lezione04

October 9, 2023

1 Fondamenti di Programmazione

Andrea Sterbini

lezione 4 - 5 ottobre 2023

1.1 RECAP

- stringhe
- slice
- variabili e riferimenti
- oggetti e metodi
- funzioni
- if-elif-else

1.2 Operazioni logiche

- **A and B** (vera se A e B sono entrambi veri)
- **A or B** (vera se almeno uno è vero)
- **not A** (vera se A è falso e viceversa)

PRECEDENZA: **not** –> **and** –> **or**

1.3 If-elif-else

```
if condizione1:
    istruzioni eseguite se è vera la condizione1
elif condizione2:
    istruzioni eseguite se è falsa condizione1 ed è vera condizione2
...
else:
    istruzioni eseguite se tutte le condizioni sono false
```

1.4 Momento Wooclap (booleani e if-then-else)

1 Go to wooclap.com
2 Enter the event code in the top banner

Event code
F23LEZ4

[28]:
```python
## SOLUZIONI
True and not False or True and False or not True
# è come
(True and (not False)) or (True and False) or (not True)
# ovvero
(True and True) or False or False
# quindi
True or False or False
```

[28]: True

[33]:
```python
# Ricordiamo che True == 1 e False == 0
True+False/True+True*True-False
# è lo stesso che
1 + 0/1 + 1*1 - 0
# ovvero
1 + (0/1) + (1*1) - 0
# quindi
1 + 0.0 + 1 - 0        # notate il float
```

[33]: 2.0

[35]:
```python
is_raining = True
got_umbrella = False
have_money = True
if not is_raining:
    print('go out')
elif not got_umbrella and have_money:
    print('buy umbrella and go out')
elif got_umbrella:
    print('go out since got umbrella')
else:
    print('stay home')
```

```
buy umbrella and go out
```

1.5 Namespaces, nomi e identificatori

QUALI sono i nomi ammessi come **identificatori** in python? - iniziano per **carattere alfabetico** oppure '_' (lineetta bassa) - non contengono spazi

1.5.1 Identificatori PARLANTI

Date SEMPRE un nome esplicativo alle funzioni, variabili, argomenti e classi

"Programma sempre come se il ragazzo che dovrà manutenere il tuo codice sia uno psicopatico violento che sa dove vivi." Martin Golding

E **DOVE** si trovano i nomi delle variabili, funzioni e classi?

Nei **NAMESPACE** (tabelle dei nomi)

Ce ne sono almeno 3, uno dentro l'altro:

- namespace **BUILT-IN** (quello più esterno) che contiene:
  - tutte le funzioni e le variabili dell'interprete Python
- namespace **GLOBALE** (quello più esterno) che contiene:
  - le variabili **GLOBALI** accessibili a tutte le funzioni che le seguono nel file
  - le funzioni e le classi definite nel file
  - i moduli/librerie importati con **import nomemodulo**
- namespace di **MODULO** (i moduli importati) che contengono:
  - tutte le variabili, funzioni e classi definite nel modulo

Per usare una variabile/funzione si usa la sintassi **nomemodulo.variabile** oppure **nomemodulo.funzione(argomenti)**

- namespace **LOCALE** alla funzione in esecuzione, che contiene:
  - gli **argomenti formali** della funzione, con i valori forniti quando è stata chiamata
  - le variabili **locali** definite dentro la funzione
  - e che scompaiono quando la funzione ha finito

Ogni volta che usate un identificatore, verrà cercato in ordine **LEGB**: 1. nel namespace della **funzione** (**L**ocal) 2. poi nei namespace che lo racchiudono (**E**nclosing) - possono essere **funzioni** oppure **moduli** 3. poi nel namespace **globale** 4. infine nel namespace **built-in**

QUINDI: una variabile locale può NASCONDERE una variabile globale con lo stesso nome

[5]:
```python
### ESEMPIO
G = 42          # definisco una variabile globale

def funzione_senza_effetti_collaterali(valore):
    'funzione che definisce G come variabile locale'
    G = valore
    print(G, 'dentro la funzione il valore di G è quello locale')

print(G, 'prima di chiamare la funzione')
```

```
funzione_senza_effetti_collaterali(555)
print(G, 'dopo aver chiamato la funzione il valore è quello globale')
```

```
42 prima di chiamare la funzione
555 dentro la funzione il valore di G è quello locale
42 dopo aver chiamato la funzione il valore è quello globale
```

[6]:
```
def funzione_CON_effetti_collaterali(valore):
    global G         # voglio usare la variabile globale e non una nuova locale
    G = valore
    print(G, 'dentro la funzione la variabile G è quella globale')

print(G, 'prima di chiamare la funzione')
funzione_CON_effetti_collaterali(666)
print(G, 'dopo aver chiamato la funzione il valore è cambiato')
```

```
42 prima di chiamare la funzione
666 dentro la funzione la variabile G è quella globale
666 dopo aver chiamato la funzione il valore è cambiato
```

1.6 Per aiutarvi

Negli HW le variabili GLOBALI (e gli attributi di classe) sono proibiti

Aggiungeremo un test che controlla che non usiate GLOBALI o attributi di classe

1.7 Ancora sulle 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 in quel punto **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**

[45]:
```
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_non_distruttiva(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, '\tlista iniziale')
togli_terza(nomi)
print(nomi, '\t\tdopo aver tolto il 3° elemento')
sostituisci_tutte_non_distruttiva(nomi)
print(nomi, '\t\tdopo sostituisci_tutte_non_distruttiva')
sostituisci_tutte_distruttiva(nomi)
print(nomi, '\t\t\t\t\t\tdopo sostituisci_tutte_distruttiva')
```

```
['Paperino', 'Topolino', 'Minnie', 'Annabella', 'Gastone']      lista iniziale
['Paperino', 'Topolino', 'Annabella', 'Gastone']                dopo aver tolto
il 3° elemento
['Paperino', 'Topolino', 'Annabella', 'Gastone']                dopo
sostituisci_tutte_non_distruttiva
['uno', 'due', 'tre']                                           dopo
sostituisci_tutte_distruttiva
```

1.8 Ancora funzioni: argomenti OBBLIGATORI e OPZIONALI

Gli **argomenti formali** nella definizione della funzione possono essere - obbligatori, posizionali, (**sempre all'inizio** della lista) - 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

[51]:
```
# supponiamo di voler concatenare 3 parole
# ad 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 preconfezionata
    return ' '.join([obb1, obb2, opz3] + opz4)

# gli argomenti opzionali sono sia posizionali che usabili per nome
concatena('Minni', 'Paperino', 'Paperoga', ['Ciccio'])
```

[51]: 'Minni Paperino Paperoga Ciccio'

[52]:
```
concatena('Minni', 'Paperino', 'Paperoga', opz4=['Pluto'])
```

[52]: 'Minni Paperino Paperoga Pluto'

[48]:
```
concatena('Minni', 'Paperino', opz4=['Ciccio'])
```

[48]: 'Minni Paperino 42 Ciccio'

[49]:
```
concatena('Minni', 'Paperino')
```

[49]: 'Minni Paperino 42 viva Topolin'

[50]:
```
# 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'])
```

[50]: 'Paperoga Minni Paperino Ciccio'

1.8.1 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

[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]
```

[7]:
```
# implementazione CONSIGLIATA
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]
```

1.9 Per aiutarvi

Negli HW gli argomenti con default mutabile saranno proibiti

Aggiungeremo un test che controlla che non usiate argomenti di default mutabili

2 Cicli ed iterazione

2.1 se sapete quante iterazioni dovete fare usate il FOR

Sintassi:

```
# alla variabile viene assegnato il prox valore
for variabile in sequenza:
    blocco di codice da ripetere per ogni valore
else:         # opzionale
    blocco che viene eseguito se si è usciti
    normalmente (senza break)
```

[11]:
```
# ESEMPIO
for X in [1, 2, 3, 4]:  # ripeto 4 volte, con X che prende i valori 1, 2, 3, 4
    print(X)            # l'istruzione print
```

```
1
2
3
4
```

Nel corpo del ciclo posso usare

```
break    # per uscire immediatamente dal ciclo
         # e proseguire DOPO tutto il FOR
continue # per saltare al prossimo elemento del FOR
         # senza completare il blocco indentato
```

[8]:
```
# ESEMPIO
for X in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    if X % 2 == 0: continue     # salto tutti i numeri pari
    if X == 13: break           # esco se incontro il valore 13
    print(X)
else:
    print('li ho stampati tutti')
```

```
1
3
5
7
9
li ho stampati tutti
```

2.2 se NON sapete quante iterazioni fare usate WHILE

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

[10]:
```
# ESEMPIO
X = 0
while X < 20:
    X += 1                          # aggiorno la condizione
    if X % 2 == 0: continue
    if X == 27:    break
    print(X)
else:
    print('li ho stampati tutti')
```

```
1
3
5
7
9
11
13
15
17
19
li ho stampati tutti
```

2.3 Generatore di sequenze: oggetto che fornisce 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

[12]:
```python
## range crea un oggetto che fornisce un nuovo valore ogni volta
## che gli viene richiesto dal for o da list
R = range(10)
for i in R:
    print(i, end=' ')
R
list(R)
```

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

[12]:
```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

2.4 I contenitori:

2.4.1 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

[23]:
```python
# 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
```

[23]:
```
([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})
```

[20]:
```python
## 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]
```

[24]:
```python
## 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)
Cell In[24], line 6
      4 set_valori.add(666)       # aggiungo un elemento
      5 print(set_valori)
----> 6 set_valori[4]             # ERRORE!!!

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

[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
```

[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!!!

print('pluto' in dizionario)
print(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}
False
[1, 2, 3, 42]
```

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

[28]:
```
# 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')
```

```
2 5 7 23 45 2 7 23              lista_valori
```

[29]:
```
for elemento in set_valori:
    print(elemento, end=' ')
print('\t\tset_valori')
```

```
5 7 45 23 666           set_valori
```

[30]:
```
for elemento in tupla_valori:
    print(elemento, end=' ')
print('\t\ttupla_valori')
```

```
2 5 7 23 45 2 7 23              tupla_valori
```

[31]:
```
# sfruttiamo gli assegnamenti multipli di Python
for chiave,elemento in dizionario.items(): # items produce le coppie
    print(chiave,elemento)
dizionario.items()
```

```
a 1
b 2
c 3
paperino 42
```

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

[32]:
```
#### come scandire un contenitore per valori
for elemento in lista_valori:
    print(elemento, end=' ')
print()
```

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

[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
```

[33]:
```
#### come scandire un contenitore per valori ma sapendone l'indice
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 2
6 7
7 23
```

[33]:
```
[(0, 2), (1, 5), (2, 7), (3, 23), (4, 45), (5, 2), (6, 7), (7, 23)]
```

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

What if elimino elementi dalla lista MENTRE la 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!!!

[35]:
```
lista_interi = [ 0, 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
```

```
0
11
22
33
55
66
77
88
```

```
---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
```

```
Cell In[35], line 3
      1 lista_interi = [ 0, 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
```

2.6 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

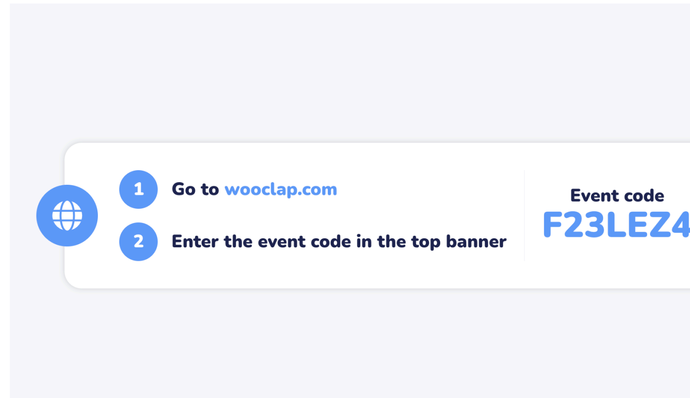
[36]:
```
# scansione a rovescio
lista_interi = [ 11, 22, 33, 44, 55, 66, 77, 88, ]
for i in range(len(lista_interi)-1, -1, -1): # range con incremento negativo
    print(lista_interi[i], end=' ')
    if i == 3:
        del lista_interi[i]
print()
print(lista_interi, 'lista_interi')
```

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

[37]:
```
# 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
```

2.7 Momento Wooclap



cosa stampa il programma ?

[39]:
```
# 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
      print(lista_valori[i])
      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
```

```
0
4
8
12
[1, 2, 3, 5, 6, 7, 9]
```

2.8 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

[40]:
```
## Per controllarlo trasformiamo contenitori vuoti in booleani
bool([]), bool(tuple()), bool({}), bool(set())
```

[40]:
```
(False, False, False, False)
```

[41]:
```
bool([1]), bool((2,)), bool({3:'tre'}), bool({4})
```

[41]:
```
(True, True, True, True)
```

[44]:
```
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
```