Chapter 18: Arguments

Extractos de codigo del libro Learning Python 5th Ed. by Mark Lutz

1. Arguments and Shared References

Paso de objetos inmutables:

In [9]:
```
def f(a):           # a is assigned to (references) the passed ob
ject
        a = 99  # Changes local variable a only

b = 88
f(b)            # a and b both reference same 88 initially
print("b = ", b)        # b is not changed
```

```
b =  88
```

Paso de objetos mutables:

In [14]:
```
def changer(a, b):          # Arguments assigned references to ob
jects
                a = 2                   # Changes local name's va
lue only
                b[0] = 'spam'   # Changes shared object in place


x = 1
l = [1, 2]
changer(x, l)           # Caller: Pass immutable and mutable obje
cts
print("l = ", l)        # ['spam', 2]  # x is unchanged, l is dif
ferent!
```

```
l =  ['spam', 2]
```

Cómo evitar que una routina modifique los elementos mutables:

- Pasar una copia

In [15]:
```
l = [1, 2]
changer(x, l[:])        # Pass a copy, so our 'l' does not change

print("l = ", l)
```

```
l =  [1, 2]
```

- Utilizar invariantes (en las estructuras de datos de entrada)

In [1]:
```
# Opción 2

l = [1, 2]
def changer(a, b):
    b = b[:]            # Copy input list so we don't impact call
er
    a = 2
    b[0] = 'spam'       # Changes our list copy only

changer(x, l)
assert l == [1, 2], "l es una invariante"
print("l = ", l)
```

```
---------------------------------------------------------------------
----------
NameError                                 Traceback (most recent
call last)
<ipython-input-1-a8f928696de8> in <module>
      7     b[0] = 'spam'       # Changes our list copy only
      8
----> 9 changer(x, l)
     10 assert l == [1, 2], "l es una invariante"
     11 print("l = ", l)

NameError: name 'x' is not defined
```

- Convertir el objeto mutable a un objeto inmutable:

In [2]:
```
l = [1, 2]
def changer(a, b):
                a = 2
                b[0] = 'spam'   # TypeError: 'tuple' object does
not support item assignment

changer(x, tuple(l))    # Pass a tuple, so changes are errors

assert l == [1, 2], "l es una invariante"
print("l = ", l)
```

```
---------------------------------------------------------------------
----------
NameError                                 Traceback (most recent
call last)
<ipython-input-2-dcc851c82251> in <module>
      4                 b[0] = 'spam'   # TypeError: 'tuple' obje
ct does not support item assignment
      5
----> 6 changer(x, tuple(l))    # Pass a tuple, so changes are er
rors
      7
      8 assert l == [1, 2], "l es una invariante"

NameError: name 'x' is not defined
```

2 Special Argument-Matching Modes

ver figura 06 en Drive

Be careful not to confuse the special `name=value` syntax in a function header and a function call; in the call it means a match-by-name keyword argument, while in the header it specifies a default for an optional argument.

Keyword and Default Examples

1. Comportamiento por defecto:

In [3]:
```
def f(a, b, c):
    print(a, b, c)

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

```
1 2 3
```

2. Keywords

Left-to-right order of the arguments no longer matters when keywords are used because arguments are matched by name, not by position.

In [4]:
```
def f(a, b, c):
    print(a, b, c)

f(c=3, b=2, a=1)    # 1 2 3
```

```
1 2 3
```

In [5]:
```
f(1, c=3, b=2)          # a gets 1 by position, b and c passed by
name
                    # 1 2 3

# they make your calls a bit more self documenting
# f(name='Bob', age=40, job='dev')
```

```
1 2 3
```

3. Defaults

Defaults allow us to make selected function arguments optional; if not passed a value, the argument is assigned its default before the function runs.

In [6]:
```
def f(a, b=2, c=3):     # a required, b and c optional
        print(a, b, c)


f(1)        # 1 2 3
f(a=1)      # 1 2 3
```

```
1 2 3
1 2 3
```

If we pass two values, only `c` gets its default, and with three values, no defaults are used:

In [7]:
```
def f(a, b=2, c=3):     # a required, b and c optional
        print(a, b, c)

f(1, 4)     # 1 4 3
f(1, 4, 5)      # Override defaults: 1 4 5

f(1, c=6)       # Choose defaults: 1 2 6
```

```
1 4 3
1 4 5
1 2 6
```

4. Combining keywords and defaults

Notice again that when keyword arguments are used in the call, the order in which the arguments are listed doesn't matter; **Python matches by name, not by position**.

The caller must supply values for `spam` and `eggs`, but they can be matched by position or by name.

In [8]:
```
def func(spam, eggs, toast=0, ham=0):
        print((spam, eggs, toast, ham))         # First 2 require
d

func(1, 2)                                                      #
Output: (1, 2, 0, 0)
func(1, ham=1, eggs=0)                          # Output: (1, 0,
0, 1)
func(spam=1, eggs=0)                            # Output: (1, 0,
0, 0)
func(toast=1, eggs=2, spam=3)           # Output: (3, 2, 1, 0)
func(1, 2, 3, 4)                                        # Output:
(1, 2, 3, 4)
```

```
(1, 2, 0, 0)
(1, 0, 0, 1)
(1, 0, 0, 0)
(3, 2, 1, 0)
(1, 2, 3, 4)
```

5. Python 3.X Keyword-Only Arguments

We can also use a * character by itself in the arguments list to indicate that a function does not accept a variable-length argument list but still expects all arguments following the * to be passed as keywords.

In [9]:
```
def kwonly(a, *b, c):
    print(a, b, c)

kwonly(1, 2, c=3)       # 1 (2,) 3

kwonly(a=1, c=3)        # 1 () 3

kwonly(1, 2, 3)         # TypeError: kwonly() missing 1 required
keyword-only argument: 'c'
```

```
1 (2,) 3
1 () 3

---------------------------------------------------------------------
----------
TypeError                                 Traceback (most recent
call last)
<ipython-input-9-f9d62b949278> in <module>
      6 kwonly(a=1, c=3)        # 1 () 3
      7
----> 8 kwonly(1, 2, 3)         # TypeError: kwonly() missing 1 r
equired keyword-only argument: 'c'

TypeError: kwonly() missing 1 required keyword-only argument: 'c'
```

You can still use defaults for keyword-only arguments, even though they appear after the * in the function header:

In [ ]:
```
def kwonly(a, *, b='spam', c='ham'):
        print(a, b, c)

kwonly(1)               # 1 spam ham
kwonly(1, c=3)          # 1 spam 3
kwonly(a=1)             # 1 spam ham
kwonly(c=3, b=2, a=1)   # 1 2 3
kwonly(1, 2)            # TypeError: kwonly() takes 1 positional
argument but 2 were given
```

In [ ]:
```
def kwonly(a, *, b, c='spam'):
    print(a, b, c)

kwonly(1, c='eggs')     # TypeError: kwonly() missing 1 required
keyword-only argument: 'b'
```