

# Fondamenti di programmazione

Canale A-L - Prof. Sterbini



AA 25-26 - Lezione 6

## RECAP

- funzioni, argomenti, valori di ritorno, vita del namespace locale
- analisi di problemi e suddivisione in sottoproblemi più piccoli
- quesiti con la Susi

**FDPLEZ6**



Namespaces, nomi e identificatori

**QUALI** sono i nomi ammessi come **identificatori** in python (per variabili, funzioni, classi e metodi)?

- iniziano per **carattere alfabetico** oppure `_` (lineetta bassa/underscore)
- non contengono spazi (AHA!!! ecco a cosa serve il pool delle stringhe!)
- continuano con caratteri alfanumerici

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 del programma principale) 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/LIBRERIA** (i moduli importati) che contengono:
  - tutte le variabili, funzioni e classi definite nel modulo
  - i moduli importati a loro volta dal modulo

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

- namespace **LOCALE** alla funzione in esecuzione, che contiene:
  - i nomi degli **argomenti formali** della funzione, che contengono i valori forniti quando è stata chiamata (parametri attuali)
  - le variabili **locali** definite dentro la funzione

**NOTA:** Tutto questo namespace scompare quando la funzione ha finito

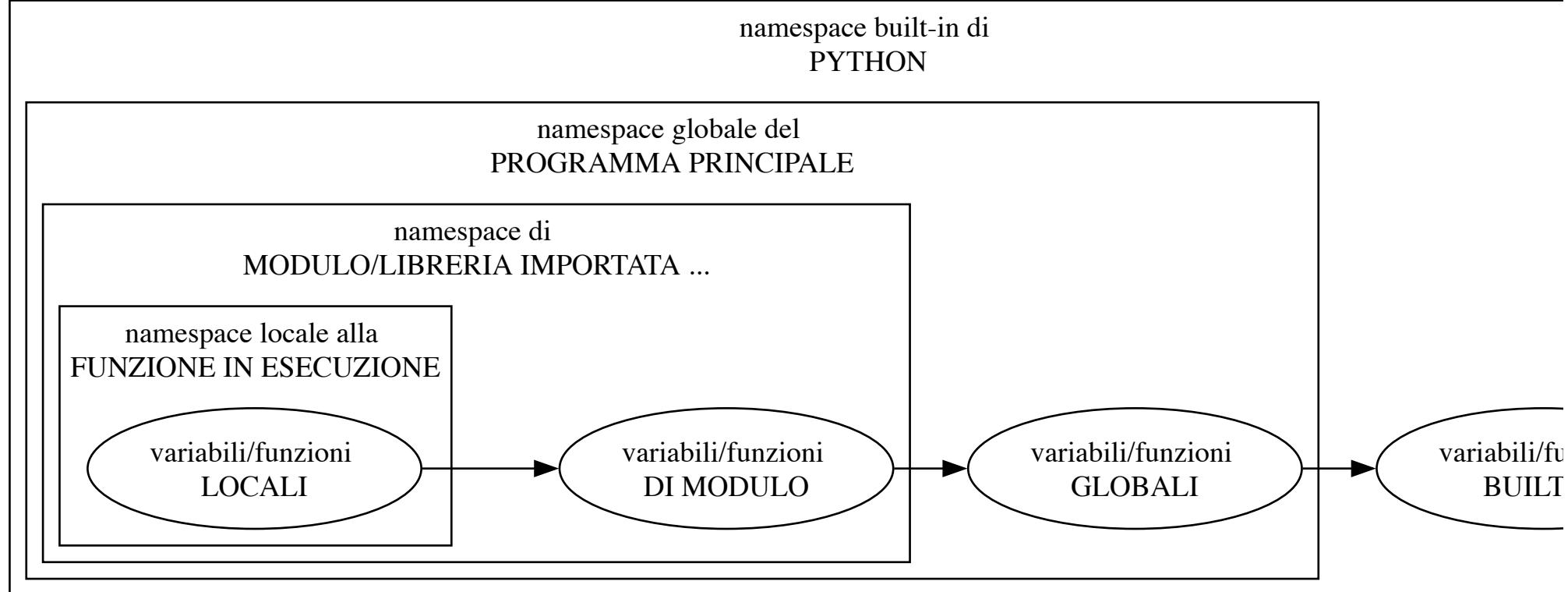
In [1...]

```
# CODICE PER GENERARE IL GRAFICO
from pygraphviz import AGraph
figura3 = AGraph(directed=True, rankdir='LR', label="Ordine di ricerca dei nomi di variabili/funzioni")
figura3.add_node('BI', label='variabili/funzioni \nBUILT-IN')
figura3.add_node('G', label='variabili/funzioni \nGLOBALI')
figura3.add_node('M', label='variabili/funzioni \nDI MODULO')
figura3.add_node('F', label='variabili/funzioni \nLOCALI')
figura3.add_path(['F', 'M', 'G', 'BI'])
BI = figura3.add_subgraph(['BI', 'G', 'M', 'F'], name='cluster_BI', label='namespace built-in di \nPYTHON',
                           name='cluster_G', label='namespace globale del \nPROGRAMMA PRINCIPALE',
                           name='cluster_M', label='namespace di \nMODULO/LIBRERIA IMPORTATA ...',
                           name='cluster_F', label='namespace locale alla \nFUNZIONE IN ESECUZIONE')
figura3.layout(prog='dot')
```

In [2...]

figura3

Out [2...]



Ordine di ricerca dei nomi di variabili/funzioni

# Ordine di ricerca dei nomi

Ogni volta che usate un nome, verrà cercato in ordine **LEGB**:

1. nel namespace della **funzione** in cui vi trovate (**Local**)
2. poi nei namespace che la racchiudono (**Enclosing**)
  - possono essere **moduli** oppure **funzioni**  
(ed è anche possibile definire funzioni nidificate ...)
3. poi nel namespace **globale** del programma principale
4. infine nel namespace **built-in** di Python

La prima corrispondenza trovata VINCE

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

```
In [3...]  
# ESEMPIO  
G = 42          # definisco una variabile globale  
  
### ESEMPIO di funzione che 'nasconde' la globale G con una locale G  
def funzione_senza_effetti_collaterali(valore):  
    "funzione che definisce G come variabile locale e ci copia l'argomento"  
    # definisco un nuovo nome 'G' nel namespace locale alla funzione  
    G = valore  
    print('G = ', G, 'dentro la funzione, dove la variabile G è locale')
```

```
In [4...]  
print('G = ', G, 'prima di chiamare la funzione G è quella globale')  
funzione_senza_effetti_collaterali(555)  
# dopo la chiamata il namespace locale della funzione è stato disallocato  
print('G = ', G, 'dopo aver chiamato la funzione, dove G è quella globale')
```

G = 42 prima di chiamare la funzione G è quella globale  
G = 555 dentro la funzione, dove la variabile G è locale  
G = 42 dopo aver chiamato la funzione, dove G è quella globale

```
In [5...]  
# Esempio di EFFETTI COLLATERALI: (modifiche all'ambiente esterno alla funzione)  
def funzione_CON_effetti_collaterali(valore):
```

```
"funzione che modifica la variabile globale"
# voglio proprio usare la variabile globale e non crearne una nuova locale
global G          # con la parola chiave 'global'
G = valore        # la modifico inserendoci l'argomento fornito
print('G =', G, 'dentro la funzione la variabile G è quella globale')
```

In [6...]

```
print('G = ', G, 'prima di chiamare la funzione G è quella globale')
funzione_CON_effetti_collaterali(666)
# dopo la chiamata il namespace locale della funzione è stato disallocato ma abbiamo usato la globale
print('G = ', G, 'dopo aver chiamato la funzione il valore di G globale è cambiato')
```

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

## Modificare gli argomenti si può?

**NOTA:** gli argomenti sono **variabili locali**, ovvero nomi inseriti nel namespace locale e associati ai valori attuali forniti nella chiamata

- potete modificarli all'interno della funzione
  - un assegnamento `nome = espressione` cambia l'associazione LOCALE nome->oggetto,  
il programma principale **NON se ne accorgerà** perchè questa associazione è solo locale alla funzione (OK)
  - **MA:** se si tratta di oggetti **mutabili** e ne modificate il contenuto  
il programma chiamante **se ne accorgerà** (ATTENZIONE)
    - perchè gli avete modificato l'interno dell'oggetto  
(EFFETTO COLLATERALE)
    - da fare con criterio

In [2...]

```
## Esempio di effetto collaterale modificando una lista in una funzione
nomi = ['Paperino', 'Topolino', 'Minnie', 'Annabella', 'Gastone']
nomi_originale = nomi # mi servirà dopo
def togli_terza(parole):
    "funzione che modifica la lista togliendo l'elemento a indice 2"
    parole.pop(2)      # modifico il CONTENUTO dell'oggetto fornito come argomento
```

```
In [3...]: print(nomi, 'lista iniziale')
togli_terza(nomi)      # eseguo la funzione che ha un EFFETTO COLLATERALE
print(nomi, 'dopo aver tolto il 3° elemento')
nomi is nomi_originale

['Paperino', 'Topolino', 'Minnie', 'Annabella', 'Gastone'] lista iniziale
['Paperino', 'Topolino', 'Annabella', 'Gastone'] dopo aver tolto il 3° elemento
```

Out[3...]: True

```
In [4...]: ## esempio che sostituisce il contenuto di 'parole' nel namespace locale
## (e non nel namespace globale del programma principale)
def sostituisci_tutte_non_distruttiva(parole):
    "funzione che rimpiazza nella variabile locale 'parole' una nuova lista"
    # sostituisco l'oggetto associato al nome 'parole'
    parole = ['uno', 'due', 'tre']
    # il resto del programma NON se ne accorge
    # perchè ho cambiato la corrispondenza nome_locale -> valore nel namespace LOCALE
    # senza toccare il namespace del programma che chiama la funzione
```

```
In [5...]: ## PENSavo di aver sostituito nomi con una nuova lista MA
print(nomi, 'prima di sostituisci_tutte_non_distruttiva')
sostituisci_tutte_non_distruttiva(nomi)
print(nomi, 'dopo sostituisci_tutte_non_distruttiva (resta uguale)')
# MA la lista rimane invariata nel programma che ha chiamato la funzione
nomi is nomi_originale

['Paperino', 'Topolino', 'Annabella', 'Gastone'] prima di sostituisci_tutte_non_distruttiva
['Paperino', 'Topolino', 'Annabella', 'Gastone'] dopo sostituisci_tutte_non_distruttiva (resta uguale)
```

Out[5...]: True

```
In [8]: ## Esempio che rimpiazza IL CONTENUTO della lista passata come argomento
## per modificare il CONTENUTO della lista posso usare un assegnamento a slice
def sostituisci_tutte_distruttiva(parole):
    """funzione che rimpiazza il **contenuto**
       della lista fornita in 'parole'
       con una nuova lista"""
    # sostituisco tutti gli elementi INTERNI alla lista con i nuovi valori
    parole[:] = ['uno', 'due', 'tre']    # con un assegnamento a slice
```

`['Paperino', 'Topolino', 'Annabella', 'Gastone'] prima di sostituisci_tutte_distruttiva`  
`['uno', 'due', 'tre'] dopo sostituisci_tutte_distruttiva`

Out[9... True

**Quindi:**

possiamo modificare le info del programma principale solo DENTRO oggetti mutevoli forniti come argomenti

Va fatta con attenzione perchè è un effetto collaterale che non è ESPLICITAMENTE chiaro leggendo il programma

Per restituire informazioni al programma chiamante  
è meglio usare `return`

- è più chiaro, è una istruzione che ESPLICATAMENTE ci dice cosa la funzione dà indietro a chi la chiama
  - quando vedrete le annotazioni di tipo, vedrete che possiamo annotare sia gli argomenti che il valore ritornato, e indicare di che tipo sono

```
def nome_funzione(argomenti):  
    corpo della funzione  
    if condizione:
```

```
    return risultato1  
altro codice  
    return risultato2
```

- **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**

**DEBUG TRICK:** se il vostro programma si lamenta di un **NoneType** cercate la funzione in cui vi siete dimenticati di mettere il **return**

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

## Ancora Funzioni: argomenti OBBLIGATORI e OPZIONALI

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

- obbligatori, posizionali, (e sono sempre all'inizio degli argomenti)
  - vengono riconosciuti dalla posizione che hanno nell'elenco di argomenti
- opzionali, con un valore di default da usare se il valore attuale non viene fornito (sempre alla fine dell'elenco di argomenti)
  - vengono riconosciuti per NOME ma anche dalla POSIZIONE
- PRIMA tutti gli obbligatori POI tutti gli opzionali con i loro default

Nella chiamata si mettono prima i **valori attuali obbligatori** nello stesso ordine,

poi gli **opzionali** per posizione oppure per nome

## Sintassi

- obbligatori: `nome_argomento`
- opzionali: `nome_argomento = valore_di_default`
  - il valore di default viene assegnato (nel namespace LOCALE)
  - solo se la chiamata non lo ha fornito
  - viene creato in memoria NEL MOMENTO DELLA DEFINIZIONE

In [1...]

```
# supponiamo di voler concatenare 3 parole
# ad una lista di altre parole di lunghezza variabile
def concatena(obb1, obb2, opz3=42, opz4=None):
    'i primi due argomenti sono obbligatori e gli altri 2 opzionali'
    if not isinstance(opz3, str):      # se il 3° argomento NON è una stringa
        opz3 = str(opz3)                # lo trasformo in stringa
    if opz4 is None:                  # se il 4° argomento non c'è
        opz4 = ['viva', 'Topolin']    # uso una lista di valori preconfezionata
    lista_parole = [obb1, obb2, opz3] + opz4
    return ' '.join(lista_parole) # e li concateno tutti
```

In [1...]

```
# anche gli argomenti opzionali possono essere usati per posizione
# esempio: se li passo tutti e quattro va tutto bene
concatena('Minni', 'Paperino', 'Paperoga', ['Ciccio'])
#          obb1      obb2       opz3        opz4
```

Out[1...]

```
'Minni Paperino Paperoga Ciccio'
```

In [1...]

```
# oppure li fornisco per nome, proviamo a sostituire il 4° con una lista diversa di parole
# indico esplicitamente il nome dell'argomento
concatena('Minni', 'Paperino', 'Paperoga', opz4=['Pluto', 'Clarabella'])
```

Out[1...]

```
'Minni Paperino Paperoga Pluto Clarabella'
```

In [1...]

```
# qui invece NON fornisco opz3, per cui viene usato il valore di default 42
# DEVO passare opz4 per nome altrimenti per posizione verrebbe preso come opz3
concatena('Minni', 'Paperino', opz4=['Ciccio'])
```

```
Out[1]: 'Minni Paperino 42 Ciccio'
```

```
In [1]: # qui passo solo gli obbligatori e non fornisco nè opz3 nè opz4  
concatena('Minni', 'Paperino')
```

```
Out[1]: 'Minni Paperino 42 viva Topolin'
```

```
In [1]: # posso passare gli argomenti *fuori sequenza* usando i nomi  
# potete anche passare tutti gli argomenti per nome in qualsiasi ordine  
concatena(obb2='Minni', opz3='Paperino', obb1='Paperoga', opz4=['Ciccio'])
```

```
# (e se ripetete più volte lo stesso nome di argomento, viene assegnato l'ultimo valore)
```

```
Out[1]: 'Paperoga Minni Paperino Ciccio'
```

```
In [1]: # NOTA: gli obbligatori CI DEVONO ESSERE TUTTI (qui manca obb1)  
concatena(obb2='Minni', opz3='Paperino', opz4=['Ciccio'])
```

```
-----  
TypeError
```

```
Cell In[19], line 2
```

```
    1 # NOTA: gli obbligatori CI DEVONO ESSERE TUTTI (qui manca obb1)  
----> 2 concatena(obb2='Minni', opz3='Paperino', opz4=['Ciccio'])
```

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

```
TypeError: concatena() missing 1 required positional argument: 'obb1'
```

## APPROFONDIMENTO:

Un errore molto difficile da notare:  
valori di default modificabili

Python crea i valori di default degli argomenti opzionali

**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 [1..]

```
# Esempio con effetti collaterali "incomprensibili"
# 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
# e viene condivisa tra tutte le chiamate
def modifico_il_default(X=[]):
    X.append(12)                      # modifico la lista X
    return X
print(modifico_il_default([])) # non passo nessun valore, [] viene riusato
print(modifico_il_default([])) # non passo nessun valore, [] viene riusato
print(modifico_il_default()) # non passo nessun valore, [] viene riusato
print(modifico_il_default()) # non passo nessun valore, [12] viene riusato
print(modifico_il_default()) # non passo nessun valore, [12,12] viene riusato
# si vede come la lista X cambia mano a mano
```

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

In [1..]

```
## implementazione CONSIGLIATA, uso None come default (valore immutabile)
def non_modifico_il_default(X=None):
    if X is None:          # se NON fornisco l'argomento X
        X = []              # lo valorizzo con una lista vuota creata NUOVA per ciascuna chiamata
    X.append(12)            # e poi faccio quello che mi pare
    return X
print(non_modifico_il_default()) # non passo nessun valore, viene creata una [] nuova
print(non_modifico_il_default()) # non passo nessun valore, viene creata una [] nuova
```

```
print( non_modifico_il_default([1,2,3])) # non passo nessun valore, viene creata una [] nuova
```

```
[12]  
[12]  
[1, 2, 3, 12]
```

In [2...]

```
# 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]
```

## Quesito con la Susi 2 (Settimana Enigmistica)

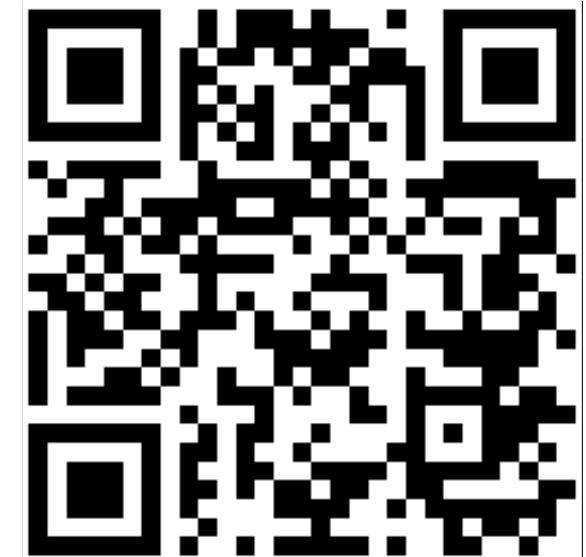
## Quesito con la Susi



- un amico della Susi ha fatto un esame a crocette True/False con 100 domande
- una volta uscito ha saputo che:
  - le risposte giuste **True** erano tutte quelle **multiple di 3** e il resto **False**
  - ma lui ha marcato **False** tutte le domande **multiple di 4** ed il resto **True**

Quante ne ha azzeccate?

FDPLEZ6



# Ancora problem-solving!

Schematizziamo una possibile soluzione SU CARTA

## MODO 1: scandendo le risposte e contando

- per contare quelle azzeccate
  - scandisco i numeri da 1 a 100 e per ciascuno
    - calcolo la risposta corretta (sottoproblema più semplice)
    - calcolo la risposta data (sottoproblema più semplice)
    - se sono uguali incremento il conteggio (ovvio)
  - torno il conteggio (ovvio)

(cosa manca? l'inizializzazione delle variabili!)

In [2...]

```
# per contare le risposte giuste
def conta_azzeccate():
    # inizializzo una variabile per contare quelle azzeccate a 0
    azzeccate = 0
    for i in range(1,101):
        corretta = risposta_corretta(i)
        data     = risposta_data(i)
        if corretta == data:
            azzeccate += 1
    return azzeccate
    # scandisco i numeri da 1 a 100 compreso
    # calcoliamo la risposta corretta (DA DEFINIRE)
    # calcoliamo la risposta data (DA DEFINIRE)
    # se la riposta data è uguale alla risposta corretta
    # incremento il conteggio
    # alla fine torno il conteggio
```

- per calcolare la risposta **corretta** X-esima
  - torno True se **X** è multiplo di 3

In [2...]

```
# per calcolare la risposta corretta
def risposta_corretta(X):
    # se il numero è divisibile per 3
        # la risposta corretta è True
    # altrimenti
```

```
# la risposta corretta è False
return X%3 == 0           # posso usare direttamente il risultato della condizione
```

- per calcolare la risposta **data** X-esima
  - torno True se **X NON è multiplo di 4**

In [2...]

```
# per calcolare la risposta data
def risposta_data(X):
    # se il numero è divisibile per 4
        # lui ha risposto False
    # altrimenti
        # lui ha risposto True
    return X % 4 != 0

# Noooo Non vi do subito il risultato 3-)
```

## MODO 2: Un modo completamente diverso: con gli insiemi

- costruisco i due insiemi delle risposte **corrette\_True** e **corrette\_False**
- costruisco i due insiemi delle risposte **date\_True** e **date\_False**
- le risposte azzeccate sono:
  - le **date\_True** che sono **corrette\_True**
  - e le **date\_False** che sono **corrette\_False**

In [2...]

```
# un altro modo di rispondere: con gli insiemi
# raccolgo le risposte corrette
#   in due insiemi corrette_True e corrette_False
def conta_azzeccate_con_set():
    domande      = set(range(1,101))          # insieme dei numeri da 1 a 100
    corrette_True = set(range(3,101,3))        # insieme dei multipli di 3    (che sono True)
    corrette_False= domande - corrette_True    # insieme dei NON multipli di 3 (che sono False)
    # raccolgo le risposte date in due insiemi date_True e date_False
    date_False = set(range(4,101,4))          # insieme dei multipli di 4    (che ha risposto False)
    date_True = domande - date_False          # insieme dei NON multipli di 4 (che ha risposto True)
    # Le risposte giuste sono:
```

```

# l'intersezione delle risposte date_False con le risposte corrette_False (and/intersection)
# assieme a (or/union)
# l'intersezione delle risposte date_True con le risposte corrette_True (and/intersection)
azzeccate = (corrette_True & date_True) | (corrette_False & date_False)
print('azzeccate con set', azzeccate)
return len(azzeccate)

# il risultato cercato è il numero di elementi dell'insieme

# Noooo ... non ve lo dico, prima dovete risolverlo voi

```

### MODO 3: Ma se invece usassimo dei dizionari domanda->risposta ?

- costruisco il dizionario **risposte\_corrette** che contiene **domanda->risposta corretta** (True/False)
- costruisco il dizionario **risposte\_date** che contiene **domanda->risposta data** (True/False)
- confronto i dizionari e conto le coppie uguali

In [2...]

```

## Un terzo modo usando i dizionari
# costruisco i dizionari
#   numero -> risposta giusta
#   numero -> risposta data
def conta_azzeccate_con_dict():
    dizionario_risposte = {}
    for i in range(1, 101):                      # per ogni indice
        dizionario_risposte[i] = (i % 3 == 0)    # setto il valore True/False
    dizionario_date = {}
    for i in range(1, 101):
        dizionario_date[i] = (i % 4 != 0)
    # e poi calcolo l'intersezione dei due insiemi di items (domanda, risposta)
    azzeccate_diz = set(dizionario_risposte.items()) & set(dizionario_date.items())
    # La risposta dopo il break
    print('azzeccate con dict', azzeccate_diz)
    return len(azzeccate_diz)

```

In [2...]

```

# vediamo finalmente quante ne ha azzeccate
print('azzeccate', conta_azzeccate(), conta_azzeccate_con_set(), conta_azzeccate_con_dict())

```

```
azzeccate con set {3, 4, 6, 8, 9, 15, 16, 18, 20, 21, 27, 28, 30, 32, 33, 39, 40, 42, 44, 45, 51, 52, 54, 56, 57, 63, 64, 66, 68, 69, 75, 76, 78, 80, 81, 87, 88, 90, 92, 93, 99, 100}
azzeccate con dict {(69, True), (4, False), (45, True), (92, False), (27, True), (3, True), (28, False), (93, True), (51, True), (8, False), (52, False), (32, False), (76, False), (18, True), (56, False), (100, False), (80, False), (9, True), (99, True), (75, True), (42, True), (57, True), (33, True), (66, True), (90, True), (16, False), (81, True), (6, True), (39, True), (15, True), (40, False), (20, False), (64, False), (21, True), (30, True), (44, False), (88, False), (68, False), (87, True), (63, True), (78, True), (54, True)}
azzeccate 42 42 42
```

Altre idee? (pensateci voi :-) )

## Ancora argomenti formali delle definizioni delle funzioni

Avrete notato che la funzione **print** accetta un numero indefinito di argomenti

Ma come fa? E' speciale?

No, approfitta della sintassi degli **assegnamenti multipli** e del **packing/unpacking** (stay tuned)

## Ancora assegnamenti multipli: "packing" ed "unpacking"

Finora abbiamo visto assegnamenti multipli in cui il numero di variabili ed il numero di elementi da assegnare **dovevano essere uguali**

Che succede se invece il numero di elementi non corrisponde? ERRORE

```
In [2...]: A, B, C = [1, 2, 3, 4]
```

```
-----  
ValueError
```

```
Cell In[27], line 1
```

```
----> 1 A, B, C = [1, 2, 3, 4]
```

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

```
ValueError: too many values to unpack (expected 3)
```

Con il **packing** possiamo raccogliere in una sola variabile tutti i valori rimanenti di una lista

Il nome della variabile deve essere preceduto da asterisco \*

**NOTA** il packing avviene nelle variabili a SINISTRA di un assegnamento

In [2...]

```
# PACKING: raccolgo in C tutti i valori rimanenti
# tranne i primi due che vanno in A e B
# mettendo un asterisco subito prima del nome della variabile C
A, B, *C = [ 1, 2,
print('A:',A)
print('B:',B)
print('C:',C)  # C contiene una lista
```

A: 1  
B: 2  
C: [3, 4, 5, 6]

In [3...]

```
# Esempio di packing che raccoglie *lista* vuota
A, B, *C = (1,2)
print('C ora è', C)
```

C ora è []

In [3...]

```
# ancora meglio, posso mettere la variabile "packed" all'inizio!!!
*A, B, C = [ 1, 2, 3, 4, 5, 6]
print('A:',A)
print('B:',B)
print('C:',C)
```

A: [1, 2, 3, 4]  
B: 5  
C: 6

In [3...]

```
# oppure in mezzo!!!
A, *B, C = [ 1, 2, 3, 4, 5, 6]
print('A:',A)
print('B:',B)
print('C:',C)
```

A: 1  
B: [2, 3, 4, 5]  
C: 6

Basta che ci sia una sola variabile "packed"  
in modo che Python sappia cosa mettere nelle altre,  
così metterà il resto in quella con asterisco

In [3...]: ## Quindi la definizione di print dev'essere simile a questa

```
def my_print(*roba_da_stampare, end='\n', sep=' '):
    "Funzione che permette un numero variabile di valori, più due argomenti opzionali"
    # roba_da_stampare conterrà una *tupla* con tutti i valori passati
    # NOTA: print prima converte tutto in stringhe per cui
    stringhe = []
    for valore in roba_da_stampare:
        stringhe.append(str(valore))
    # e poi stampa (qui uso print per semplicità)
    print(sep.join(stringhe), end=end)

my_print(1, 1.3, [1, 2, 3])
```

1 1.3 [1, 2, 3]

E l' "unpacking" cos'è?

Sempre con l'asterisco possiamo "spacchettare" una variabile che contiene più valori all'interno di una altra **espressione** (ad esempio nella parte DESTRA di un assegnamento)

```
X = contenitore
[ ..., *X, ... ]
diventa
[ ..., elementi di X, ... ]
```

**NOTA** l'unpacking avviene quando si **calcola** il valore della espressione

In [3...]: # esempio di unpacking di un \*set\* all'interno di una tupla

```
X = {11, 22, 33, 44}
D = (1, 2, *X, 4)      # nella tupla appaiono gli elementi di X (in ordine indefinito perchè è un set)
```

```
print('D ora è:', D)
```

```
D ora è: (1, 2, 33, 11, 44, 22, 4)
```

```
In [3...]: # Esempio di unpacking di una sequenza di caratteri all'interno di una lista  
X = 'ABCDE'  
[ 1, 2, 3, 4, *X, 5, 6, 7, 8, ] # la espando in mezzo all'altra sequenza
```

```
Out[3...]: [1, 2, 3, 4, 'A', 'B', 'C', 'D', 'E', 5, 6, 7, 8]
```

**RIASSUMENDO:** se un asterisco precede il nome di una variabile:

- a SINISTRA di un **assegnamento** fa il **packing** di zero o più valori in una **lista** (**tupla** se sono gli argomenti di una funzione)
- in una **espressione** (a DESTRA di un assegnamento) fa l'**unpacking** dei valori contenuti nella variabile

## Input di valori da tastiera

Per prendere dei dati da tastiera si usa la funzione **input( [prompt] )** che:

- prende come argomento un "prompt" opzionale (un testo che viene mostrato per guidare l'utente)
- torna come risultato la **stringa** inserita, compresa di '**\n**' finale

**NOTA** torna sempre una stringa, se vi serve un tipo di dato diverso dovete convertirlo

```
In [1...]: risposta = input('Inserisci un intero tra 1 e 100: ')  
numero = int(risposta)  
print(numero)
```

```
ValueError                                     Traceback (most recent call last)
Cell In[15], line 2
      1 risposta = input('Inserisci un intero tra 1 e 100: ')
----> 2 numero   = int(risposta)
      3 print(numero)

ValueError: invalid literal for int() with base 10: '3.56'
```

```
In [3...]: trovato = False
while not trovato:
    risposta = input('Inserisci un intero tra 1 e 100: ')
    trovato = 1 <= int(risposta) <= 100
numero = int(risposta)
numero
```

```
Out[3...]: 35
```

ma come gestire input completamente errati?

- float: questi potremmo accettarli
- parole: questi no

## Come catturare e gestire gli errori

```
try:
    codice che potrebbe generare un errore
# catturo in 'E' l'oggetto che contiene le info dell'errore
# ma SOLO se è del tipo indicato
except NomeDellaClasseDell'Errore as E:
    codice da eseguire se è c è proprio questo errore
... altri except
finally: # opzionale
    codice eseguito dopo tutti gli except IN OGNI CASO
si continua qui solo se l errore è stato catturato
```

```
In [3...]  
def is_float_or_int(S):  
    'funzione che torna True se una stringa rappresenta un float o un int'  
    try:  
        float(S) # provo a convertire la stringa in float (prende anche gli int)  
        return True # qui potrei generare un ValueError  
    except ValueError:  
        # se tutto è andato bene è un int o float e torno True  
        # altrimenti se catturo un ValueError  
        return False # non è né intero né float e torno False
```

```
In [3...]  
# esempio di input che gestisce anche valori float e desidera un numero tra 1 e 100 compresi  
risposta = input('Inserisci un numero tra 1 e 100: ')  
while (not is_float_or_int(risposta)) or not (1 <= float(risposta) <= 100):  
    # ripeto se non è int o float  
    # o se non è tra 1 e 100  
    risposta = input('Inserisci un numero tra 1 e 100: ') # lo ri-chiedo  
  
# alla fine voglio che sia un intero ma potrebbe avere la virgola  
numero = int(float(risposta))  
print(numero)
```

9

## Digressione: l'operatore `:=` walrus (tricheco)

Se in una espressione avete una parte del calcolo che volete memorizzare in una variabile, potete usare l'operatore walrus (`:=`). Per esempio:

```
L = [1, 2, 3, 4, 5]  
# calcolo la somma E INOLTRE la metto nella variabile `somma`  
media = (somma := sum(L))/len(L)
```

La espressione `somma := sum(L)` ha lo stesso valore di `sum(L)` ma definisce anche la variabile `somma` e ci mette il risultato del calcolo a destra del `:=`.

E' comodo ma talvolta rende il codice meno leggibile. Conviene calcolare e memorizzare il pezzo di espressione che ci interessa in una istruzione separata, e dopo usare la variabile col valore calcolato.

```
In [4...]  
# Esempio in cui uso il := per evitare di ricalcolare più volte float(risposta)  
# e per scrivere il while in modo più compatto
```

```

risposta = input('Inserisci un numero tra 1 e 100: ')
while (not is_float_or_int(risposta)) # ripeto se non è int o float
    or not (1 <= (numero := float(risposta)) <= 100)): # tra 1 e 100
    risposta = input('Inserisci un numero tra 1 e 100: ') # altrimenti lo ri-chiedo
numero = int(numero) # alla fine voglio che sia un intero
print(intero, numero)
# nota: passando per i float 99.99999999999999 dà 99 e 99.99999999999999 dà 100 ... perchè?

```

100 100.0

## Come ordinare gli elementi di un contenitore

- col metodo **sort** delle liste (modifica distruttiva)
  - le tuple no, che non sono modificabili
  - i set no, che non hanno "ordine"
  - i dizionari no, che ordinano in base all'inserimento
- oppure con la funzione **sorted(contenitore) -> list** che accetta qualsiasi contenitore e torna una nuova lista ordinata

L'ordinamento applicato è quello **crescente** naturale per il tipo di dato (crescente numerico per gli **int** e **float** oppure crescente alfabetico per le **str**)

In [4...]

```

# Esempio
L = [ 'uno', 'due', 'tre', 'quattro', 'cinque', 'sei', 'sette', 'otto', 'nove']
print('usando sorted(L):', sorted(L))
L.sort()
print('usando L.sort() :', L)

```

usando sorted(L): ['cinque', 'due', 'nove', 'otto', 'quattro', 'sei', 'sette', 'tre', 'uno']  
 usando L.sort() : ['cinque', 'due', 'nove', 'otto', 'quattro', 'sei', 'sette', 'tre', 'uno']

**NOTA** tutti gli elementi da ordinare devono essere **confrontabili**

- **OK** interi e float
- **OK** stringhe e stringhe
- **NOT OK** interi e stringhe

```
In [4...]: ## Otteniamo l'ordine OPPOSTO con l'argomento opzionale reverse=True  
print(sorted(L, reverse=True))
```

```
['uno', 'tre', 'sette', 'sei', 'quattro', 'otto', 'nove', 'due', 'cinque']
```

## Ordinamenti complessi

Come fare per ordinarle in modo più sofisticato?

Esempio: voglio ordinare `[ 'uno', 'due', 'tre', 'quattro', 'cinque', 'sei', 'sette' ]`:

1. per **lunghezza crescente delle parole**
2. e se sono di lunghezza uguale, in ordine alfabetico

Potrei **trasformare ciascun elemento in una coppia**:

1. lunghezza della parola
2. parola

e cercare di ordinare la nuova lista di coppie

```
In [4...]: L = [ 'uno', 'due', 'tre', 'quattro', 'cinque', 'sei', 'sette']  
# costruisco la lista di coppie  
L1 = []  
for elemento in L:  
    L1.append((len(elemento), elemento))  
print('le coppie sono :',L1)  
# la ordino  
L1.sort()  
print('una volta ordinate:',L1)
```

```
le coppie sono : [(3, 'uno'), (3, 'due'), (3, 'tre'), (7, 'quattro'), (6, 'cinque'), (3, 'sei'), (5, 'sette')]  
una volta ordinate: [(3, 'due'), (3, 'sei'), (3, 'tre'), (3, 'uno'), (5, 'sette'), (6, 'cinque'), (7, 'quattro')]
```

**FUNZIONA! MA PERCHE'?**

Perchè per confrontare due coppie (e capire chi viene prima):

1. prima confrontiamo il primo elemento (lunghezza della parola)
2. poi solo in caso di parità passiamo a confrontare il secondo elemento (ordine alfabetico delle parole)

... e questo era proprio quello che volevamo!

In [4...]

```
# a questo punto basta estrarre dalle coppie solo la parola originale
L2 = []
for lunghezza,elemento in L1:
    L2.append(elemento)
L2
```

Out[4...]

```
['due', 'sei', 'tre', 'uno', 'sette', 'cinque', 'quattro']
```

## Ma se volessi che l'ordinamento fosse STABILE ?

Ovvero che elementi "uguali" che erano in un certo ordine relativo

mantengano lo stesso ordine relativo

**BASTA aggiungere alla tupla anche la posizione originale dell'elemento**

In [4...]

```
## Esempio
L = [ 'uno', 'due', 'tre', 'sette', 'cinque', 'sei', 'sette']
# costruisco la lista di terne
L1 = []
# assieme all'elemento ne estraggo la posizione con enumerate
for pos,elemento in enumerate(L):
    # aggiungo la posizione *pos* come ulteriore criterio da usare DOPPO gli altri
    L1.append((len(elemento), elemento, pos))
L1
# la ordino, i due 'sette' sono nello stesso ordine di prima
L1.sort()
L1
```

```
Out[4... [(3, 'due', 1),  
          (3, 'sei', 5),  
          (3, 'tre', 2),  
          (3, 'uno', 0),  
          (5, 'sette', 3),  
          (5, 'sette', 6),  
          (6, 'cinque', 4)]
```

Questo tecnica di trasformazione di ciascun elemento si chiama [trasformazione di Schwartz](#)

E' stata inventata da Randal L. Schwartz nel 1977 per ordinare in modo efficiente delle liste in Perl

e può essere semplificata introducendo una piccola funzione che trasforma ciascun elemento in una tupla  
da passare a **sorted** (che genera sempre un ordinamento [stabile](#))

```
In [4... # definisco la trasformazione di un solo elemento nella coppia corrispondente  
# la posizione non serve perché sorted è stabile  
def trasforma_elemento(parola):  
    return len(parola), parola
```

```
In [4... # con la funzione *sorted* costruiamo una nuova lista ordinata  
# fornendo col parametro opzionale *key* la funzione di trasformazione (per NOME!!!)  
sorted(L, key=trasforma_elemento)  
# NOTA: non c'è bisogno di mettere la posizione,  
# visto che *sorted* è stabile (ci pensa lei)
```

```
Out[4... ['due', 'sei', 'tre', 'uno', 'sette', 'sette', 'cinque']
```

Proviamo di nuovo, ma con un ordinamento più complicato

1. in ordine di lunghezza
2. in caso di parità **alfabetico ignorando il case** (MAIUSCOLO/minuscolo)
3. in caso di parità **alfabetico**

```
In [4... # il nuovo criterio di ordinamento costruisce una terna
```

```
def criterio_di_ordinamento(parola):
    return ( len(parola),          # per prima cosa la lunghezza
            parola.lower(),       # poi la parola in minuscole
            parola )              # poi la parola così com'è
L = [ 'PaPerino', 'plUTO', 'TopoLINO', 'minniE', 'PLUTO', 'papeRINO', 'papEROne', 'gasTONE', 'MInnie' ]
S = sorted(L, key=criterio_di_ordinamento, reverse=True)
```

In [4...]: # ed ecco il risultato (bisognava notare l'argomento reverse=True)  
print(S)

```
['TopoLINO', 'papEROne', 'papeRINO', 'PaPerino', 'gasTONE', 'minniE', 'MInnie', 'plUTO', 'PLUTO']
```