Laboratorio di programmazione

Python

A.A. 2020-2021

Lezione 12

Approfondimento Python 3.8

Unpacking degli argomenti

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

```
>>> l = [1, 2, 3]
>>> def f(a, b, c):
...     print(a, b, c)


>>> f(*l)
1 2 3
```

In [5]:

```
a, b, c = (1,2,3)
print(a,b,c)
```

```
1 2 3
```

In [5]:

```
a, b, c = (1,2,3)
print(a,b,c)
```

```
1 2 3
```

In [6]:

```
a,b = (1,2,3)
print(a,b,c)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-fe59f0df5c05> in <module>
----> 1 a,b = (1,2,3)
      2 print (a,b,c)

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

In [5]:

```
a, b, c = (1,2,3)
print(a,b,c)
```

```
1 2 3
```

In [6]:

```
a,b = (1,2,3)
print(a,b,c)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-fe59f0df5c05> in <module>
----> 1 a,b = (1,2,3)
      2 print (a,b,c)

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

In [8]:

```
a, b, _ = (1,2,3)
print(a,b)
```

```
1 2
```

In [5]:
```
a, b, c = (1,2,3)
print(a,b,c)
```

```
1 2 3
```

In [6]:
```
a,b = (1,2,3)
print(a,b,c)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-6-fe59f0df5c05> in <module>
----> 1 a,b = (1,2,3)
      2 print (a,b,c)

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

In [8]:
```
a, b, _ = (1,2,3)
print(a,b)
```

```
1 2
```

In [9]:
```
a, *b = (1,2,3)
print(a,b)
```

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

Unpacking e *split*

Unpacking e *split*

In [10]:
```
frase = 'Che bella questa lezione'
v = frase.split(' ')
print(v)
```

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

Unpacking e *split*

In [10]:
```
frase = 'Che bella questa lezione'
v = frase.split(' ')
print(v)
```

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

In [11]:
```
v = frase.split(' ', 1)  # Uso max args
print(v)
```

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

Unpacking

Unpacking

In [12]:
```
v,v2 = frase.split(' ')
print(v,v2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-12-24f464723456> in <module>
----> 1 v,v2 = frase.split(' ' )
      2 print(v,v2)

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

Unpacking

In [12]:
```
v,v2 = frase.split(' ')
print(v,v2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-12-24f464723456> in <module>
----> 1 v,v2 = frase.split(' ' )
      2 print(v,v2)

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

In [14]:
```
v, *v2 = frase.split(' ')
print(v, v2)
```

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

Unpacking

In [12]:
```
v,v2 = frase.split(' ')
print(v,v2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-12-24f464723456> in <module>
----> 1 v,v2 = frase.split(' ' )
      2 print(v,v2)

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

In [14]:
```
v, *v2 = frase.split(' ')
print(v, v2)
```

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

In [15]:
```
v, *v2, v3 = frase.split(' ')
print(v, v2, v3)
```

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

Unpacking

In [12]:
```
v,v2 = frase.split(' ')
print(v,v2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-12-24f464723456> in <module>
----> 1 v,v2 = frase.split(' ' )
      2 print(v,v2)

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

In [14]:
```
v, *v2 = frase.split(' ')
print(v, v2)
```

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

In [15]:
```
v, *v2, v3 = frase.split(' ')
print(v, v2, v3)
```

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

In [16]:
```
v, *_ = frase.split(' ')
print(v)
```

```
Che
```

Unpacking di strutture chiave-valore

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

Unpacking di strutture chiave-valore

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

In [4]:
```
d = {'a':1, 'b':2}
def f(a, b):
    print(a, b)
```

Unpacking di strutture chiave-valore

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

In [4]:
```
d = {'a':1, 'b':2}
def f(a, b):
    print(a, b)
```

In [5]:
```
f(*d)  # operatore *
```

```
a b
```

Unpacking di strutture chiave-valore

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

In [4]:
```
d = {'a':1, 'b':2}
def f(a, b):
    print(a, b)
```

In [5]:
```
f(*d)  # operatore *
```

```
a b
```

In [6]:
```
f(**d) # operatore **
```

```
1 2
```

Unione di dizionari

```
>>> d  = {'a':1, 'b':2}
>>> d1 = {'c':3, 'd':4}
>>> 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**

Forzare argomenti tramite keyword

Nelle funzioni gli **argomenti** sono passati per **posizione**

In [26]:
```
def f(a, b, c):
    return a*b + c

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

```
17
23
```

Forzare argomenti tramite keyword

Nelle funzioni gli **argomenti** sono passati per **posizione**

In [26]:
```
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

Forzare argomenti tramite keyword

Nelle funzioni gli **argomenti** sono passati per **posizione**

In [26]:
```
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

In [24]:
```
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 escludendo tutti gli argomenti posizionali:

Se voglio *costringere* l'utilizzo della funzione **solo** tramite keyword escludendo tutti gli argomenti posizionali:

In [27]:

```
def f(*, a, b, c):
    return a*b + c

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-1c772565de81> in <module>
      2     return a*b + c
      3
----> 4 print( f(3,4,5) )

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

Se voglio *costringere* l'utilizzo della funzione **solo** tramite keyword escludendo tutti gli argomenti posizionali:

In [27]:
```
def f(*, a, b, c):
    return a*b + c

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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-27-1c772565de81> in <module>
      2     return a*b + c
      3
----> 4 print( f(3,4,5) )

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

In [28]:
```
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 così

Estendere le funzionalità

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

In [7]:

```
# funzione 'estensione'
def estensione(funzione) :
    def wrapper_della_funzione():
        print("Il mio primo programma era:")
        return funzione()
    return wrapper_della_funzione

# funzione 'base'
def primo_programma() :
    print("Hello World!")

# uso funzione 'base'
primo_programma()

print()

# uso funzione 'estesa'
primo_programma_esteso = estensione(primo_programma)
primo_programma_esteso()
```

```
Hello World!

Il mio primo programma era:
Hello World!
```

Il *decoratore*

La stessa cosa può essere fatta con un decoratore

Il *decoratore*

La stessa cosa può essere fatta con un decoratore

In [44]:

```
# definisco il decoratore 'decoratore'
def decoratore(funzione):
    def wrapper_della_funzione():
        print("Il mio primo programma era:")
        return funzione()
    return wrapper_della_funzione

# definisco la funzione 'estesa'
@decoratore
def primo_programma() :
    print("Hello World!")


primo_programma()
```

```
Il mio primo programma era:
Hello World!
```

con degli argomenti

con degli argomenti

In [47]:

```python
# funzione decoratore
def metti_titolo(func):
    def wrap(*args, **kwargs): # unpacking di tutta la lista argomenti e keyword
        print("La somma dei due numeri è:")
        return func(*args, **kwargs)
    return wrap

# scrivo la funzione normalmente ma aggiungo il decoratore
@metti_titolo
def somma_numeri(a,b):
    print (a + b)

somma_numeri(2,3)
```

```
La somma dei due numeri è:
5
```