Table of Contents

Fondamenti di Programmazione

Andrea Sterbini

lezione 4 - 13 ottobre 2022

Ancora funzioni

```
def nome_funzione(argomenti):
    corpo della funzione
    return risultato
```

NOTA:

- l'istruzione **return** può essere ovunque nel corpo della funzione, ne ferma l'esecuzione **fornendo al programma chiamante** (tornando) il valore indicato
- se NON si esegue return la funzione torna il valore speciale **`None`**

Reminder: Le variabili locali nascono alla chiamata della funzione e scompaiono al suo completamento (return)

ATTENZIONE: gli argomenti sono **variabili locali** e potete modificarli

- se si tratta di valori **immutabili** il programma chiamante **NON se ne accorgerà**
- se si tratta di valori **mutabili e ne modificate il contenuto** il programma chiamante **se ne accorgerà**
- se invece sostituite il **riferimento** della variabile locale, il programma principale **NON se ne accorgerà**

In [10]:

```
nomi = ['Paperino', 'Topolino', 'Minnie', 'Annabella', 'Gastone']
def togli_terza(parole):
    "qui modifico la lista togliendo l'elemento a indice 2"
    parole.pop(2)                          # modifico l'oggetto riferito
def sostituisci_tutte(parole):
    "qui rimpiazzo nella variabile locale 'parole' una nuova lista"
    parole = ['uno', 'due', 'tre']    # sostituisco il RIFERIMENTO
def sostituisci_tutte_distruttiva(parole):
    "qui rimpiazzo nella variabile locale 'parole' una nuova lista"
    parole[:] = ['uno', 'due', 'tre']    # sostituisco il RIFERIMENTO

print(nomi)
togli_terza(nomi)
print(nomi)
sostituisci_tutte(nomi)
print(nomi)
sostituisci_tutte_distruttiva(nomi)
print(nomi)

```

```
['Paperino', 'Topolino', 'Minnie', 'Annabella', 'Gastone']
['Paperino', 'Topolino', 'Annabella', 'Gastone']
['Paperino', 'Topolino', 'Annabella', 'Gastone']
['uno', 'due', 'tre']
```

Ancora funzioni

Gli **argomenti formali** nella definizione della funzione possono essere

- obbligatori, posizionali, (**sempre all'inizio**)
- opzionali, con un valore di default da usare se il valore attuale non viene fornito (**sempre alla fine**)
- PRIMA gli obbligatori POI gli opzionali con i loro default

Nella chiamata mettete prima i **valori attuali obbligatori**,
poi gli **opzionali** per posizione oppure per nome

In [16]:
```python
# supponiamo di voler concatenare 3 parole
# ed una lista variabile di altre parole
def concatena(obb1, obb2, opz3=42, opz4=None):
    if not isinstance(opz3, str):   # se il 3° arg NON è una stringa
        opz3 = str(opz3)            # lo trasformo in stringa
    if opz4 is None:                # se il 4° arg non c'è
        opz4 = ['viva', 'Topolin']  # uso una lista di valori preconfezionat
    return ' '.join([obb1, obb2, opz3] + opz4)

# gli argomenti opzionali sono posizionali E ANCHE NO
concatena('Minni', 'Paperino', 'Paperoga', ['Ciccio'])
concatena('Minni', 'Paperino', 'Paperoga', opz4=['Pluto'])
concatena('Minni', 'Paperino', opz4=['Ciccio'])
concatena('Minni', 'Paperino')
# per passarli *fuori sequenza* va usato il nome
# in realtà potete anche passare tutti gli argomenti per nome
# in qualsiasi ordine
concatena(obb2='Minni', opz3='Paperino', obb1='Paperoga', opz4=['Ciccio'])

```

Out[16]:
```
'Paperoga Minni Paperino Ciccio'
```

Un errore molto difficile da notare: valori di default modificabili

Python crea i valori di default **nel momento della definizione della funzione** (e non alla sua CHIAMATA)

Per cui i valori di default **vengono riusati in tutte le chiamate**

- se sono immutabili non c'è problema
  (numeri, stringhe, None, tuple)
- se sono mutabili **NON DOVETE CAMBIARLI**
  altrimenti in altre chiamate saranno diversi!
- se vi serve che siano modificati **USATE `None` come default**
  e create il valore che vi serve come default SOLO a run-time

In [22]:

```
# mi aspetterei che ad ogni chiamata senza parametri
# X sia valorizzato con una nuova lista vuota
# ma non è così, la lista è creata una sola volta
# nel momento della definizione della funzione
def modifico_il_default(X=[]):
    X.append(12)                # modifico la lista X
    return X
print( modifico_il_default()) # non passo nessun valore
print( modifico_il_default()) # non passo nessun valore
print( modifico_il_default()) # non passo nessun valore
# si vede come la lista X cambia mano a mano
```

```
[12]
[12, 12]
[12, 12, 12]
```

In [17]:

```
# implementazione corretta
def non_modifico_il_default(X=None):
    # costruisco una nuova lista vuota per ogni chiamata senza parametri
    if  X is None:
        X = []
    X.append(12)
    return X
print( non_modifico_il_default()) # non passo nessun valore
print( non_modifico_il_default()) # non passo nessun valore
print( non_modifico_il_default()) # non passo nessun valore

```

```
[12]
[12]
[12]
```

Cicli ed iterazione

Sintassi:

```
# alla variabile viene assegnato il prox valore
for variabile in sequenza:
    blocco di codice da ripetere per ogni valore
else:         # opzionale
```

Nel corpo del ciclo posso usare

```
break    # per uscire immediatamente dal ciclo
continue # per saltare al prossimo elemento
         # senza completare il blocco
```

se NON sapete quante iterazioni fare

```
while condizione_vera:
    blocco di codice da ripetere
    aggiornamento della condizione
else:       # opzionale
    blocco di codice eseguito se NON si esce
    con break
```

Sequenze: oggetti che forniscono un elemento per volta

La funzione range

range(fine)	**0, 1, 2, 3, 4.... (fine-1)**
range(inizio, fine)	**inizio, inizio+1, ... , fine-1**
range(inizio, fine, incremento)	**inizio, inizio+incremento, ..., fine-1**

In [18]:
```
## range crea un oggetto che fornisce un nuovo valore ogni volta
## che gli viene richiesto dal for
R = range(10)
for i in R:
    print(i, end=' ')
R
list(R)
```

```
0 1 2 3 4 5 6 7 8 9
```

Out[18]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

I contenitori:

liste, tuple, dizionari, insiemi

- **liste**: `[ 1, 2, 3, 4, 5, ]` indicizzate e modificabili
- **tuple**: `( 1, 2, 3, 4, 5, )` indicizzate e immutabili
- **insiemi**: `{ 1, 2, 3, 4, 5, }` senza ripetizioni, NON indicizzati e mutabili
- **dizionari**: `{ 'a': 1, 'b': 2, 'c': 3, }` associazioni chiave unica -> valore, indicizzati sulle chiavi, modificabili

In [19]:
```
# Esempio:
lista_valori = [2, 5, 7, 23, 45, 2, 7, 23, ]
tupla_valori = tuple(lista_valori)   # converto la lista in tupla
set_valori   = set(lista_valori)     # converto la lista in set
dizionario   = { 'a': 1, 'b': 2, 'c': 3, }
lista_valori, tupla_valori, set_valori, dizionario
```

Out[19]:
```
([2, 5, 7, 23, 45, 2, 7, 23],
 (2, 5, 7, 23, 45, 2, 7, 23),
 {2, 5, 7, 23, 45},
 {'a': 1, 'b': 2, 'c': 3})
```

In [20]:

```
## le liste sono indicizzate E **modificabili**
print(lista_valori[4])
lista_valori[5] = 666
print(lista_valori)
```

```
45
[2, 5, 7, 23, 45, 666, 7, 23]
```

In [21]:

```
## gli insiemi NON sono indicizzati e sono **modificabili**
print(set_valori)
print(set_valori.pop())    # estraggo un elemento a caso
set_valori.add(666)        # aggiungo un elemento
print(set_valori)
set_valori[4]              # ERRORE!!!
```

```
{2, 5, 7, 45, 23}
2
{5, 7, 45, 23, 666}
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/tmp/ipykernel_11546/2758334000.py in <cell line: 6>()
      4 set_valori.add(666)       # aggiungo un elemento
      5 print(set_valori)
----> 6 set_valori[4]

TypeError: 'set' object is not subscriptable
```

In [62]:
```
## le tuple sono indicizzate E **immutabili**
print(tupla_valori[4])
tupla_valori[5] = 666   ### se provo a modificarla ERRORE!
```

```
45

---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
/var/tmp/ipykernel_706375/3966009171.py in <cell line: 3>()
      1 ## le tuple sono indicizzate E **immutabili**
      2 print(tupla_valori[4])
----> 3 tupla_valori[5] = 666   ### ERRORE

TypeError: 'tuple' object does not support item assignment
```

In [27]:
```
# i dizionari sono indicizzati dalle chiavi e **modificabili**
print(dizionario.keys())          # estraggo le chiavi -> oggetto
print(list(dizionario.keys()))    # estraggo le chiavi -> lista
print(dizionario['a'])      # stampo il valore associato ad 'a'
dizionario['paperino']  = 42  # aggiungo una coppia chiave -> valore
print(dizionario)           # il dizionario è cambiato
#dizionario['pluto']         # se la chiave non c'è ERRORE!!!

'pluto' in dizionario
list(dizionario.values())
{ 'uno':1, 'due':2, 'uno':11 }
```

```
dict_keys(['a', 'b', 'c', 'paperino'])
['a', 'b', 'c', 'paperino']
1
{'a': 1, 'b': 2, 'c': 3, 'paperino': 42}
```

Out[27]:
```
{'uno': 11, 'due': 2}
```

In [29]:
```
# se voglio stampare gli elementi dei diversi contenitori
for elemento in lista_valori:
    print(elemento, end=' ')   # stampa seguita da spazio invece che '\n'
print('\t\tlista_valori')
for elemento in set_valori:
    print(elemento, end=' ')
print('\t\tset_valori')
for elemento in tupla_valori:
    print(elemento, end=' ')
print('\t\ttupla_valori')
for chiave,elemento in dizionario.items(): # items produce le coppie
    print(chiave,elemento)
dizionario.items()
```

```
2 5 7 23 45 666 7 23            lista_valori
5 7 45 23 666           set_valori
2 5 7 23 45 2 7 23              tupla_valori
a 1
b 2
c 3
paperino 42
```

Out[29]:
```
dict_items([('a', 1), ('b', 2), ('c', 3), ('paperino', 42)])
```

In [30]:
```
#### come scandire un contenitore per valori
for elemento in lista_valori:
    print(elemento, end=' ')
print()
```

```
2 5 7 23 45 666 7 23
```

In [14]:
```
#### come scandire un contenitore per indice
for i in range(len(lista_valori)):
    print(i, lista_valori[i])
```

```
0 2
1 5
2 7
3 23
4 45
5 2
6 7
7 23
```

In [32]:
```
#### come scandire un contenitore per valori ma sapendone l'indice
print()
for indice,elemento in enumerate(lista_valori):
    print(indice, elemento)

list(enumerate(lista_valori))
```

```
0 2
1 5
2 7
3 23
4 45
5 666
6 7
7 23
```

Out[32]:
```
[(0, 2), (1, 5), (2, 7), (3, 23), (4, 45), (5, 666), (6, 7), (7, 23)]
```

E' "PROIBITO" modificare la lista sulla quale si sta iterando?

What if elimino elementi dalla lista che sto scandendo?

- sono nella posizione 3,

- elimino l'elemento,
- il 4° si sposta in posizione 3
- passo all'elemento in posizione 4
- e **MI PERDO QUELLO CHE ERA IN POSIZIONE 4!!!**
- e **HO ERRORE SULL'ULTIMO ELEMENTO!!!**

Insomma, mi tiro via il tappeto da sotto i piedi!!!

In [16]:
```
lista_interi = [ 11, 22, 33, 44, 55, 66, 77, 88, ]
for i in range(len(lista_interi)):
    print(lista_interi[i])
    if i == 3:
        del lista_interi[i]  # elimino l'elemento in posizione 3
```

```
11
22
33
44
66
77
88

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
/tmp/ipykernel_92110/4204336819.py in <cell line: 2>()
      1 lista_interi = [ 11, 22, 33, 44, 55, 66, 77, 88, ]
      2 for i in range(len(lista_interi)):
----> 3     print(lista_interi[i])
      4     if i == 3:
      5         del lista_interi[i]  # elimino l'elemento in posizione 3

IndexError: list index out of range
```

COME RISOLVERE?

- **opzione 1**: scandisco la lista dalla fine all'inizio

- in questo modo le modifiche spostano elementi GIA' ESAMINATI
- ed inoltre gli indici esistono tutti (no IndexError)

- **oppure**: costruisco una nuova lista
  - in questo modo non modifico la lista originale
- **in genere**: itero su una lista SENZA MODIFICARLA (o s una sua copia), e mi costruisco altre strutture dati

In [93]:
```
# scansione a rovescio
lista_interi = [ 11, 22, 33, 44, 55, 66, 77, 88, ]
for i in range(len(lista_interi)-1, -1, -1): # range con incremento negativo
    print(lista_interi[i], end=' ')
    if i == 3:
        del lista_interi[i]
print()
print(lista_interi, 'lista_interi')
```

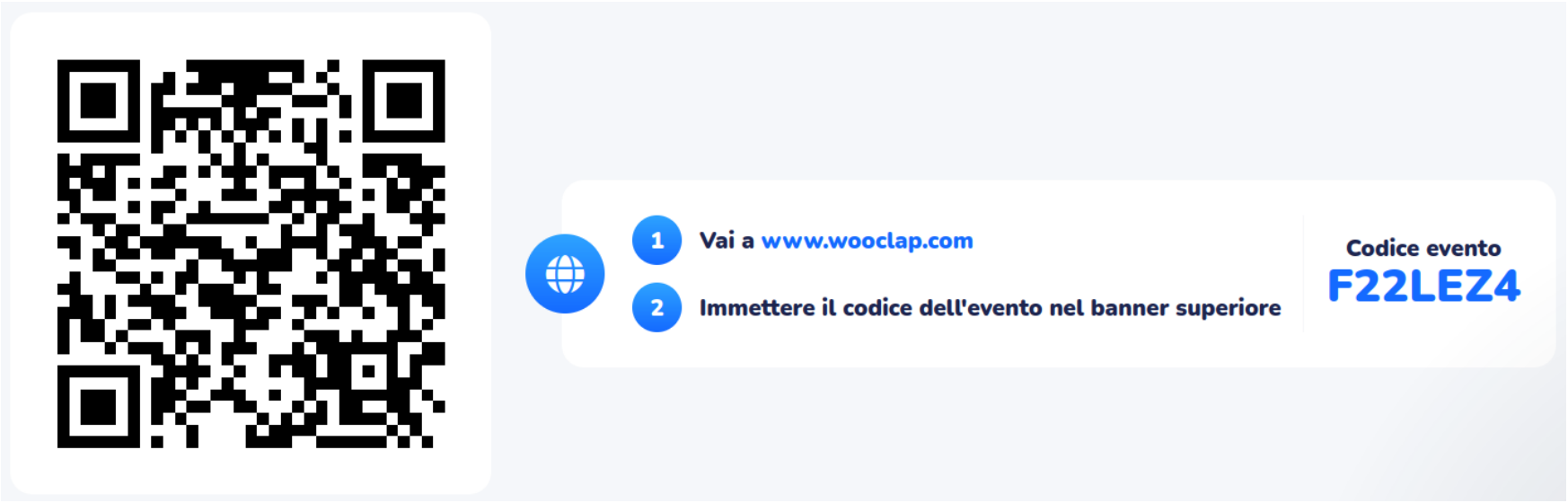
```
88 77 66 55 44 33 22 11
[11, 22, 33, 55, 66, 77, 88] lista_interi
```

In [101]:
```
# oppure costruisco una nuova lista
nuova_lista  = []
lista_interi = [ 11, 22, 33, 44, 55, 66, 77, 88, ]
for i in range(len(lista_interi)):
    if i != 3:
        nuova_lista.append(lista_interi[i])
        print(lista_interi[i], end=' ')
print()
print(lista_interi, 'lista_interi')
print(nuova_lista, 'nuova_lista')
```

```
11 22 33 55 66 77 88
[11, 22, 33, 44, 55, 66, 77, 88] lista_interi
[11, 22, 33, 55, 66, 77, 88] nuova_lista
```

Momento Wooclap

cosa stampa il programma ?



In [33]:
```
# Soluzione
lista_valori = list(range(10))
somma = 0
for i in range(10):
    if i == len(lista_valori):   # esco se sono finiti gli elementi
      break
    if i % 3 == 0:               # per tutti i multipli di 3
      somma += lista_valori[i]   # li sommo
      del lista_valori[i]        # e li elimino
print(somma)
print(lista_valori)  # valori rimasti nella lista
# volevamo sommare          0 3 6 9 => 18
# e invece abbiamo sommato 0 4 8   => 12
```

```
12
[1, 2, 3, 5, 6, 7, 9]
```

Scorciatoie logiche per controllare i contenitori nei test if/then/else

Spesso dobbiamo controllare se un contenitore è vuoto o contiene elementi (oppure se una variabile è 0 o diversa da 0). Per semplificare gli if-then-else:

- un contenitore vuoto vale come **False**
- un contenitore con almeno un elemento vale **True**

Ed inoltre

- un valore 0 vale False
- un valore diverso da zero vale True

In [6]:
```
## Per controllarlo trasformiamo contenitori vuoti in booleani
bool([]), bool(tuple()), bool({}), bool(set())
```

Out[6]:
```
(False, False, False, False)
```

In [5]:
```
bool([1]), bool((2,)), bool({3:'tre'}), bool({4})
```

Out[5]:
```
(True, True, True, True)
```

In [9]:
```
def stampa_lista(valori):
    # se la lista contiene almeno un elemento
    if valori:
        print(valori[0])            # stampo il primo
        stampa_lista(valori[1:])    # stampo gli altri

dati = [1, 2, 3, 4, 5]
stampa_lista(dati)
```

```
1
2
3
4
5
```