print("The result of f(x) is: {}".format(f_x))
    return f_x


# Cridem a eval_and_print amb funcions diferents
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.- Introducció a la programació funcional

La programació funcional és un paradigma de programació basat en l'avaluació de funcions matemàtiques, en el qual la sortida d'una funció depèn únicament de la seva entrada, i no de l'estat del programa.

En aquesta secció, presentarem tres funcions de Python que són útils a l'hora de programar amb el paradigma de programació funcional: `map`, `filter` i `reduce`.

2.4.1 1.4.1.- Map

La funció `map` rep com a paràmetres una funció i un iterable, i retorna un iterador que aplica la funció proporcionada a cadascun dels elements de l'iterable.

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


# Apliquem la funció sum_2 a cadascun dels elements de la llista a_list
a_list = [1, 2, 3, 4]
r = map(sum_2, a_list)
print(list(r))
```

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

El resultat d'aplicar el `map` sobre la llista és un iterable que conté el resultat de cridar la funció `sum_2` sobre cadascun dels elements de la llista original, `a_list`. Això seria equivalent a executar la *list comprehension* següent:

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

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

[43]:
```
# Definim una funció que converteix cadenes de caràcters que expressen
# valors numèrics (possiblement decimals) en enters
def convert_to_int(x):
    """Convert strings to int."""
    return int(float(x))
```

```
# Apliquem la funció convert_to_int a cadascun dels elements de la
# llista a_list
a_list = ["42.45", "13.4", "12000"]
list(map(convert_to_int, a_list))
```

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

Als exemples anteriors, la funció que es passa com a argument a `map` (`sum_2` i `convert_to_int`) té un únic paràmetre, de manera que `map` la pot aplicar directament sobre cadascun dels elements de la llista (llista que rep com a segon argument). Ara bé, la funció que es passa com a paràmetre a `map` pot tenir més d'un paràmetre. En aquest cas, la funció `map` rebrà tantes llistes com paràmetres necessiti la funció que rep.

En l'exemple següent, definim una funció que calcula el preu d'un pis a partir dels metres quadrats que té (`sqm`), l'estat de conservació (`status`) i el veïnat en què es troba (`neigh`):

[44]:
```
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

    # Preu base és metres quadrats per preu per metre quadrat
    price = sqm * price_per_sqm

    # Si el pis es troba en un barri considerat bo, s'aplica un factor
    # multiplicatiu al preu del pis
    if neigh in nice_neigh:
        price *= nice_neigh_factor

    # Si el pis és nou, s'aplica un factor multiplicatiu al
    # preu del pis
    if status == "New":
        price *= new_factor

    return price
```

[45]:
```
# Calculem el preu d'un pis nou de 100m^2 que es troba en un barri
# catalogat com a "A"
compute_price(100, "New", "A")
```

[45]:
```
250000.0
```

Una immobiliària té 5 pisos a la seva disposició, i vol fer servir la funció `compute_price` per calcular el preu que hauria de tenir cadascun dels pisos. Podem fer servir la funció `map` per calcular-ho, partint de tres llistes que indiquin els metres quadrats, els estats i els barris dels pisos:

[46]:
```
# Definim tres llistes amb les dades dels pisos
sqms = [100, 120, 125, 190, 200]
statuses = ["New", "New", "Used", "Unknown", "Used"]
neigs = ["A", "B", "B", "D", "A"]

# Apliquem la funció compute_price sobre cadascun dels 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 funció `filter` rep també com a paràmetres una funció i un iterable, i retorna un iterador que recorre els elements de l'iterable tals que l'avaluació de la funció és True.

[47]:
```
# Definim la funció is numeric, que retorna True si rep un enter o un float
# com a argument, i False en cas contrari
def is_numeric(x):
    if type(x) == float or type(x) == int:
        return True
    return False
```

[48]:
```
# Definim una llista que conté enters, reals, cadenes, llistes i un valor None
a_list_with_str_and_nums = [1, 2, "three", "four", 5, 5.5, 6, [0, 0, 1], None]

# Apliquem filter sobre la llista amb la funció 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 funció `reduce` rep, de nou, una funció i un iterable. La funció aplica, de manera acumulativa, la funció que rep com a argument als elements de l'iterable. La funció que rep com a argument ha de ser una funció que rebi dos paràmetres i retorni un únic valor. Aleshores, `reduce` aplica la funció als dos primers valors de l'iterable, després al tercer valor i al resultat de l'operació anterior, etc. Per exemple, sigui 'f' la funció a aplicar i [1, 2, 3, 4] l'iterable, `reduce` calcularia:

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

Veiem alguns exemples:

[49]:
```
# Importem reduce del mòdul functools
from functools import reduce

# Calculem la suma dels valors d'una llista
a_list = [1, 2, 3, 4]
reduce(sum_two_values, a_list)
```

[49]:
```
10
```

Recordeu que la funció `sum_two_values` rebia dos paràmetres i retornava un sol valor, corresponents a la suma dels paràmetres. Així, a l'aplicar-la amb `reduce` sobre una llista, el que obtenim és la suma dels valors de la llista: s'aplica la funció `sum_two_values` als dos primers valors, 1 i 2, el que dóna com a resultat 3; es torna a aplicar la funció amb 3 i 3 com a arguments (el resultat de la primera crida a `sum_two_values` i el tercer valor de la llista), resultant amb 6; i finalment es crida la funció amb 6 i 4 (el resultat de l'última crida a la funció i el valor de l'últim element de la llista); el resultat final és doncs 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 anàloga, podem fer servir `reduce` per calcular el valor màxim d'una llista, definint primer una funció que retorni el màxim de dos valors, i aplicant-la sobre una llista amb `reduce`.

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


# Apliquem reduce amb max_two sobre una llista per calcular-ne el màxim
a_list = [10, 1, 15, 19, 30]
reduce(max_two, a_list)
```

[50]:
```
30
```

En aquest cas, l'execució del `reduce` seria la següent:

```
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
```

Veiem un tercer exemple d'ús de `reduce` que ens permet 'aplanar' llistes, és a dir, donada una llista de llistes, obtenir una llista que contingui els elements de les llistes interiors:

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


# Cridem join_lists amb dues llistes, per veure'n el seu efecte
join_lists([1, 2], [3, 4])
```

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

[52]:
```
# Fem servir join_lists amb reduce per aplanar una llista
list_to_flatten = [[0], [1, 2, 3], [5]]
reduce(join_lists, list_to_flatten)
```

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

Opcionalment, la funció `reduce` també pot rebre un tercer paràmetre, que correspon al valor inicial. Si hi és, aleshores la primera crida a la funció es fa amb el valor inicial i el primer valor de l'iterable, en comptes de fer servir els dos primers valors de l'iterable.

[53]:
```
# Fem servir reduce amb sum_two_values per sumar una llista,
# indicant 0 com a valor inicial per assegurar la correctesa del resultat
a_list = [1, 2, 3, 4]
reduce(sum_two_values, a_list, 0)
```

[53]:
```
10
```

[54]:
```
# Fem servir el valor inicial de reduce per afegir 15 al sumatori d'una llista
reduce(sum_two_values, a_list, 15)
```

[54]:
```
25
```

[55]:
```
# Fem servir reduce amb max_two per calcular el màxim d'una llista
# indicant 0 com a valor inicial, de manera que obtindrem el màxim de la
# llista si hi ha algun valor positiu o bé 0 si tots els números de
# la llista són negatius
print(reduce(max_two, [10, 1, 15, 19, 30], 0))
print(reduce(max_two, [-5, -3, -2], 0))
```

```
30
0
```

Per acabar aquesta secció, veurem un exemple que combina les tres funcions que acabem de presentar (`reduce`, `map`, i `filter`). Suposem que volem definir una funció que rebi una llista i retorni el producte dels quadrats de tots els elements senars de la llista. Combinant les tres funcions que hem vist, podem fer aquest càlcul en una sola línia:

[56]:
```
# Definim una funció que retorna el producte de dos valors
def prod_two_values(x, y):
    return x*y


# Definim una funció que retorna el quadrat d'un valor
def sq(x):
    return x**2


# Definim una funció que retorna True si un valor és senar
# i False en cas contrari
def is_odd(x):
    return x % 2
```

```
# Definim la funció que retorna el producte dels quadrats dels senars
# d'una llista
def prod_sq_if_odd(l):
    return reduce(prod_two_values, map(sq, filter(is_odd, l)))
```

[57]:
```
# Executem la funció prod_sq_if_odd per a una llista i comprovem
# el resultat 'manualment'
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.- Funcions anònimes

Les funcions anònimes són un tipus especial de funcions que no tenen nom i que es troben limitades a una sola expressió. Aquest tipus de funcions són útils quan necessitem una funció normalment simple que no voldrem reutilitzar (o bé la reutilitzarem molt poc) en el nostre codi, i es fan servir sovint en combinació amb les funcions de programació funcional que hem vist a la secció anterior.

Les funcions anònimes només poden tenir una única expressió. Aquesta expressió és avaluada amb els arguments quan es crida la funció, i el resultat d'aquesta avaluació és el valor que retorna la funció.

Per definir una funció anònima en Python es fa servir la paraula clau `lambda`, seguida dels paràmetres de la funció, uns dos punts, i l'expressió de retorn de la funció. Per aquest motiu, també fem servir l'expressió funció lambda per anomenar les funcions anònimes.

La següent celůla mostra una funció anònima que suma dos valors:

[58]:
```
# Definim una funció lambda que suma dos valors
lambda x, y: x+y
```

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

Podem cridar una funció lambda tant directament (envoltant-la de parèntesis i passant-li els arguments), o bé assignant-li un nom:

[59]:
```
# Definim una funció lambda que suma dos valors i la cridem
# amb els valors 15 i 19
(lambda x, y: x+y)(15, 19)
```

[59]:
```
34
```

[60]:
```
# Definim una funció lambda i li assignem el nom suma,
# per poder-la cridar posteriorment
suma = lambda x, y: x+y
suma(15, 19)
```

[60]:
```
34
```

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

Normalment, però, trobarem les funcions lambda dins d'altres expressions. De fet, l'exemple anterior es considera mala praxi, i s'inclou únicament per ajudar a la comprensió d'aquest tipus de funcions.

Un exemple d'una situació en la qual faríem servir funcions anònimes és l'últim exemple de la secció anterior, on hem hagut de definir tres funcions molt simples, que segurament no tornarem a fer servir en cap altra part del codi, per tal de poder fer el càlcul de `prod_sq_if_odd`. Fent servir funcions anònimes, podríem reduir la definició de la funció `prod_sq_if_odd` a una sola expressió, sense necessitat de definir amb anterioritat les tres funcions auxiliars com hem fet abans:

[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 aquest cas, en comptes de definir les funcions `prod_two_values`, `sq` i `is_odd`, hem fet servir funcions anònimes equivalents, de manera que la definició de la funció `prod_sq_if_odd_with_lamb` és molt més compacta.

A continuació, veurem com reescriure alguns dels altres exemples de programació funcional que hem vist en aquest notebook fent servir funcions anònimes:

[62]:
```
# Sumem 2 a cadascun dels elements de la llista a_list
# fent servir una funció anònima
a_list = [1, 2, 3, 4]
r = map(lambda x: x + 2, a_list)
print(list(r))
```

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

[63]:
```
# Convertim les cadenes de caràcters que expressen valors numèrics
# (possiblement decimals) d'una llista en enters fent servir
# una funció anònima
a_list = ["42.45", "13.4", "12000"]
list(map(lambda x: int(float(x)), a_list))
```

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

[64]:
```
# Filtrem d'una llista els valors numèrics
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]:
```
# Calculem el màxim d'una llista amb reduce i una funció anònima
# que retorna el màxim de dos valors
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

Els *docstrings* són cadenes de text que contenen la documentació de les funcions, entre d'altres objectes, en Python. Ja hem anat fent servir *docstring* per documentar algunes de les funcions d'aquest notebook. A continuació en repassarem alguns detalls sobre el seu funcionament i sobre les convencions que es fan servir a l'hora d'escriure'ls.

Podem accedir al *docstring* a través de la funció `help` o, directament, a través de l'atribut `doc`.

[66]:
```
# Accedim al docstring de la funció built-in max amb 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]:
```
# Accedim al docstring de la funció built-in max a través de l'atribut 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.'
```

Les convencions sobre l'ús de *docstring* en Python es troben descrites al PEP-257. Els *docstrings* es defineixen com a primera sentència dins de la definició de la funció i distingim entre *docstrings* d'una única línia i *docstrings* multilínia.

La convenció en la definició de *docstrings* d'una sola línia és de fer servir triples cometes dobles per indicar el comentari (encara que no sigui necessari ja que el comentari ocupi únicament una línia), escriure les cometes d'inici i tancament a la mateixa línia del text, començar la frase del comentari en majúscules i acabar en un punt, i no incloure salt de línia ni abans ni després del comentari.

Així, per exemple, aquest seria un *docstring* d'una sola línia que segueix les convencions:

[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, els *docstrings* multilínia es delimiten també amb triples cometes dobles. En aquest cas, consisteixen d'una línia de resum, una línia en blanc, i després una descripció més completa, que es pot estendre diverses línies.

El ***docstring* d'una funció hauria de resumir el seu comportament**, i llistar els seus arguments, valor de retorn, les excepcions que pot llançar, els efectes colůlaterals que pot tenir la seva crida, i les restriccions a tenir en compte quan la cridem.

[70]:
```python
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]:
```python
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
```

Hi ha diferents formats per escriure el *docstring*, que detallen com s'han d'especificar els arguments, els valors de retorn, les excepcions, etc. Us recomanem llegir el post d'stackoverflow sobre els formats de dosctring the Python per veure'n alguns exemples (lectura opcional, per a ampliar coneixements).

Per acabar, és important notar que no només les funcions fan ús de *docstrings* per documentar-ne el seu comportament. Els mòduls, les classes, i els mètodes també es documenten amb *docstring*.

[72]:
```
# Accedim al docstring de la classe 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.- Exercicis per practicar

A continuació hi trobareu un conjunt de problemes que us poden servir per a practicar els conceptes explicats en aquesta unitat. Us recomanem que intenteu fer aquests problemes vosaltres mateixos i que, una vegada realitzats, compareu la solució que us proposem amb la vostra solució. No dubteu en adreçar tots els dubtes que sorgeixin de la resolució d'aquests exercicis o bé de les solucions proposades al fòrum de l'aula.

1. Definiu una funció que rebi com a paràmetres dos valors (`x` i `y`) que seran dos vectors d'enters, i retorni la distància euclidiana entre els punts representats per els vectors. Cal que el cos de la funció contingui una única expressió, que calculi i retorni el resultat. Els vectors poden tenir una mida arbitrària, però tots dos vectors tindran el mateix número d'elements. Només es poden fer servir funcions de la llibreria estàndard de Python.

[73]:
```
# Resposta
```

2. Definiu una funció que rebi com a paràmetres dos valors (`x` i `y`) que seran dos vectors d'enters, i retorni la distància de Manhattan entre els punts representats per els vectors. Cal que el cos de la funció contingui una única expressió, que calculi i retorni el resultat. Els vectors poden tenir una mida arbitrària, però tots dos vectors tindran el mateix número d'elements. Només es poden fer servir funcions de la llibreria estàndard de Python.

[74]:
```
# Resposta
```

3. Definiu una funció `compute_all_distances` que rebi com a paràmetres dos valors (`x` i `y`) que seran dos vectors d'enters, i retorni una tupla de dos elements, amb les distàncies euclidiana i de Manhattan entre els punts representats pels vectors.

Per fer-ho, encapsuleu el codi de les funcions de les activitats anteriors a dins de la funció `compute_all_distances`.

[75]:
```
# Resposta
```

4. A la fruiteria del barri tenen un problema que requereix la nostra ajuda. Reiteradament, se'ls trenquen les prestatgeries on hi posen les taronges, i volen evitar que això torni a passar. Han calculat que els prestatges de fusta suporten sense problemes un pes de 50 kilos, i els de plàstic 30 kilos, però sempre dubten de si poden afegir-hi algun pis de taronges més (ja que això sempre llueix més davant dels clients).

Les taronges es troben apilades en una piràmide de base quadrada. Així doncs, a dalt de tot hi ha una sola taronja, al segon pis n'hi ha 4, al tercer pis n'hi ha 9, etc. Els pisos sempre estan complets.

Definiu una funció que rebi com a paràmetres el número de pisos de taronges que volen fer, el pes mitjà de cada taronja, i el tipus de material del prestatge, i retorni un booleà indicant si el prestatge aguantarà el pes o no. La funció sempre rebrà el número de pisos de taronges, però els paràmetres de pes mitjà i material són opcionals, i prendran un valor per defecte de 0.2 i fusta (`"Wood"`), respectivament.

[76]:
```
# Resposta
```

5. Definiu una funció que pugui rebre un número de paràmetres qualsevol, superior a 1, i que retorni el resultat de sumar el resultat d'aplicar la funció que rep com a primer paràmetre a cadascun dels altres paràmetres.

Per exemple, si la funció rep com a primer argument una funció que calcula quadrats, com a segon argument un 5, i com tercer argument un 10, hauria de retornar $5^2 + 10^2 = 125$.

Crideu a la funció definida amb els valors de l'exemple esmentat a l'enunciat, i comproveu que obteniu el resultat correcte. Comproveu també que la funció retorna els resultats esperats per a una crida amb 2 arguments i amb 5 arguments.

[77]:
```
# Resposta
```

6. Definiu una funció que rebi dos paràmetres, una llista d'enters i un enter, i retorni una llista amb els mateixos elements que la llista original eliminant-ne totes les aparicions de l'enter especificat.

6.1. Implementeu una funció que modifiqui la llista original. Feu una crida a la funció definida i mostreu que, efectivament, la llista original es modifica.

[78]:
```
# Resposta
```

6.2. Implementeu una funció que **no** modifiqui la llista original. Feu una crida a la funció definida i mostreu que, efectivament, no es modifiqui la llista original.

[79]:
```
# Resposta
```

3.1 2.1.- Solucions als exercicis per practicar

1. Definiu una funció que rebi com a paràmetres dos valors (`x` i `y`) que seran dos vectors d'enters, i retorni la distància euclidiana entre els punts representats per els vectors. Cal que el cos de la funció contingui una única expressió, que calculi i retorni el resultat. Els vectors poden tenir una mida arbitrària, però tots dos vectors tindran el mateix número d'elements. Només es poden fer servir funcions de la llibreria estàndard de Python.

[80]:
```
# Resposta

# Importem math, que està a la llibreria estàndard i permet
# calcular arrels quadrades
import math
```

```
def eucl_dist(x, y):
    # Definim la funció, que calcula la distància fent servir una list
    # comprehension unint els valors dels vectors x i y amb zip
    return math.sqrt(sum([(coord[0]-coord[1])**2 for coord in zip(x, y)]))
```

2. Definiu una funció que rebi com a paràmetres dos valors (`x` i `y`) que seran dos vectors d'enters, i retorni la distància de Manhattan entre els punts representats per els vectors. Cal que el cos de la funció contingui una única expressió, que calculi i retorni el resultat. Els vectors poden tenir una mida arbitrària, però tots dos vectors tindran el mateix número d'elements. Només es poden fer servir funcions de la llibreria estàndard de Python.

[81]:
```
# Resposta


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

3. Definiu una funció `compute_all_distances` que rebi com a paràmetres dos valors (`x` i `y`) que seran dos vectors d'enters, i retorni una tupla de dos elements, amb les distàncies euclidiana i de Manhattan entre els punts representats pels vectors.

Per fer-ho, encapsuleu el codi de les funcions de les activitats anteriors a dins de la funció `compute_all_distances`.

[82]:
```
# Resposta


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. A la fruiteria del barri tenen un problema que requereix la nostra ajuda. Reiteradament, se'ls trenquen les prestatgeries on hi posen les taronges, i volen evitar que això torni a passar. Han calculat que els prestatges de fusta suporten sense problemes un pes de 50 kilos, i els de plàstic 30 kilos, però sempre dubten de si poden afegir-hi algun pis de taronges més (ja que això sempre llueix més davant dels clients).

Les taronges es troben apilades en una piràmide de base quadrada. Així doncs, a dalt de tot hi ha una sola taronja, al segon pis n'hi ha 4, al tercer pis n'hi ha 9, etc. Els pisos sempre estan complets.

Definiu una funció que rebi com a paràmetres el número de pisos de taronges que volen fer, el pes mitjà de cada taronja, i el tipus de material del prestatge, i retorni un booleà indicant si el prestatge aguantarà el pes o no. La funció sempre rebrà el número de pisos de taronges, però els paràmetres de pes mitjà i material són opcionals, i prendran un valor per defecte de 0.2 i fusta (`"Wood"`), respectivament.

[83]:
```
# Resposta


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. Definiu una funció que pugui rebre un número de paràmetres qualsevol, superior a 1, i que retorni el resultat de sumar el resultat d'aplicar la funció que rep com a primer paràmetre a cadascun dels altres paràmetres.

Per exemple, si la funció rep com a primer argument una funció que calcula quadrats, com a segon argument un 5, i com tercer argument un 10, hauria de retornar $5^2 + 10^2 = 125$.

Crideu a la funció definida amb els valors de l'exemple esmentat a l'enunciat, i comproveu que obteniu el resultat correcte. Comproveu també que la funció retorna els resultats esperats per a una crida amb 2 arguments i amb 5 arguments.

[85]:
```
# Resposta


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. Definiu una funció que rebi dos paràmetres, una llista d'enters i un enter, i retorni una llista amb els mateixos elements que la llista original eliminant-ne totes les aparicions de l'enter especificat.

6.1. Implementeu una funció que modifiqui la llista original. Feu una crida a la funció definida i mostreu que, efectivament, la llista original es modifica.

[89]:
```
# Resposta


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. Implementeu una funció que **no** modifiqui la llista original. Feu una crida a la funció definida i mostreu que, efectivament, no es modifiqui la llista original.

[90]:
```
# Resposta


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.- Bibliografia

4.1 3.1.- Bibliografia bàsica

Us recomanem revisar la documentació oficial de les funcions i classes descrites en aquesta unitat, que trobareu enllaçades en cada un dels apartats que les descriuen.

4.2 3.2.- Bibliografia addicional - Ampliació de coneixements

Per aprofundir en el funcionament del pas de paràmetres en funcions en Python, us recomanem consultar aquest post de medium

Al notebook hem vist com definir funcions dins de funcions i les conseqüències que això té sobre la visibilitat de les variables. També hem comentat una de les utilitats d'aquesta definició, que és la d'encapsular codi. Una altra de les utilitats és la definició de *closures*. Si voleu conèixer aquesta construcció, us recomanem la lectura de l'article Python nested functions.

Com hem comentat, hi ha diferents formats per escriure el *docstring*. Us recomanem llegir el post d'stackoverflow sobre els formats de dosctring the Python per veure'n alguns exemples.