Unpacking, Decoratori, Generatori, Eccezioni

Unpacking degli argomenti

Gli elementi di una tupla o lista possono essere **spacchettati** con l'utilizzo dell'operatore `*`

In [1]:
```
l = [1, 2, 3]

def f(a, b, c):
    # stampa 3 argomenti
    print(a)
    print(b)
    print(c)

# chiama funzione "spacchettando" la lista negli argomenti
f(*l)
```

```
1
2
3
```

In [2]:
```
# spacchetta la tupla in variabili
a, b, c = (1,2,3)
print(a)
print(b)
print(c)
```

```
1
2
3
```

In [3]:
```
# spacchetta la tupla, ma non ci sono abbastanza variabili
a,b = (1,2,3)
print(a)
print(b)
print(c)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [3], in <cell line: 2>()
      1 # spacchetta la tupla, ma non ci sono abbastanza variabili
----> 2 a,b = (1,2,3)
      3 print(a)
      4 print(b)

ValueError: too many values to unpack (expected 2)
```

In [4]:
```
# spacchetta la tupla ma ci sono troppe variabili
a,b,c,d = (1,2,3)
print(a)
print(b)
print(c)
print(d)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [4], in <cell line: 2>()
      1 # spacchetta la tupla ma ci sono troppe variabili
----> 2 a,b,c,d = (1,2,3)
      3 print(a)
      4 print(b)

ValueError: not enough values to unpack (expected 4, got 3)
```

In [5]:
```
# spacchetta la tupla, ma sono interessato solo a 2 valori
a, b, _ = (1,2,3)
print(a)
print(b)
```

```
1
2
```

In [6]:
```
# usare _ è una convenzione
print(_)
```

```
3
```

In [7]:
```
# spaccchetta la tupla in un oggetto "spacchettato"?
a, *b = (1,2,3)
print(a)
print(b)
```

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

Unpacking e *split*

In [8]:
```
frase = 'Che bella questa lezione'
v = frase.split(' ')
print(v)
```

```
['Che', 'bella', 'questa', 'lezione']
```

In [9]:
```
v = frase.split(' ', 1)  # Uso max args opzione di split
print(v)
```

```
['Che', 'bella questa lezione']
```

Unpacking

In [10]:
```
# split in 2 variabli
v,v2 = frase.split(' ')
print(v)
print(v2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [10], in <cell line: 2>()
      1 # split in 2 variabli
----> 2 v,v2 = frase.split(' ')
      3 print(v)
      4 print(v2)

ValueError: too many values to unpack (expected 2)
```

In [11]:
```
# e se ..la seconda è uno "spacchettamento"?
v, *v2 = frase.split(' ')
print(v)
print(v2)
```

```
Che
['bella', 'questa', 'lezione']
```

In [12]:
```
# split in piu' variabili
v, *v2, v3 = frase.split(' ')
print(v)
print(v2)
print(v3)
```

```
Che
['bella', 'questa']
lezione
```

In [13]:
```
# se sono interessato solo al primo elemento, butto gli altri
v, *_ = frase.split(' ')
print(v)
```

```
Che
```

Unpacking di strutture chiave-valore

Nal caso di struttura chiave-valore (dizionari), l'unpacking si può fare con l'operatore `**`

In [14]:
```
d = {'a':1, 'b':2}

def f(a, b):
    print(a)
    print(b)
```

In [15]:
```
# operatore unpacking *
f(*d)
```

```
a
b
```

In [17]:
```
# operatore **
f(**d)
```

```
1
2
```

Unione di dizionari

Se possiamo fare l'*unpacking* dei dizionari, allora possiamo sfruttarlo per unirne gli elementi:

In [18]:
```
d  = {'a':1, 'b':2}
d1 = {'c':3, 'd':4}

# il dizionario "somma" è dato dalla lista di tutti gli elementi spacchettati
d_tot = {**d, **d1}

print(d_tot)
```

```
{'a': 1, 'b': 2, 'c': 3, 'd': 4}
```

Forzare argomenti tramite keyword

Nelle funzioni gli **argomenti** sono passati per **posizione**: `args`

In [19]:
```
def f(a, b, c):
    return a*b + c

print( f(3,4,5) )
print( f(5,4,3) )
```

```
17
23
```

però è possibile **assegnare** ad ogni variabile (**keyword**) il suo **valore**: `kwargs`

In [20]:
```
print( f(3,4,5) )
print( f(c=5, b=4, a=3) )
```

```
17
17
```

Se voglio **costringere** l'utilizzo della funzione **solo** tramite keyword (`kwargs`) escludendo tutti gli argomenti posizionali, usiamo l'unpacking:

In [21]:
```
# l'unpackig come primo argomento della funzione "consuma" i parametri passati alla chiamata
def f(*, a, b, c):
    return a*b + c

print( f(3,4,5) )
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Input In [21], in <cell line: 5>()
      2 def f(*, a, b, c):
      3     return a*b + c
----> 5 print( f(3,4,5) )

TypeError: f() takes 0 positional arguments but 3 were given
```

In [22]:
```
# se gli argomenti sono assegnati alle keyword (kwargs), non sono consumeti dall'unpacking
print( f(a=3, b=4, c=5) )
```

```
17
```

Decoratori

Abbiamo visto nello scrivere metodi delle classi i *decoratori* `@classmethod` e `@staticmethod`.
I *decoratori* sono delle funzioni che modificano il comportamento della funzione a cui sono applicati.
Sono utili per **estendere le funzionalità** di una funzione **Senza modificarla**.

Estendere le funzionalità

Posso scrivere una funzione che **estende le funzionalità** di un'altra funzione così:

In [23]:
```
# funzione 'base' che voglio estendere
def funzione_base() :
    print("Io sono la funzione base!")
```

```
    # funzione 'estensione': accetta come argomento la "funzione" da estendere
    def estensione(funzione) :
        # contenitore
        def wrapper_della_funzione():
            # operazioni che "estendono la funzionalità"
            print("...aggiungo delle operazioni alla funzione...")
            # ritorna le funzione con le sue operazioni
            return funzione()
        # ritorno il contenitore
        return wrapper_della_funzione
```

In [24]:
```
    # uso funzione 'base'
    print('funzione originale:')
    funzione_base()

    print('---')

    # uso l'estensione applicandola alla funzione
    print('funzione estesa:"')
    funzione_estesa = estensione(funzione_base)
    funzione_estesa()
```

```
funzione originale:
Io sono la funzione base!
---
funzione estesa:"
...aggiungo delle operazioni alla funzione...
Io sono la funzione base!
```

Il *decoratore*

La stessa operazione può essere fatta con un decoratore che usa la funzione estensione. Lo scopo è quello di potermi concentrare solo sulla scrittura della "funzione base" senza preoccuparmi delle operazioni che farà l'estensione

In [25]:
```
    # definisco la funzione estensione che usero' come decoratore
    def estensione(funzione):
        # contenitore
        def wrapper_della_funzione():
            # operazioni
            print("...aggiungo delle operazioni alla funzione...")
            # ritorno funzione
            return funzione()
        #ritorno contenitore
        return wrapper_della_funzione


    # funzione 'base'
    @estensione
    def funzione_base() :
        print("Io sono la funzione base!")



    funzione_base()
```

```
...aggiungo delle operazioni alla funzione...
Io sono la funzione base!
```

Funzioni con degli argomenti

In [27]:
```
    # funzione decoratore
    def metti_titolo(funzione):
        # la funzione contenitore deve fare l'unpacking di tutta la lista argomenti per posizione e per keyw
        def wrapper_della_funzione(*args, **kwargs):
            print("L'operazione ha risultato:")
            return funzione(*args, **kwargs)
        return wrapper_della_funzione
```

In [28]:
```
    # scrivo una normale funzione per fare la somma ma aggiungo il decoratore
    @metti_titolo
    def somma_numeri(a,b):
        print (a + b)

    somma_numeri(2,3)
```

```
L'operazione ha risultato:
5
```

Generatori

Un **generatore** è una funzione *speciale* che ritorna un `lazy iterator`, cioè un oggetto che puo' essere iterato (loop) come una lista.

Sono utili quando è necessario mantenere lo stato di una operazione, senza costruire un oggetto (classe) specifico, e quando non si vuole occupare memoria per costruire in anticipo l'oggetto, ma si *genera* (lazy) al momento dell'utilizzo

In [29]:
```
# Una lista di valori
valori = [1,2,3,4,5]

# list comprehension
quadrato_L = [x*x for x in valori]

# generatore
quadrato_G = (x*x for x in valori)

print(quadrato_L)
print(quadrato_G)
```

```
[1, 4, 9, 16, 25]
<generator object <genexpr> at 0x000001E7BEC976F0>
```

In [30]:
```
for v1 in quadrato_L:
    print(v1)

print('---')

for v2 in quadrato_G:
    print(v2)
```

```
1
4
9
16
25
---
1
4
9
16
25
```

In [31]:
```
# il generatore "finisce"
for v1 in quadrato_L:
    print(v1)

print('---')

for v2 in quadrato_G:
    print(v2)
```

```
1
4
9
16
25
---
```

Un generatore è una funzione spaciale. Si crea come una funzione normale ma si usa `yield` al posto del `return`.

La differenza tra le due keyword è :

- **return** indica il punto in cui la funzione ritorna un risultato **e finisce**
- **yield** indica il punto in cui la funzione ritorna un risultato **e si mette in pausa** nella attule configurazione

In [32]:
```
def generatore():

    print("primo")
    yield 101

    print("secondo")
    yield 202

    print("terzo")
    yield 303
```

```
for v in generatore():
    print(v)
```

```
primo
101
secondo
202
terzo
303
```

Per *chiamare* l'elemento successivo di un generatore, si usa la funzione *built-in* `next()'

In [33]:
```
def generatore_2():

    print("primo")
    yield 101

    print("secondo")
    yield 202

    print("terzo")
    yield 303

g = generatore()
```

In [34]:
```
# primo
next(g)
```

```
primo
```

Out[34]: 101

In [35]:
```
# secondo
next(g)
```

```
secondo
```

Out[35]: 202

In [36]:
```
# terzo
next(g)
```

```
terzo
```

Out[36]: 303

In [37]:
```
# quarto
next(g)
```

```
---------------------------------------------------------------------------
StopIteration                             Traceback (most recent call last)
Input In [37], in <cell line: 2>()
      1 # quarto
----> 2 next(g)
```

StopIteration:

I generatori sono funzioni particolari che possono essere "messe in pausa" nel punto in cui l'interprete incontra la keyword `yield`

La differenza principale con le funzoni, infatti, è che la keyword `return` restituisce un risultato e **termina** la funzione, mentre `yield` retituisce un risultato e **mette in pausa** la funzione salvando tutto il suo attuale stato e permette di continuare le operazioni alla prossima chiamata

Dopo il return non ci possono essere più istruzioni, mentre dopo lo yield si

I generatori, quando sono chiamati, ritornano un **iteratore** ma non cominciano immediatamente l'esecuzione delle istruzioni: l'esecuzione delle operazini dai metodi *dunder* `__next__` e `__iter__` che sono attivati durante un loop oppure alla chiamate della funzione `next()`

Quando il generatore *termina*, si genera automaticamente un *errore runtime* di tipo `StopIteration` che indica la fine dell'iterazione nelle chamate successive (cioè il loop *for* termina)

Eccezioni e come gestirle

Abbiamo visto che le *eccezioni* sono degli **errori di runtime** che succedono mentre il programma è in esecuzione.

In Pyhton, non sempre una eccezione indica che qualcosa è andato storto e che il programma non funziona. Ad esempio, come abbiamo visto per i generatori, ci sono delle eccezioni come `StopIteration` che indicano il termine di una operazione e non un "errore".

Le eccezioni sono dei valori restituiti quando qualcosa di particolare o non valido accade. A differenze dei `return` di una funzione, le eccezioni si propagano tra tutte le chiamate di funzioni annidate finche non sono **catturate** e *gestite*. Se una eccezione non è gestita, il programma mostra un messaggio di errore e termina.

In molti casi è utile poter intercettare (**catturare**) le eccezioni e decidere come procedere con le operazioni. Questo si fa tramite il costrutto `try except`

In [38]:
```
# se faccio una operazione "particolare"

5/0
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Input In [38], in <cell line: 3>()
      1 # se faccio una operazione "particolare"
----> 3 5/0

ZeroDivisionError: division by zero
```

In [39]:
```
# se racciudo l'operazione in un blocco try-except, posso decidere cosa fare
try:
    5/0
except ZeroDivisionError:
    print("Non posso fare questa operazione")
```

```
Non posso fare questa operazione
```

In [40]:
```
# se voglio piu' informazioni
try:
    5/0
except ZeroDivisionError as testo_errore:
    print(f"Non posso fare questa operazione: {testo_errore}")
```

```
Non posso fare questa operazione: division by zero
```

E se volessi continuare le operazioni in ogni caso? `finally`

In [41]:
```
# se voglio piu' informazioni
for numero in [5, 0]:

    try:
        print(f'provo divisione {numero}')

        3/numero
        print('ok')

    except ZeroDivisionError as testo_errore:
        print(f"Non posso fare questa operazione: {testo_errore}")

    finally:
        print('continuo')
```

```
provo divisione 5
ok
continuo
provo divisione 0
Non posso fare questa operazione: division by zero
continuo
```

Diverse eccezioni

Le eccezioni sono raggruppate in diversi tipi e sotto-tipi. Il comando `except` cattura tutte le eccezioni del tipo specificato includendo tutti i sottotipi

Il tipo principale è `Exception` , attenzione a come si usa pero'!

In [42]:
```
try:
    5/0
except Exception as testo_errore :
    print(f"Non posso fare questa divisione: {testo_errore}")
```

```
Non posso fare questa divisione: division by zero
```

In [43]:
```
# ZeroDivisionError è un sottotipo di Exception, quindi
try:
    1 + 'ciao'

except Exception as testo_errore :
    print(f"Non posso fare questa divisione: {testo_errore}")
```

```
Non posso fare questa divisione: unsupported operand type(s) for +: 'int' and 'str'
```

In [44]:
```
# pero'
try:
    5/0

except ZeroDivisionError as testo_errore:
    print(f"Non posso fare questa divisione: {testo_errore}")

except Exception as testo_errore :
    print(f"Errore nel fare il conto: {testo_errore}")

print('---')

try:
    5 - 'ciao'/0

except ZeroDivisionError as testo_errore:
    print(f"Non posso fare questa divisione: {testo_errore}")

except Exception as testo_errore :
    print(f"Errore nel fare il conto: {testo_errore}")
```

```
Non posso fare questa divisione: division by zero
---
Errore nel fare il conto: unsupported operand type(s) for /: 'str' and 'int'
```

LBYL vs. EAFP

Python usa le eccezioni per indicare gli errori e gestire il flusso di lavoro.

Per questo motivo spesso si fa la distinzione sul modo in cui si scrive in python rispetto ad altri linguaggi identificando due approcci

- Look Before You Leap
- Easier Ask Forgiveness than Permission

In [46]:
```
# riempire un dizionario con le lettere
frase = ' una frase con delle lettere ripetute'

dizionario = dict()
for c in frase:
    dizionario[c] += 1

print(dizionario)
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Input In [46], in <cell line: 5>()
      4 dizionario = dict()
      5 for c in frase:
----> 6     dizionario[c] += 1
      8 print(dizionario)

KeyError: ' '
```

In [47]:
```
# LBYL
frase = ' una frase con delle lettere ripetute'

dizionario = dict()
for c in frase:
    if c in dizionario.keys():
        dizionario[c] += 1
    else:
```

```
            dizionario[c] = 1

    print(dizionario)
```

```
{' ': 6, 'u': 2, 'n': 2, 'a': 2, 'f': 1, 'r': 3, 's': 1, 'e': 8, 'c': 1, 'o': 1, 'd': 1, 'l': 3, 't': 4, 'i':
1, 'p': 1}
```

In [48]:
```
# EAFP

frase = ' una frase con delle lettere ripetute'

dizionario = dict()
for c in frase:
    try:
        dizionario[c] += 1
    except KeyError:
        dizionario[c] = 1

print(dizionario)
```

```
{' ': 6, 'u': 2, 'n': 2, 'a': 2, 'f': 1, 'r': 3, 's': 1, 'e': 8, 'c': 1, 'o': 1, 'd': 1, 'l': 3, 't': 4, 'i':
1, 'p': 1}
```

Fine

In [49]:
```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```