A.D. 1308
unipg

DIPARTIMENTO
DI INGEGNERIA

Python: Variabili e Espressioni

Gabriele Costante

www.unipg.it

Prima di iniziare: la funzione *print()*

- Mentre nella modalità interattiva l'output delle istruzioni viene visualizzato nel terminale senza richiedere esplicitamente di farlo, nella modalità script è necessario chiedere all'interprete di stampare tramite la console il valore desiderato.
- Per farlo, è possibile utilizzare la funzione *print()* di Python
- Questa funzione è di estrema utilità soprattutto durante la fase di sviluppo per il debug e la verifica del corretto funzionamento del codice
- La funzione *print()* è estremamente flessibile:
    - Può prendere come un numero arbitrario di argomenti separati da virgole
    - Gli argomenti non devono essere necessariamente tutti dello stesso tipo
    - Dopo aver stampato il valore richiesto, la funzione inserisce il carattere '\n' automaticamente (ovvero manda a capo il testo nella console). Si può cambiare il comportamento al termine della stampa del valore fornendo il parametro *end=stringa* che di default è '\n':
        - print('ciao', end=', ')

Prima di iniziare: la funzione *print()*

Codice

```
1  print("Hello world")
2  print("In aula ci sono", 50, "studenti")
3  print("Posso visualizzare interi come", 10, "oppure float come ", 3.5)
```

Console output

```
Hello world
In aula ci sono 50 studenti
Posso visualizzare interi come 10 oppure float come  3.5
```

Prima di iniziare: il valore None

- In Python il valore ‘nullo’ è codificato dalla keyword **None**
- È una costante speciale che rappresenta l’assenza di un valore o un valore nullo

Prima di iniziare: commenti in Python

- A volte per rendere il codice più comprensibile e leggibile (soprattutto nel caso di programmi molto complessi e lunghi) è utile inserire dei commenti
- In Python, i commenti possono essere inseriti utilizzando il carattere cancelletto #
- Se all'inizio di una riga viene inserito il carattere cancelletto, tutto il testo che lo segue verrà considerato un commento e sarà quindi ignorato dall'interprete
- Nel caso di commenti molto lunghi, che si articolano in più righe è possibile:
    - Spezzare semplicemente il commento in più righe inserendo il cancelletto # all'inizio di ogni riga
    - Racchiudere il commento in un blocco delimitato da 3 virgolette o apici (" oppure ') all'inizio e alla fine. In questo caso, finché l'interprete non trova le 3 virgolette (o 3 apici) chiusura, interpreterà tutto il testo come commento

Prima di iniziare: commenti in Python

Codice

```
1   # Questo è un commento inserito utilizzando il cancelletto
2
3   # Questo è un commento molto molto molto molto molto molto molto
4   # molto molto molto molto molto molto molto molto molto
5   # molto molto molto molto molto molto molto lungo inserito utilizzando
6   # il cancelletto all'inizio di ogni riga
7
8   """ Questo è un commento molto molto molto molto molto molto molto
9   molto molto molto molto molto molto molto molto molto
10  molto molto molto molto molto molto molto lungo inserito in un blocco
11   racchiuso tra 3 virgolette doppie all'inizio e alla fine
12  """
```

Prima di iniziare: Errori e debug

- **Errori di sintassi**

Codice

```
1    print("Questo è un errore di sintassi)
```

Console output

```
    print("Questo è un errore di sintassi)
          ^
SyntaxError: unterminated string literal (detected at line 1)

Process finished with exit code 1
```

Prima di iniziare: Errori e debug

- **Errori di indentazione**

Codice

```
1   x = 10
2   if x > 10:
3   print("Questo è un errore di indentazione")
```

Console output

```
    print("Questo è un errore di indentazione")
    ^
IndentationError: expected an indented block after 'if' statement on line 2

Process finished with exit code 1
```

Prima di iniziare : Errori e debug

- **Errori runtime**

Codice

```
1  a = 0
2  print(10/a)
```

Console output

```
    print(10/a)
ZeroDivisionError: division by zero

Process finished with exit code 1
```

- **Errori semantici:** Un programma con un errore semantico verrà eseguito correttamente, nel senso che l'interprete non genererà nessun messaggio di errore. Tuttavia, il programma si comporterà diversamente da quanto ci aspettiamo

Prima di iniziare : Errori e debug

- Il debug è un operazione fondamentale nella programmazione. Durate lo sviluppo di software di complessità significativa è impensabile che nel codice non si generino bug o errori
- Se l'esecuzione di uno script genera degli errori è opportuno leggerli attentamente. I messaggi di errore dell'interprete Python sono, in generale, descrittivi e danno numerosi suggerimenti sull'origine e sul tipo di errore:
  - Viene segnalata la riga nello script che ha generato l'errore
  - Viene segnalato lo stack delle chiamate dei metodi (nel caso di software più complessi) per risalire alla fonte dell'errore
  - Viene indicato il tipo di eccezione (es. divisione per zero, accesso a indici fuori dall'array etc...)
- Il debug di errori semantici è, in generale, **il più difficile**, in quanto non viene segnalato dall'interprete

Oggetti e Variabili

- **Oggetto:** un oggetto rappresenta un dato (un numero, del testo, un insieme di oggetti, ...) che viene gestito da un software
- In Python qualsiasi cosa è un oggetto, ovvero un'istanza di una certa classe: OOP
- Gli oggetti sono, in generale, caratterizzati da:
    - **Tipo:** codifica la classe dell'oggetto, ovvero la classe da cui è stato istanziato l'oggetto. Useremo in generale la parola tipo dell'oggetto o classe in maniera intercambiabile
    - **Valore:** il dato stesso
    - **Id:** l'identità univoca dell'oggetto. Può essere inteso come il riferimento alla zona di memoria allocata per l'oggetto

Oggetti e Variabili

- **Esempio: Valore di un Oggetto**

Codice

```
1  # 42 è un oggetto di tipo int, con la funzione print visualizziamo il suo valore
2  print(42)
3  # 42.3 è un oggetto di tipo float, con la funzione print visualizziamo il suo valore
4  print(42.3)
5  # Per verificare che sono effettivamente oggetti usiamo la funzione isinstance
6  print(isinstance(42, int))
7  print(isinstance(42.3, int))
```

Console output

```
42
42.3
True
False
```

Oggetti e Variabili

- **Esempio: Tipo di un Oggetto**

Codice

```
1  # Per accedere al tipo di un oggetto si può utilizzare la funzione built-in type()
2  print(type(42))
3  print(type(42.3))
4  print(type("Hello world"))
```

Console output

```
<class 'int'>
<class 'float'>
<class 'str'>
```

Oggetti e Variabili

- **Esempio: Id di un Oggetto**

Codice

```
2   print(id(42))
3   print(id(42.3))
4   print(id("Hello world"))
```

Console output

```
2724820944400
2724821981808
2724826038000
```

N.B.: Ad ogni esecuzione del programma, l'id assegnato, in generale, cambia (anche un oggetto viene istanziato con lo stesso valore

Oggetti e Variabili

- Gli esempi nelle slide precedenti creavano degli oggetti direttamente nella funzione *print()*
- Gli oggetti venivano quindi istanziati (allocati in una zona di memoria) e le loro caratteristiche venivano visualizzate nella console tramite la funzione *print()*
- Tuttavia, dopo l'esecuzione della funzione *print()* non c'era più modo di utilizzare l'oggetto in quanto non erano presenti strumenti per accedervi o farvi riferimento
- Abbiamo quindi bisogno di uno strumento che ci permetta di accedere all'oggetto anche dopo averlo istanziato

Oggetti e Variabili

- **Variabile:** Le variabili sono riferimenti ad oggetti. Sono come dei nomi per gli oggetti a cui fanno riferimento
- La variabili puntano alla zona di memoria dell'oggetto a cui fanno riferimento
    - Il valore di una variabile è il valore dell'oggetto a cui fa riferimento
    - Le funzioni *type()* e *id()* chiamate su una variabile restituiscono il tipo e l'id dell'oggetto a cui fanno riferimento
- Le variabili in Python hanno lo stesso ruolo dei puntatori del linguaggio C: tutte le variabili Python sono riferimenti ad oggetti istanziati in memoria

Oggetti e Variabili

Codice

```
1  a = 50.3
2  # chiamando la funzione print() su a, visualizziamo
3  # a schermo il valore dell'oggetto a cui fa riferimento
4  print(a)
5
6  #Le funzioni type() e id() restituiscono il tipo e l'id dell'oggetto
7  # a cui fa riferimento la variabile a
8  print(type(a))
9  print(id(a))
```

Console output

```
50.3
<class 'float'>
2211666453296
```

Oggetti e Variabili

- **Alcune indicazioni per la scelta dei nomi delle variabili**
  - Possono contenere solo lettere numeri e il carattere speciale «Underscore», ovvero «_»
  - Non possono iniziare con un numero
  - Alcune parole non possono essere utilizzate per dare nomi a variabili perché sono riservate nel linguaggio di programmazione

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

Oggetti e Variabili

- **Operatore di assegnazione**
  - L’operatore «=» assegna un oggetto a una variabile
  - Se l’oggetto era già stato creato, la variabile semplicemente punterà alla zona di memoria di quell’oggetto
  - Se l’oggetto non esisteva ancora, prima viene istanziato in memoria e successivamente assegnato alla variabile

Oggetti e Variabili

- **Operatore di assegnazione**

Codice

```
# Creiamo un oggetto con valore '10' e assegnamolo alla variabile 'a'
a = 10
print(id(a))
# Creiamo un oggetto con valore '20' e assegnamolo alla variabile 'b'
b = 20
print(id(b))
# Assegnamo l'oggetto referenziato da 'a' alla variabile 'b' e verifichiamo che
#gli id delle due variabili sono gli stessi dato che puntano allo stesso oggetto
b = a
print(id(a))
print(id(b))
print(a)
print(b)
```

Oggetti e Variabili

- **Operatore di assegnazione**

Codice

```
# Creiamo un oggetto con valore '10' e assegnamolo alla variabile 'a'
a = 10
print('id di a:', id(a))
# Creiamo un oggetto con valore '20' e assegnamolo alla variabile 'b'
b = 20
print('id di b:',id(b))
# Assegnamo l'oggetto referenziato da 'a' alla variabile 'b' e verifichiamo che
#gli id delle due variabili sono gli stessi dato che puntano allo stesso oggetto
b = a
print('id di a:',id(a))
print('id di b:',id(b))
print('valore di a:',a)
print('valore di b:',b)
```

Console output

```
id di a: 2039731061264
id di b: 2039731061584
id di a: 2039731061264
id di b: 2039731061264
valore di a: 10
valore di b: 10
```

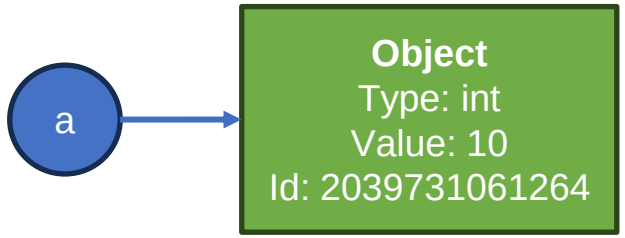
Oggetti e Variabili

- **Operatore di assegnazione**

Codice

```
1   # Creiamo un oggetto con valore '10' e assegnamolo alla variabile 'a'
2   a = 10
3   print('id di a:', id(a))
4   # Creiamo un oggetto con valore '20' e assegnamolo alla variabile 'b'
5   b = 20
6   print('id di b:',id(b))
7   # Assegnamo l'oggetto referenziato da 'a' alla variabile 'b' e verifichiamo che
8   #gli id delle due variabili sono gli stessi dato che puntano allo stesso oggetto
9   b = a
10  print('id di a:',id(a))
11  print('id di b:',id(b))
12  print('valore di a:',a)
13  print('valore di b:',b)
```

Riga 2



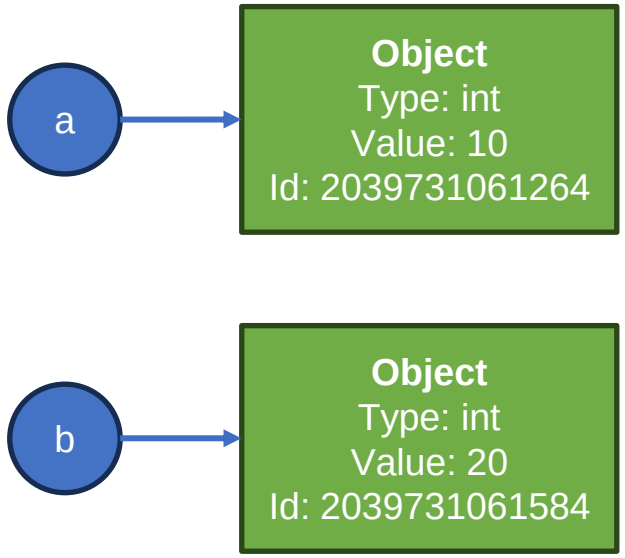
Oggetti e Variabili

- **Operatore di assegnazione**

Codice

```
1   # Creiamo un oggetto con valore '10' e assegnamolo alla variabile 'a'
2   a = 10
3   print('id di a:', id(a))
4   # Creiamo un oggetto con valore '20' e assegnamolo alla variabile 'b'
5   b = 20
6   print('id di b:',id(b))
7   # Assegnamo l'oggetto referenziato da 'a' alla variabile 'b' e verifichiamo che
8   #gli id delle due variabili sono gli stessi dato che puntano allo stesso oggetto
9   b = a
10  print('id di a:',id(a))
11  print('id di b:',id(b))
12  print('valore di a:',a)
13  print('valore di b:',b)
```

Riga 5



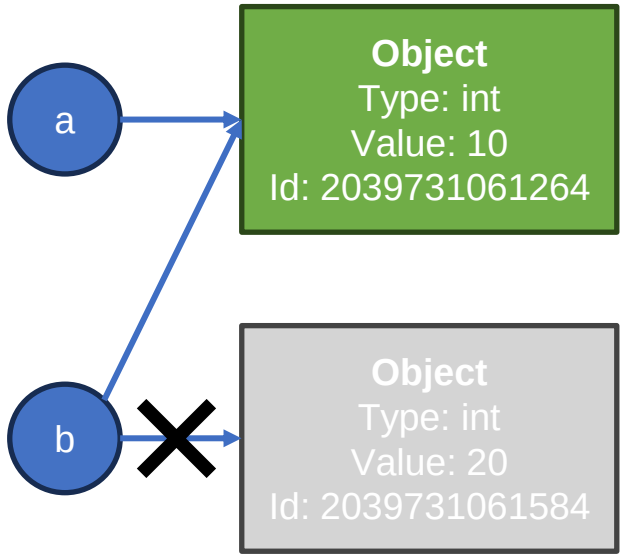
Oggetti e Variabili

- **Operatore di assegnazione**

Codice

```
1   # Creiamo un oggetto con valore '10' e assegnamolo alla variabile 'a'
2   a = 10
3   print('id di a:', id(a))
4   # Creiamo un oggetto con valore '20' e assegnamolo alla variabile 'b'
5   b = 20
6   print('id di b:',id(b))
7   # Assegnamo l'oggetto referenziato da 'a' alla variabile 'b' e verifichiamo che
8   #gli id delle due variabili sono gli stessi dato che puntano allo stesso oggetto
9   b = a
10  print('id di a:',id(a))
11  print('id di b:',id(b))
12  print('valore di a:',a)
13  print('valore di b:',b)
```

Riga 9



Python Garbage Collector

- Python ha una gestione automatica della memoria (in C++, ad esempio, il programmatore può controllare più nel dettaglio l'allocazione della memoria per gli oggetti, ad esempio, con gli operatori *new* e *delete*)
- Come anticipato, le variabili si comportano come i puntatori del linguaggio C
- Quando assegniamo un oggetto a una variabile, se questo ancora non esiste, viene creato in una zona di memoria
- La variabile viene fatta puntare a quella zona (*reference*)

- Cosa accade agli oggetti che non sono più referenziati da variabili?
- Python ha un contatore interno che memorizza il numero di variabili che puntano ad un oggetto.
- Quando questo numero diventa zero, il **garbage collector** rimuove l'oggetto liberando la memoria
- In Python questo è gestito automaticamente: non è possibile cancellare «manualmente» un oggetto da una zona di memoria

Oggetti Mutabili e Immutabili

- Gli oggetti posson essere di due tipologie: **mutabili e immutabili**
- Se un oggetto è mutabile, è possibile accedervi (successivamente alla sua creazione) tramite la variabile che lo referenzia per cambiarne le caratteristiche, **mantenendo la stessa zona di memoria in cui era stato creato**
- Se un oggetto è immutabile, non è possibile cambiarne le caratteristiche dopo averlo creato. Un tentativo di alterare, ad esempio, il valore di un oggetto immutabile tramite la variabile associata risulterebbe nella creazione di un nuovo oggetto (assegnato ad una **nuova zona di memoria**), valorizzato con il nuovo dato. La variabile punterebbe quindi al nuovo oggetto e verrebbe quindi «disconnessa» dal precedente oggetto. A questo punto quest'ultimo, in caso non fosse più referenziato da alcuna variabile, verrebbe quindi cancellato dal garbage collector

Oggetti Mutabili e Immutabili

Type	Meaning	Domain	Mutable?
`bool`	Condition	`True`, `False`	No
`int`	Integer	$\mathbb{Z}$	No
`float`	Rational	$\mathbb{Q}$ (more or less)	No
`str`	Text	Text	No
`list`	Sequence	Collections of things	Yes
`tuple`	Sequence	Collections of things	No
`dict`	Map	Maps between things	Yes

Oggetti Mutabili e Immutabili

- **Esempio: oggetto mutabile**

Codice

```
1   # Le liste Python sono oggetti mutabili
2   # Istanziamo una lista con 3 numeri e visualizziamo il suo id
3   lista1 = [2, 3, 4]
4   print("valore di lista1: ", lista1)
5   print("id di lista1: ", id(lista1))
6   # Alteriamo il valore dell'oggetto referenziato da lista1
7   # Aggiungiamo alcuni numeri alla lista
8   lista1 += [5, 6, 7]
9   # cambiamo il valore del primo elemento della lista
10  lista1[0] = 40
11  print(lista1)
12  #L'id della variabile associata a lista1 (e quindi dell'oggetto) è invariato
13  print("valore di lista1: ", lista1)
14  print("id di lista1: ", id(lista1))
```

Console output

```
valore di lista1:  [2, 3, 4]
id di lista1:  2398104497472
[40, 3, 4, 5, 6, 7]
valore di lista1:  [40, 3, 4, 5, 6, 7]
id di lista1:  2398104497472
```

Oggetti Mutabili e Immutabili

- **Esempio: oggetto immutabile**

Codice

```
1   # Gli oggetti di tipo int sono immutabili
2   a = 10
3   print("valore di a: ", a)
4   print("Tipo di a: ", type(a))
5   print("Id di a: ", id(a))
6   # Se proviamo a cambiare il valore di 'a', non verrà cambiato il valore
7   # dell'oggetto esistente, ma verrà creato un nuovo oggetto referenziato da 'a'
8   a = 20
9   print("valore di a: ", a)
10  print("Tipo di a: ", type(a))
11  print("Id di a: ", id(a))
```

Console output

```
valore di a:  10
Tipo di a:  <class 'int'>
Id di a:  1602287239696
valore di a:  20
Tipo di a:  <class 'int'>
Id di a:  1602287240016
```

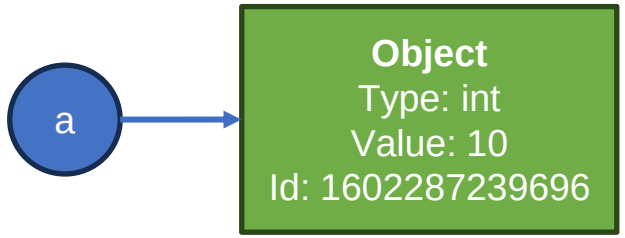
Oggetti Mutabili e Immutabili

- **Esempio: oggetto immutabile**

Codice

```
1   # Gli oggetti di tipo int sono immutabili
2   a = 10
3   print("valore di a: ", a)
4   print("Tipo di a: ", type(a))
5   print("Id di a: ", id(a))
6   # Se proviamo a cambiare il valore di 'a', non verrà cambiato il valore
7   # dell'oggetto esistente, ma verrà creato un nuovo oggetto referenziato da 'a'
8   a = 20
9   print("valore di a: ", a)
10  print("Tipo di a: ", type(a))
11  print("Id di a: ", id(a))
```

Riga 2



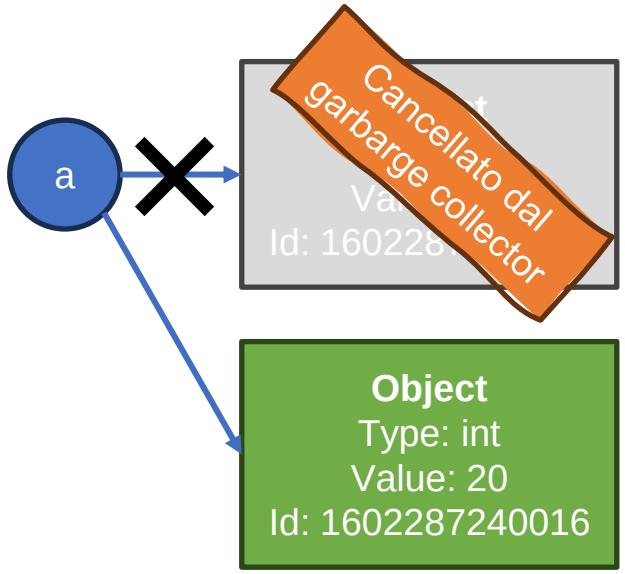
Oggetti Mutabili e Immutabili

- **Esempio: oggetto immutabile**

Codice

```
1   # Gli oggetti di tipo int sono immutabili
2   a = 10
3   print("valore di a: ", a)
4   print("Tipo di a: ", type(a))
5   print("Id di a: ", id(a))
6   # Se proviamo a cambiare il valore di 'a', non verrà cambiato il valore
7   # dell'oggetto esistente, ma verrà creato un nuovo oggetto referenziato da 'a'
8   a = 20
9   print("valore di a: ", a)
10  print("Tipo di a: ", type(a))
11  print("Id di a: ", id(a))
```

Riga 8



Oggetti Mutabili e Immutabili

- **Un altro esempio sugli oggetti immutabile**

Codice

```
1   # Gli oggetti di tipo int sono immutabili
2   a = 10
3   print("valore di a: ", a)
4   print("Tipo di a: ", type(a))
5   print("Id di a: ", id(a))
6   # Anche nel caso in cui il valore di 'a' vienga modificato a partire dal
7   # precedente valore, verrà comunque creato un nuovo oggetto
8   a = a + 1
9   print("valore di a: ", a)
10  print("Tipo di a: ", type(a))
11  print("Id di a: ", id(a))
```

Console output

```
valore di a:  10
Tipo di a:  <class 'int'>
Id di a:  1522062721552
valore di a:  11
Tipo di a:  <class 'int'>
Id di a:  1522062721584
```

Python e tipizzazione

- Negli esempi precedenti, quando abbiamo creato degli oggetti non abbiamo mai specificato il loro tipo
- Tuttavia, utilizzando la funzione *type()* il tipo dell'oggetto veniva correttamente visualizzato. Come è possibile?
- Python è un linguaggio **debolmente tipizzato**, ovvero non è necessario specificare la classe dell'oggetto che stiamo per creare.
- Sarà l'interprete Python che automaticamente determinerà la classe più adeguata da utilizzare per istanziare l'oggetto sulla base del valore che vogliamo assegnare
- In questo senso Python è diverso da molti altri linguaggi di programmazione, in cui è necessario specificare la classe dell'oggetto quando viene istanziato

Python e tipizzazione

Codice

```python
# Creando un oggetto con valore '10', Python assegnerà
# automaticamente il tipo 'int'
a = 10
print('Tipo di a: ', type(a))
# Creando un oggetto con valore '10.5', Python assegnerà
# automaticamente il tipo 'float'
b = 10.5
print('Tipo di b: ', type(b))
# Creando un oggetto con valore '10.0', Python assegnerà
# automaticamente il tipo 'float'
c = 10.0
print('Tipo di c: ', type(c))
# Creando un oggetto con valore 'Hello', Python assegnerà
# automaticamente il tipo 'str' (stringa)
d = "Hello"
print('Tipo di d: ', type(d))

```

Console output

```
Tipo di a:  <class 'int'>
Tipo di b:  <class 'float'>
Tipo di c:  <class 'float'>
Tipo di d:  <class 'str'>
```

Tipi numerici e operazioni

- I tipi numerici fondamentali in Python sono *int* e *float*
- Tipi numerici possono essere manipolati con gli **operatori numerici**

Simbolo	**Operazione**
+, -, *	Somma, differenza, prodotto
/	Divisione
//	Divisione intera
%	Modulo
**	Potenza

Tipi numerici e operazioni

Codice

```
a = 5
b = 10
c = 7
# Somma, sottrazione, prodotto e divisione
somma = a+b
sottrazione = a-b
prodotto = a*c
divisione = c/a
print("Somma: ", somma)
print("Sottrazione: ", sottrazione)
print("Prodotto: ", prodotto)
print("Divisione: ", divisione)
# Divisione intera e modulo
div_intera = c//a
modulo = c%a
print("Divisione intera: ", div_intera)
print("Modulo: ", modulo)
#Potenza
potenza = b**a
print("Potenza: ", potenza)
```

Console output

```
Somma:  15
Sottrazione:  -5
Prodotto:  35
Divisione:  1.4
Divisione intera:  1
Modulo:  2
Potenza:  100000
```

Ordine delle operazioni

Regola **PEMDAS,** dalla priorità più alta alla più bassa:

1. **P:** Le Parentesi hanno la precedenza più elevata e possono essere utilizzate per forzare l'ordine di valutazione come desiderato
2. **E:** l'elevamento a potenza ha la priorità successiva
3. **MD:** Moltiplicazione e Divisione (incluse divisione intera e modulo) hanno la stessa priorità
4. **AS**: anche addizione e sottrazione hanno la stessa priorità

Nel caso di operatori con la stessa priorità, l'ordine di valutazione è effettuato da sinistra verso destra. Ad esempio 4/2*3: in questo caso prima viene effettuata la divisione (con risultato 2) e poi la moltiplicazione (con risultato 6)

Ordine delle operazioni

Codice

```
a = 5
b = 10
c = 2

res1 = a+a*b
res2 = a**2%a+b
res3 = ((b*3)//a**c%c)**b

print(res1)
print(res2)
print(res3)
```

Console output

```
55
10
1
```

Assegnazione composta

In Python è possibile applicare un operazione ad una variabile e riassegnare alla stessa variabile il risultato (che, nel caso di tipi immutabili, sarà un nuovo oggetto!)

Codice

```
1   a = 3
2   print("a: ", a)
3   a += 4
4   print("a: ", a)
5
6   b = 2
7   print("b: ", b)
8   b *= 4
9   print("b: ", b)
10
11  c = 4
12  print("c: ", c)
13  c **=b
14  print("c: ", c)
```

Console output

```
a:  3
a:  7
b:  2
b:  8
c:  4
c:  65536
```

Conversione del tipo di oggetto

La conversione del tipo di un oggetto può essere:

- **Automatica:** l'interprete cambia il tipo numerico a seguito di un'operazione
- **Manuale (costruttori di classe):** utilizzando i costruttori *int()* e *float()* che prendono come parametro un valore o una variabile che fa riferimento ad un oggetto e istanziano un nuovo oggetto di tipo *int* o *float*, rispettivamente. È un operazione simile al **casting**.

N.B. Nel caso il costruttore *int*() viene utilizzato su un oggetto float (cast da float a int), i decimali vengono troncati all'intero inferiore più vicino.

ATTENZIONE: informalmente si tende a dire «cambiare il tipo di un oggetto», in realtà, come abbiamo visto prima, nel caso di oggetti immutabili (come int e float) non stiamo cambiando il tipo dell'oggetto, ma ne stiamo creando uno nuovo con un nuovo tipo!

Conversione del tipo di oggetto

Codice

```
a = 5
b = 3
c = 4.6
print("Tipo di a: ", type(a))
print("Id di a: ", id(a))
print("Tipo di b: ", type(b))
print("Id di b: ", id(b))
print("Tipo di c: ", type(c))
print("Id di c: ", id(c))
#Conversione automatica del tipo
a = a + c
print("Valore di a dopo l'operazione: ", a)
print("Tipo di a dopo l'operazione: ", type(a))
print("Id di a dopo l'operazione: ", id(a))
#Cast
b = float(b)
print("Valore di b dopo il cast: ", b)
print("Tipo di b dopo il cast: ", type(b))
print("Id di b dopo il cast: ", id(b))
c = int(c)
print("Valore di c dopo il cast: ", c)
print("Tipo di c dopo il cast: ", type(c))
print("Id di c dopo il cast: ", id(c))
```

Console output

```
Tipo di a:  <class 'int'>
Id di a:  2115043721584
Tipo di b:  <class 'int'>
Id di b:  2115043721520
Tipo di c:  <class 'float'>
Id di c:  2115044762416
Valore di a dopo l'operazione:  9.6
Tipo di a dopo l'operazione:  <class 'float'>
Id di a dopo l'operazione:  2115044759952
Valore di b dopo il cast:  3.0
Tipo di b dopo il cast:  <class 'float'>
Id di b dopo il cast:  2115044759792
Valore di c dopo il cast:  4
Tipo di c dopo il cast:  <class 'int'>
Id di c dopo il cast:  2115043721552
```

Tipo Booleano

- In Python, il tipo di oggetto **bool** è caratterizzato da due possibili valori: **True** e **False** (entrambi con le iniziali maiuscole)
- Gli operatori per i tipi booleani sono **and, or** e **not**
- È possibile anche applicare gli operatori che sono stati introdotti per i tipi numerici (+, -, /, *...) in quando Python associa al valore True e False i valori numeri 1 e 0 rispettivamente. Quando l'interprete applica un operatore usato normalmente per tipi numerici a variabili booleane, converte automaticamente True e False in 1 e 0 e poi applica l'operazione
- Oggetti booleani possono anche essere creati come risultato di operazioni di comparazione (es. maggiore di, minore di, uguale a o diverso da)

Tipo Booleano

- Tabelle di verità

a	not a
False	True
True	False

a	b	a and b	a or b
False	False	False	False
True	False	False	True
False	True	False	True
True	True	True	True

- Comparatori

Istruzione Python	Risultato
a!=b	True se e solo se a $\neq$ b
a==b	True se e solo se a = b
a<b	True se e solo se a<b
a>b	True se e solo se a>b
a<=b	True se e solo se a $\leq$ b
a>=b	True se e solo se a $\geq$ b

Tipo Booleano

Codice

```python
a = True
b = False
print("a and b: ", a and b)
print("a or b: ", a or b)
print("not a: ", not a)

# operatori +, * applicati a booleani
print("a + b: ", a+b)
print("a * b: ", a*b)

# Comparatori
a = 10
b = 20
print("a uguale a b: ", a==b)
print("a diverso da b: ", a!=b)
print("a maggiore o uguale da b: ", a>=b)
print("a minore o uguale da b: ", a<=b)
```

Console output

```
a and b:  False
a or b:  True
not a:  False
a + b:  1
a * b:  0
a uguale a b:  False
a diverso da b:  True
a maggiore o uguale da b:  False
a minore o uguale da b:  True
```

Tipo Booleano

- Gli operatori booleani vengono valutati dopo gli operatori numerici:

Simbolo	Operazione
**	Potenza
* / // %	Prodotto, divisione, divisione intera, modulo
+ -	Addizione e sottrazione
<= < > >=	Maggiore (uguale), minore (uguale)
== !=	Uguale, diverso
not or and	Operatori logici

Funzioni in Python: anticipazioni

- Abbiamo già utilizzato delle funzioni Python nelle slide precedenti: *print(), id(), type()*
- In Python, come in altri linguaggi, una funzione è **invocata** utilizzando il nome della funzione, seguito tra parentesi dalla lista dei suoi parametri.

```
result = f(par1, par2, ....)
```

- Le funzioni possono restituire («*ritornare*») dei risultati che possono essere assegnati ad una variabile.
- Una funzione può anche non restituire nessun risultato (analogo delle funzioni **void** in altri linguaggi)
- In Python esistono molte funzioni **built-in** (ovvero disponibili nell'interprete di base, senza necessità di importare moduli esterni)

Funzioni in Python: anticipazioni

abs()	Ritorna il valore assoluto di un numero
max()	Ritorna il massimo tra due o più valori (prende in input un numero arbitrario di parametri)
min()	Ritorna il minimo tra due o più valori (prende in input un numero arbitrario di parametri)
round()	Arrotonda un numero floating point ad un numero desiderato di cifre
input()	Permette all'utente di inserire da console un input che verrà poi salvato su una variabile. L'esecuzione dello script entra in pausa finché l'utente non preme invio. La funzione può prendere un parametro opzionale (una stringa) da mostrare in console per spiegare il tipo di input atteso

Funzioni in Python: anticipazioni

Codice

```python
# abs() function
absolute_value = abs(-5.0)
print("Valore assoluto di -5.0: ", absolute_value)

# max() function
max_value = max(-5.0, 1.0, -10.5, 40, 2, 300, 20)
print("Massimo: ", max_value)

# round() function
round_value = round(2.49958373, 4)
print("Valore arrotondato: ", round_value)

# input() function - \n è usato per fare in modo che
# il numero venga inserito in una nuova linea della console
valore_inserito = input("Inserisci un numero: \n")
print("Valore inserito: ", valore_inserito)
```

Console output

```
Valore assoluto di -5.0:  5.0
Massimo:  300
Valore arrotondato:  2.4996
Inserisci un numero:
40
Valore inserito:  40
```

Metodi in Python: anticipazioni

- Un metodo è una funzione di istanza, di classe o statica, ovvero è messo a disposizione da una certa classe (o tipo) di oggetto e viene invocato:
  - Attraverso l'istanza dell'oggetto stesso -> metodo di istanza
  - Attraverso un riferimento alla classe -> metodi di classe e metodi statici.
- Come le funzioni, i metodi possono ritornare valori oppure no (void)
  - **Metodo di istanza**: result = object.function(par1, par2, ...) *(object deve essere stato istanziato precedentemente)*
  - **Metodo di classe (o statico)**: result = Class.function(par1, par2, ...)

Tipo Stringa

- Le stringhe sono oggetti **immutabili:** ogni operazione che modifica le caratteristiche di un oggetto stringa (attraverso la variabile che lo referenzia) risulteranno nella creazione di un nuovo oggetto
- La stringa vuota è ancora considerata una stringa

Codice

```
1  first_string = 'Stringa definita tra apici singoli'
2  second_string = "Stringa definita tra virgolette"
3  third_string = """Stringa definita tra tre virgolette"""
4
5  print(first_string)
6  print(second_string)
7  print(third_string)
```

Console output

```
Stringa definita tra apici singoli
Stringa definita tra virgolette
Stringa definita tra tre virgolette
```

Tipo Stringa

- Alcuni caratteri non possono essere rappresentati esplicitamente all'interno di stringhe. È necessario utilizzare il backslash come prefisso

\\	Backslash
\n	nuova riga (new line)
\t	Tab
\'	Singolo apice
\"	Doppi apici
\xxx	Carattere unicode (esadecimale)

Tipo Stringa

Codice

```
1  print("Stringa in \n due righe")
2  print("Stringa con \t tabulazione")
3  print("Stringa con \\ backslash")
4  print("Stringa con carattere unicode \x00")
```

Console output

```
Stringa in
 due righe
Stringa con      tabulazione
Stringa con \ backslash
Stringa con carattere unicode ␀
```

Conversione tra stringa e tipi numerici

- Funzione *str(n)*: converte un numero in una stringa
- Funzione *int(s)*: converte una stringa in un intero
- Funzione *float(s)*: converte una stringa in un float

Codice

```
1   #conversione da numero a stringa
2   int_number = 20
3   int2str = str(int_number)
4   print(int_number, type(int_number))
5   print(int2str, type(int2str))
6
7   #conversione da stringa a float
8   string_number = "3.14"
9   str2float = float(string_number)
10  print(string_number, type(string_number))
11  print(str2float, type(str2float))
```

Console output

```
20 <class 'int'>
20 <class 'str'>
3.14 <class 'str'>
3.14 <class 'float'>
```

Operatori su oggetti stringa

Return value	**Operatore**	**Descrizione**
int	len(str)	**Funzione** che restituisce la lunghezza (numero dei caratteri) di una stringa
str	str + str	Concatena due stringhe
str	str*int	Replica una stringa un numero di volte specificate da *int*
bool	str in str	Controlla se una stringa è presente in un'altra stringa
str	str[int]	Restituisce il carattere alla posizione specificata dall'indice *int*
str	str[int1:int2]	Restituisce la sotto stringa compresa tra gli indici (incluso) e *int2* (escluso)
bool	==, !=	Operatori per controllare se due stringhe sono uguali o diverse
bool	<, >	Operatori per verificare l'ordine lessicografico di due stringhe

Operatori su oggetti stringa: concat e len()

Codice

```
1  concat_string = "una" + " " + "stringa" + " " + "concatenata"
2  print("Stringa concatenata: ", concat_string)
3  print("Lunghezza della stringa concatenata: ", len(concat_string))
4
5  rep_string = "ciao, "*3
6  print("Stringa replicata: ", rep_string)
7  print("la stringa replicata ha ", len(rep_string), " caratteri")
```

Console output

```
Stringa concatenata:  una stringa concatenata
Lunghezza della stringa concatenata:  23
Stringa replicata:  ciao, ciao, ciao,
la stringa replicata ha  18  caratteri
```

Operatori su oggetti stringa: concat e len()

- Attenzione! Non si può concatenare una stringa con un intero (o float) senza un cast

Codice

```
1  int_value=10
2  print("Valore: " + int_value)
```

Console output

```
Traceback (most recent call last):
  File "C:\Users\sin_c\OneDrive - Università degli Stu
    print("Valore: " + int_value)
TypeError: can only concatenate str (not "int") to str
```

Codice

```
1  int_value=10
2  print("Valore: " + str(int_value))
```

Console output

```
Valore: 10
```

Operatori su oggetti stringa: “In”

- Attenzione! Non si può concatenare una stringa con un intero (o float) senza un cast

Codice

```
1  string1 = "Un corso su Python"
2
3  print("p" in string1)        False
4  print("corso" in string1)    True
5  print("CORSO" in string1)    False
6  print("" in string1)         True
7  print(" " in string1)        True
8  print(string1 in string1)    True
9  print("joun" in string1)     False
```

Console output

```
False
True
False
True
True
True
False
```

Operatori su oggetti stringa: “==, !=, <, >”

Codice

```
1  str_a = "Python"
2  str_b = "Pythom"
3  str_c = "Qython"
4
5  print(str_a == str_b)
6  print(str_a != str_b)
7  print(str_a < str_b)
8  print(str_a > str_c)
```

Console output

```
False
True
False
False
```

Operatori su oggetti stringa: indexing e slicing

- È possibile accedere al carattere di una stringa corrispondente all'indice i utilizzando s[i] -> **indexing**
- Si parla invece di **slicing** per indicare l'estrazione di una sottostringa:
  - s[start:end] -> restituisce la sottostringa compresa tra *start (incluso)* e *end (escluso)*
  - s[:end] -> restituisce la sottostringa compresa tra l'inizio della stringa s e *end (escluso)*
  - s[start:] -> restituisce la sottostringa compresa tra *start (incluso)* e la fine della stringa s

Operatori su oggetti stringa: indexing e slicing

- In Python l'indexing e lo slicing possono anche essere effettuati con indici negativi a partire dall'ultimo carattere della stringa

0	1	2	3	4	5
P	Y	T	H	O	N
-6	-5	-4	-3	-2	-1

Operatori su oggetti stringa: indexing e slicing

- In Python l'indexing e lo slicing possono anche essere effettuati con indici negativi a partire dall'ultimo carattere della stringa

Codice

```
1  string = "Python"
2
3  print(string[0])
4  print(string[-1])
5  print(string[:4])
6  print(string[-5:-3])
7  print(string[2:])
```

Console output

```
P
n
Pyth
yt
thon
```

Operatori su oggetti stringa: indexing

Codice

```
1  string = "Python"
2
3  print(string[8:])
4
5
6
7
8  print(string[8])
```

Console output

(stringa vuota)

```
Traceback (most recent call last):
  File
    print(string[8])
IndexError: string index out of range
```

- **Attenzione**! Accedere a indici fuori della stringa tramite slicing non genera un errore, ma restituisce una stringa vuota
- Invece, accedere al singolo carattere con l'indexing genera un runtime error -> *index out of range*

Metodi della classe stringa

Return value	Metodo	Descrizione
str	str.upper() - str.lower()	Restituisce la stringa con tutti i caratteri maiuscoli-minuscoli
str	str1.strip(str2) - str1.lstrip(str2) - str1.rstrip(str2)	Rimuove dall'inizio (*lstrip()*), dalla fine (*rstrip()*) o da entrambi i lati di *str1* la stringa passata come parametro (*str2*). Se il parametro non è specificato, di default assume il carattere «spazio»
str	str1.replace(str2, str3)	Cerca la sottostringa *str2* in *str1* e la sostituisce con *str3*
bool	str.startswith(str1) – str.endswith(str1)	Restituisce True se *str* inizia/finisce con la stringa *str1*
int	str.find(str1) – str.rfind(str1)	Cerca in *str* la stringa *str1* e se la trova ritorna l'indice di inizio della prima occorrenza *str1* a partire da sinistra (*find()*) o da destra (*rfind()*). Restituisce -1 se non trova *str1*
int	str.count(str1)	Conta il numero di occorrenze di *str1* in *str*

Metodi della classe stringa

Codice

```
1  str_a = "111111Python111111"
2  print("Risultato di strip: ", str_a.strip("1"))
3  print("Risultato di lstrip: ", str_a.lstrip("1"))
4  print("Risultato di rstrip: ", str_a.rstrip("1"))
5
6  str_b = "Python, un programma Python!"
7  print("Risultato di startswith 'Un':", str_b.startswith("Un"))
8  print("Risultato di startswith 'Python':", str_b.startswith("Python"))
9  print("Risultato di find 'Python':", str_b.find("Python"))
10 print("Risultato di rfind 'Python':", str_b.rfind("Python"))
11 print("Risultato di count 'Python':", str_b.count("Python"))
```

Console output

```
Risultato di strip:  Python
Risultato di lstrip:  Python111111
Risultato di rstrip:  111111Python
Risultato di startswith 'Un': False
Risultato di startswith 'Python': True
Risultato di find 'Python': 0
Risultato di rfind 'Python': 21
Risultato di count 'Python': 2
```

Tipo Stringa

- Come già anticipato, le stringhe sono oggetti **immutabili.** Tutte le operazioni e i metodi che abbiamo visto precedentemente e che restituiscono stringhe non modificano l'oggetto esistente su cui sono stati chiamati, ma ne creano uno nuovo
- Per lo stesso motivo, non è possibile modificare un carattere di una stringa accedendovi tramite indexing o slicing

Codice

```
1  str_a = "Un programma Python"
2  str_a[0] = "C"
```

Console output

```
Traceback (most recent call last):
  File "
    str_a[0] = "C"
TypeError: 'str' object does not support item assignment
```

Liste

- Le liste in Python sono sequenze ordinate di oggetti (anche di tipo diverso)
- Le liste sono oggetti **mutabili**
- La lista vuota è considerata un oggetto della classe lista
- È possibile istanziare una lista attraverso le parentesi quadre
- Le liste sono ordinate:
  - [4, 5, 6] != [6, 5, 4]
- Le liste non sono insiemi
  - [5, 5, ‘ciao’, ‘ciao’] != [5, ‘ciao’]

Codice

```
1   #Lista di numeri interi
2   int_list = [4, 5, 6, 7, 8]
3
4   #Lista di stringhe
5   str_list = ["Ciao", "Python", "casa"]
6
7   #Lista di oggetti eterogenei
8   var_list = [2.0, 'Python', 2, 4.0, "Hello world"]
9
10  #Lista vuota
11  emtpy_list = []
12
13  #Lista di liste
14  list_of_list = [
15      ["Ciao", 1, 3],
16      ["Seconda lista", 5, 6]
17  ]
```

Liste: operatori

Return value	Operatore	Descrizione
int	len(list)	**Funzione** che restituisce la lunghezza (numero degli oggetti contenuti) di una lista
list	list + list	Concatena due liste (e restituisce un nuovo oggetto lista)
list	list*int	Replica una lista un numero di volte specificate da *int* (e restituisce un nuovo oggetto lista)
list	list[start:end]	Operatore di slicing per liste. Essendo le liste mutabili, può essere usato per scrivere un nuovo valore nella slice specificata da *start:end* (non viene creato un nuovo oggetto)
obj	list[index]	Operatore di indexing per liste. Essendo le liste mutabili, può essere usato per scrivere un nuovo valore nella posizione specificata da *index* (non viene creato un nuovo oggetto)
bool	obj in list	Controlla se *obj* è presente nella lista *list*
bool	==, !=	Controlla se due liste sono uguali (contengono gli stessi oggetti)
bool	is, not is	Controlla se due liste sono lo stesso oggetto

Liste: operatori

Codice

```
#Operatore 'in'
job_list = ["Engineer", "Pilot", ["Doctor", "Manager"]]

print("Engineer" in job_list)
print("Farmer" in job_list)
print("Doctor" in job_list)
print([] in job_list)
print(["Engineer", "Pilot"] in job_list)
print(["Doctor", "Manager"] in job_list)

#Operatore di indexing
print("Id di job_list: ", id(job_list))
job_list[0] = "Farmer"
print(job_list)
print("Id di job_list: ", id(job_list))
```

Console output

```
True
False
False
False
False
True
Id di job_list:  2516953412288
['Farmer', 'Pilot', ['Doctor', 'Manager']]
Id di job_list:  2516953412288
```

Liste: metodi

Return value	metodo	Descrizione
None (void)	list.append(obj)	Aggiunge un nuovo elemento alla fine della lista
None (void)	list.extend(list1)	Aggiunge gli element in *list1* alla fine di *list*
None (void)	list.insert(index, obj)	Aggiunge a *list* un nuovo elemento alla posizione specificata da *index*
None (void)	list.remove(obj)	Rimuove la prima occorrenza di *obj* in *list*
None (void)	list.reverse()	Inverte l'ordine degli elementi nella lista
None (void)	list.sort()	Ordina gli elementi della lista
int	list.count(obj)	Conta il numero di occorrenze di *obj* in *list*

Liste: metodi

Codice

```
1  list_a = [10, 20, 30, 40]
2  print("Lista iniziale: ", list_a, " e id: ", id(list_a))
3  list_a.append(50)
4  print("Lista dopo append: ", list_a, " e id: ", id(list_a))
5  list_a.extend([1, 2, 4, 5])
6  print("Lista dopo extend: ", list_a, " e id: ", id(list_a))
7  list_a.insert(3, 100)
8  print("Lista dopo insert: ", list_a, " e id: ", id(list_a))
9  list_a.remove(10)
10 print("Lista dopo remove: ", list_a, " e id: ", id(list_a))
11 list_a.sort()
12 print("Lista dopo sort: ", list_a, " e id: ", id(list_a))
```

Console output

```
Lista iniziale:  [10, 20, 30, 40]  e id:  2280326606144
Lista dopo append:  [10, 20, 30, 40, 50]  e id:  2280326606144
Lista dopo extend:  [10, 20, 30, 40, 50, 1, 2, 4, 5]  e id:  2280326606144
Lista dopo insert:  [10, 20, 30, 100, 40, 50, 1, 2, 4, 5]  e id:  2280326606144
Lista dopo remove:  [20, 30, 100, 40, 50, 1, 2, 4, 5]  e id:  2280326606144
Lista dopo sort:  [1, 2, 4, 5, 20, 30, 40, 50, 100]  e id:  2280326606144
```

- La concatenazione (+) concatena due liste e ne crea una nuova (un nuovo oggetto), mentre i metodi append() e extend() modificano la lista su cui sono chiamati

Liste: metodi

Codice

```
1  lista1 = [12, 14, 16]
2  lista2 = [12, 14, 16]
3  print(lista1 == lista2)
4  print(lista1 is lista2)
```

Console output

```
True
False
```

Liste: mutabilità

- Essendo mutabili, è possibile modificare le liste attraverso le variabili che le referenziano senza creare un nuovo oggetto
- Se non gestite correttamente possono dar luogo a comportamenti non previsti durante l'esecuzione del codice

Liste: metodi

Codice

```
1  lista1 = [12, 14, 16]
2  lista2 = [18, 20, 50]
3  lista_3 = [lista1, lista2]
4  print("Lista 3: ", lista_3)
5  lista1[0] = 0
6  print("Lista 3: ", lista_3)
7  lista_3[1][2] = 1000
8  print("Lista 2: ", lista2)
9
10 #l'assegnazione di una variabile ad un'altra non crea
11 # una nuova lista, ma fa puntare la nuove variabile
12 # all'istanza referenziata dalla prima
13 lista_new = lista1
14 lista_new.append(-3)
15 print("Lista 1: ", lista1)
16
17 # Se vogliamo creare una nuova istanza della classe lista
18 # copiando i valori da un'altra lista (creando un clone)
19 # possiamo utilizzare l'operatore di slicing
20 lista_clone = lista1[:]
21 lista_clone.append(-10)
22 print(lista1)
23 print(lista_clone)
```

Console output

```
Lista 3:  [[12, 14, 16], [18, 20, 50]]
Lista 3:  [[0, 14, 16], [18, 20, 50]]
Lista 2:  [18, 20, 1000]
Lista 1:  [0, 14, 16, -3]
[0, 14, 16, -3]
[0, 14, 16, -3, -10]
```

Liste: metodi

- Lista_a e Lista_b hanno elementi diversi o sono uguali? Hanno la stesa lunghezza?

Codice

```
1   list_a = [1, 2, 3]
2   list_a.append([7, 8, 9])
3   print(list_a)
4   print(len(list_a))
5
6
7   list_b = [1, 2, 3]
8   list_b.extend([7, 8, 9])
9   print(list_b)
10  print(len(list_b))
```

Console output

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

Liste: metodo split split() della classe string

- Il metodo *str.split(sep)* della classe string restituisce una lista con le parole contenute in *str*. Il parametro *sep* è opzionale e indica la stringa da usare come separatore tra le parole. Se non specificato, le parole sono separate rispetto agli spazi bianchi (inclusi tab, newline, etc...)

Liste: metodo split split() della classe string

Codice

```
1  text = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, " \
2         "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua."
3  list_of_words = text.split()
4  print(list_of_words)
```

Console output

```
['Lorem', 'ipsum', 'dolor', 'sit', 'amet,', 'consectetur', 'adipiscing', 'elit,', 'sed', 'do', 'eiusmod', 'tempor', 'incididunt', 'ut', 'labore', 'et', 'dolore', 'magna', 'aliqua.']
```

Tuple

- Le tuple in Python sono una classe simile alle liste, ma sono **immutabili**
- Come le liste, sono una collezione ordinata di elementi
- Come le liste possono contenere oggetti eterogenei
- Le tuple sono utili nel caso siamo certi che la sequenza di oggetti di cui abbiamo bisogno non deve cambiare nel tempo -> le tuple immutabili sono più efficienti delle liste mutabili
- Inoltre, vedremo come funzioni e metodi possono restituire più oggetti come **return value** e in questi casi è spesso conveniente utilizzare le tuple per considerare il gruppo di oggetti restituito come un'unica entità nel programma

- Un oggetto della classe tupla può essere istanziato con le parentesi tonde

Liste: metodo split split() della classe string

Codice

```
integer_tuple = (1, 5, 8, 10)

mixed_tuple = ("Ciao", 5.0, "Hello", 2, 100)

#Una tupla vuote può essere istanziata nel seguente modo
empty_tuple = ()

# Per istanziare una tupla con un solo elemento
# è necessario inserire una virgola dopo di esso

one_el_tuple = (2,)
print(type(one_el_tuple))

# Senza la virgola viene semplicemente istanziato un oggetto
# di una classe compatibile con l'elemento. Ad esempio
# di seguito stiamo istanziando un oggetto della classe int

a = (1)
print(type(a))
```

Console output

```
<class 'tuple'>
<class 'int'>
```

Tuple: operatori (analoghi alle liste)

Return value	Operatore	Descrizione
int	len(tuple)	**Funzione** che restituisce la lunghezza (numero degli oggetti contenuti) di una tuple
tuple	tuple + tuple	Concatena due tuple (e restituisce un nuovo oggetto tupla)
tuple	tuple*int	Replica una tupla un numero di volte specificate da *int* (e restituisce un nuovo oggetto tupla)
tuple	tuple[start:end]	Operatore di slicing per tuple. Essendo le tuple immutabili, può essere usato solo per leggere i valori
obj	tuple[index]	Operatore di indexing per tuple. Essendo le liste immutabili, può essere usato solo per leggere ed accedere all'oggetto
bool	obj in tuple	Controlla se *obj* è presente nella tupla *tuple*
bool	==, !=	Controlla se due tuple sono uguali (contengono gli stessi oggetti)
bool	is, not is	Controlla se due tuple sono lo stesso oggetto

Tuple: metodi

Return value	metodo	Descrizione
int	tuple.index(obj)	Restituisce l'indice della prima occorrenza di *obj* in *tuple*
int	tuple.count(obj)	Conta il numero di occorrenze di *obj* in *tuple*

Conversione tra tuple/stringhe/liste

Return value	metodo	Descrizione
list	list(obj)	Converte un oggetto in una lista
tuple	tuple(obj)	Converte un oggetto in una tupla

Codice

```
1   integer_tuple = (1, 5, 8, 10)
2
3   str_a = "1467"
4
5   tuple2list = list(integer_tuple)
6   str2list = list(str_a)
7   str2tuple = tuple(str_a)
8
9   print(tuple2list)
10  print(str2list)
11  print(str2tuple)
```

Console output

```
[1, 5, 8, 10]
['1', '4', '6', '7', 'a', 'a', 'a', 'a']
('1', '4', '6', '7', 'a', 'a', 'a', 'a')
```

Conversione tra tuple/stringhe/liste

- **Attenzione!** È estremamente sconsigliato dare nomi alle variabili utilizzando *list*, *int*, *float*, *tuple*, *string*. Python non impedisce di farlo (non da un errore di sintassi) ma in questo modo vengono sovrascritte i costruttori built-in necessari per creare i vari tipi di oggetto

Codice

```
#è opportuno non utilizzare il nome list
# per inizializzare una variabile

list = [3, 4, 5]

#A questo punto abbiamo sovrascritto la funzione
# costruttore list() e non sarà più disponibile

list_2 = list("Ciao")
```

Console output

```
Traceback (most recent call last):
  File
    list_2 = list("Ciao")
TypeError: 'list' object is not callable
```

Dizionari

- Un dizionario è una mappa tra oggetti. In particolare, è una mappa tra oggetti **chiave** e oggetti **valore.**
- I dizionari sono oggetti **mutabili**
- A differenza di tuple e liste gli oggetti nel dizionario non sono necessariamente ordinati
- I dizionari sono istanziati attraverso le parentesi graffe e l'associazione chiave valore è effettuata con i due punti
- Le **chiavi** sono uniche: una chiave può essere associata ad un solo valore
- I **valori** invece non sono unici: chiavi diverse possono mappare allo stesso valore
- Le **chiavi** devono essere **immutabili:** non si possono usare oggetti mutabili (es. liste e dizionari stessi) come chiavi. Le tuple invece possono essere usate! (sono immutabili)
- Si possono ottenere valori accedendo al dizionario tramite una chiave, ma non il viceversa

Dizionari

Codice

```python
# Istanziare un dizionario
dict_a = {
    1: "Ciao",
    2: "Python",
    3: 45.0
}
# Le chiavi possono essere oggetti immutabili
dict_b = {
    (1, 2): "primo valore",
    (1, 1): "secondo valore",
    (2, 2): "terzo valore",
}
# Ma le chiavi non possono essere oggetti mutabili
# La seguente istruzione genera un runtime error
# 'unhashable type: list'
dict_c = {
    [1, 2]: "primo valore",
    [1, 1]: "secondo valore",
    [2, 2]: "terzo valore",
}
```

Console output

```
Traceback (most recent call last):
  File "
    dict_c = {
TypeError: unhashable type: 'list'
```

Dizionari

Codice

```python
# Istanziare un dizionario
dict_a = {
    1: "Ciao",
    2: "Python",
    3: 45.0
}
# Le chiavi possono essere oggetti immutabili
dict_b = {
    (1, 2): "primo valore",
    (1, 1): "secondo valore",
    (2, 2): "terzo valore",
}

# Si può ottenere un valore accedendo al dizionario per chiave
print(dict_b[(1, 2)])
# Ma non si può accedere per valore
print(dict_a["Ciao"])
```

Console output

```
primo valore
Traceback (most recent call last):
  File "
    print(dict_a["Ciao"])
KeyError: 'Ciao'
```

Dizionari: operatori

Return value	**Operatore**	**Descrizione**
int	len(dict)	**Funzione** che restituisce la lunghezza (numero degli oggetti contenuti) di una dizionario
obj	dict[obj]	Restituisce il valore associato alla chiave *obj*
obj	dict[obj_k]=obj_v	Aggiunge o modifica il valore associato alla chiave *obj_k*
bool	obj in dict	Controlla nel dizionario è presente la chiave *obj*

Dizionari: operatori

Codice

```python
dict_a = {
    1: "Ciao",
    2: "Python",
    3: 45.0
}

# Aggiungiamo una nuova chiave con il valore associato
dict_a[4] = "Hello"
#Modifichiamone una esistente
dict_a[1]= "Hi"
#Possiamo anche aggiungere una chiave senza valore
dict_a[5]= None

print(dict_a)
print(len(dict_a))
print(1 in dict_a)
```

Console output

```
{1: 'Hi', 2: 'Python', 3: 45.0, 4: 'Hello', 5: None}
5
True
```

Dizionari: metodi

Return value	metodo	Descrizione
list	dict.keys()	Restituisce la lista delle chiavi presenti nel dizionario
list	dict.values()	Restituisce la lista dei valori presenti nel dizionario
list di tuple	dict.items()	Restituisce una lista di tuple (coppie chiave-valore) presenti nel dizionario

Dizionari: metodi

Codice

```
1  dict_a = {
2      1: "Ciao",
3      2: "Python",
4      3: 45.0
5  }
6
7  print(dict_a.keys())
8  print(dict_a.values())
9  print(dict_a.items())
```

Console output

```
dict_keys([1, 2, 3])
dict_values(['Ciao', 'Python', 45.0])
dict_items([(1, 'Ciao'), (2, 'Python'), (3, 45.0)])
```

A.D. 1308
unipg

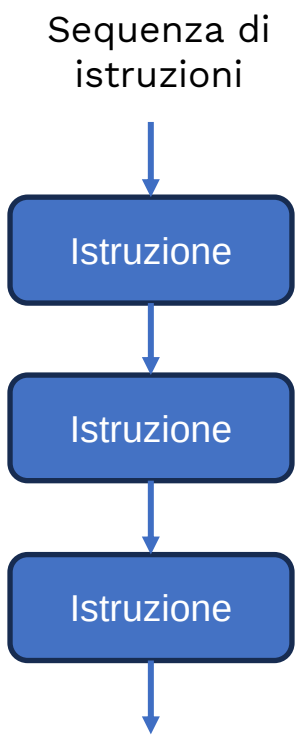
DIPARTIMENTO
DI INGEGNERIA

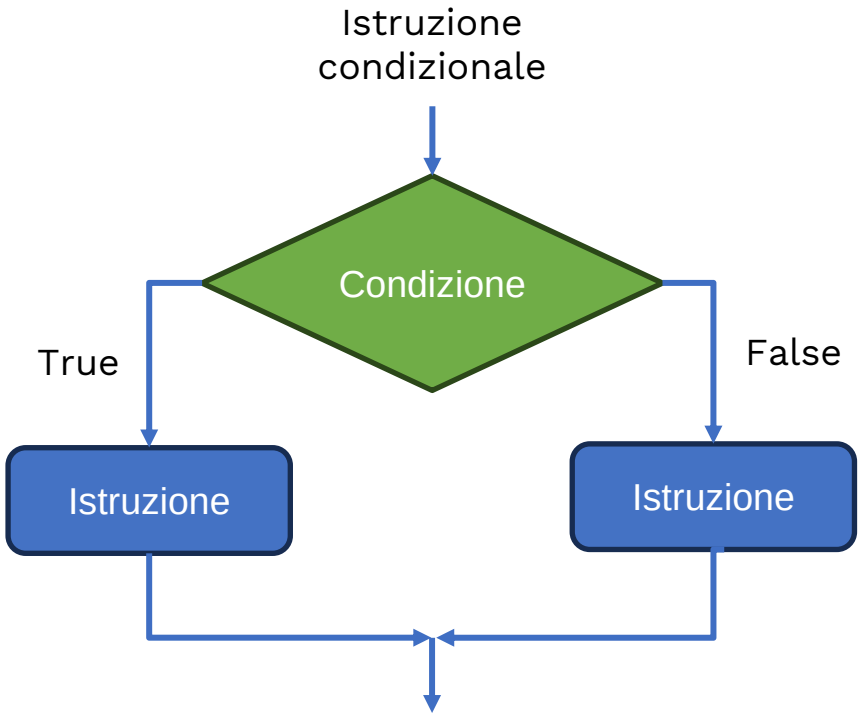
Istruzioni complesse

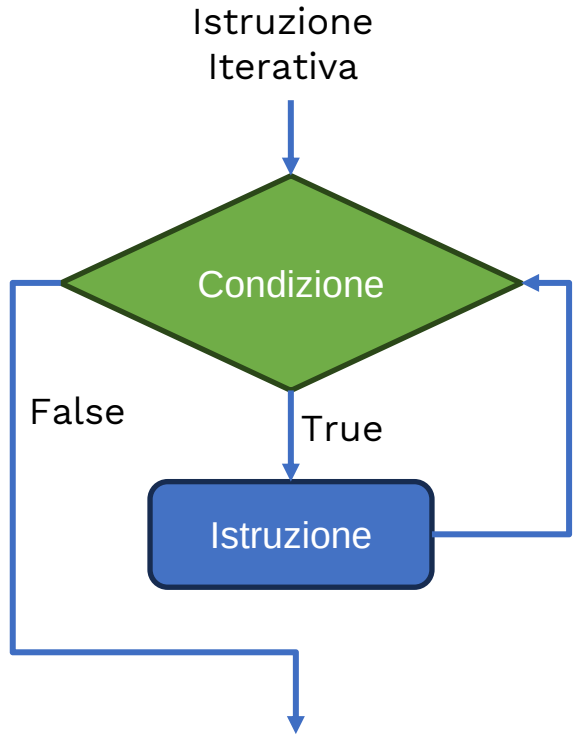
www.unipg.it

Programmazione strutturata

Paradigma di programmazione finalizzato al miglioramento della leggibilità, qualità e tempo di sviluppo di programmi utilizzando estensivamente strutture e costrutti di controllo del flusso di esecuzione, come **if/else , while/for, strutture a blocchi e funzioni**.

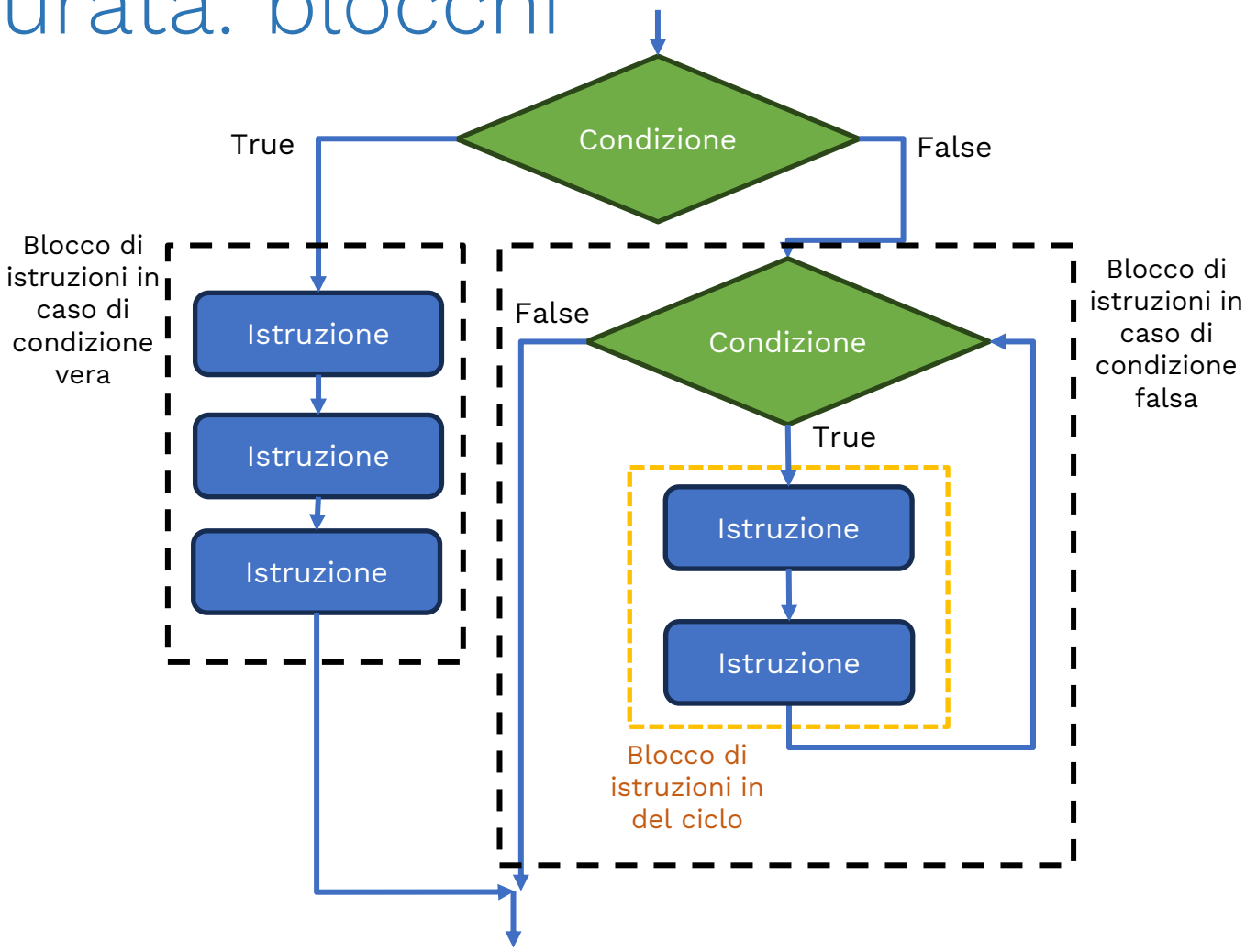






Programmazione strutturata: blocchi

Un **blocco** è un pezzo di codice Python che viene eseguito come se fosse un'istruzione **(statement)** unitaria.



Python: Indentazione

- L'indentazione è un aspetto importante e fondamentale della programmazione Python. L'indentazione in altri linguaggi, come C, C++, Java, ecc., è utilizzata principalmente per motivi di leggibilità e non è obbligatoria.
- L'indentazione in Python è invece obbligatoria e una pratica essenziale che deve essere seguita durante la scrittura del codice.
- In termini semplici, l'indentazione si riferisce agli spazi posti all'inizio delle righe di codice. Le istruzioni con lo stesso livello di indentazione, cioè le istruzioni con un numero uguale di spazi bianchi prima di loro, sono considerate come un **blocco.**
- Un **blocco** è un pezzo di codice Python che viene eseguito come se fosse un'istruzione **(statement)** unitaria.
- L'indentazione in Python svolge il ruolo di parentesi graffe negli altri linguaggi di programmazione

Indentazione

- La condizione del primo **if** è vera e perciò l'esecuzione continua all'interno del blocco **if**
- Dalla riga 3 alla riga 6: blocco del primo **if ->** 1 livello di indentazione (4 spazi di indentazione)
- Anche la condizione del secondo **if** (all'interno del blocco del primo **if**) è vera, entriamo perciò nel relativo blocco
- La riga 4 è il blocco del secondo **if->** secondo livello di indentazione (8 spazi di indentazione).
- Chiaramente la riga 4 è anche nel blocco del primo **if**

```
1  x = 10
2  if x != 0:
3      if x%2 == 0:
4          print("The number is even")
5      else:
6          print("The number is odd")
7  else:
8      print("The number is zero, i.e., it is neither even nor odd")
```

Indentazione

- L'indentazione non può essere utilizzata nella prima riga del codice Python.
- Il livello di indentazione standard in Python è di quattro spazi. Ciò indica che ogni livello di indentazione deve essere di quattro spazi.
- È opportuno non mischiare tabulazioni e spazi quando si indenta il codice in Python. È preferibile utilizzare solo spazi in quanto ciò garantisce la coerenza tra diversi editor di testo e IDE. Per evitare confusione, utilizzare sempre gli spazi su tutte le piattaforme.
- La coerenza è importante durante l'indentazione del codice in Python. Tutte le righe che rientrano in un blocco di codice devono essere rientrate allo stesso livello.
- L'indentazione è una parte importante del linguaggio di programmazione Python, tuttavia dovrebbe essere usata con moderazione e non essere abusata. Evita sempre l'indentazione aggiuntiva, in quanto renderà il codice più difficile da leggere e capire.

- **N.B.** per migliorare la leggibilità, in Python, è possibile dividere in più righe il contenuto di parentesi (), [] e {} senza incorrere in un errore di sintassi. L'interprete considererà tutto ciò che è tra le parentesi come se fosse in un'unica riga

Indentazione

- Errori di indentazione possono essere rilevati dall'interprete python come «Indentation error»
- Se, ad esempio, dopo un'istruzione condizionale non viene indentato il codice, l'interprete segnalerà un errore

Codice

```
1   x=10
2   if x!= 0:
3   print("numero 10")
```

Console output

```
    print("numero 10")
    ^
IndentationError: expected an indented block after 'if' statement on line 2

Process finished with exit code 1
```

Indentazione

- Tuttavia, un uso scorretto dell'indentazione può dar luogo ad errori semantici che non vengono segnalati dall'interprete ma che comportano un esecuzione non coerente con quanto previsto.

Codice

```
1    x = 0
2    if x != 0:
3        print("the current number is: ", x)
4    print("the current number is not zero")
```

Console output

```
the current number is not zero

Process finished with exit code 0
```

Istruzioni condizionali

- In Python, le istruzioni condizionali hanno lo stesso comportamento degli altri linguaggi
- Esistono tre tipologie di strutture condizionali: **if, if-else** e **if-elif-else**
    - **if:** il blocco di istruzioni è eseguito solo se la condizione è vera
    - **if-else:** Nel caso in cui la condizione è vera viene seguito il blocco di istruzioni relativo ad **if**, altrimenti viene eseguito il blocco dell'**else**
    - **if-elif-else:** Solo uno dei blocchi nella catena di **if-elif-else** viene eseguito. Tra le condizioni dell'**if** e dei vari **elif**, viene eseguito il blocco della prima che risulta vera. Se nessuna risulta vera viene eseguito il blocco dell'**else**

Istruzioni condizionali

Codice

```
1  input_value = int(input("Inserire un numero tra 0 e 3\n"))
2  if input_value == 0:
3      print("Valore zero")
4  elif input_value == 1:
5      print("Valore uno")
6  elif input_value == 2:
7      print("Valore due")
8  elif input_value == 3:
9      print("Valore tre")
10 else:
11     print("è stato inserito un numero non valido")
```

Console output

```
Inserire un numero tra 0 e 3
2
Valore due
```

Istruzioni condizionali

Attenzione!: Solo uno dei blocchi nella catena di **if-elif-else** viene eseguito. Tra le condizioni dell'**if** e dei vari **elif**, viene eseguito il blocco della prima che risulta vera. Se nessuna risulta vera viene eseguito il blocco dell'**else**

Codice

```
# Non è possibile con il seguente codice stampare
# delle stringhe che informano che il numero inserito
# è, ad esempio, sia positivo che pari
input_value = int(input("Inserire un numero\n"))
if input_value > 0:
    print("Numero positivo")
elif input_value%2==0:
    print("Numero pari")
elif input_value < 0:
    print("Numero negativo")
else:
    print("Numero dispari")
```

Console output

```
Inserire un numero
2
Numero positivo
```

Istruzioni condizionali

Attenzione!: Solo uno dei blocchi nella catena di **if-elif-else** viene eseguito. Tra le condizioni dell'**if** e dei vari **elif**, viene eseguito il blocco della prima che risulta vera. Se nessuna risulta vera viene eseguito il blocco dell'**else**

Codice

```
# Un possibile codice corretto per stampare sia
# se un numero è pari/dispari che se un numero è
# positivo/negativo è il seguente
input_value = int(input("Inserire un numero\n"))
bool_positivo = input_value > 0
bool_pari = input_value%2==0
if bool_positivo and bool_pari:
    print("Numero positivo e pari")
elif bool_positivo and not bool_pari:
    print("Numero positivo e dispari")
elif not bool_positivo and bool_pari:
    print("Numero negativo e pari")
else:
    print("Numero negativo e dispari")
```

Console output

```
Inserire un numero
-4
Numero negativo e pari
```

Istruzioni Iterative: ciclo for

- Come negli altri linguaggi di programmazione, in Python il ciclo **for** è un iteratore generico che può scorrere una sequenza ordinata di oggetti
- Può essere applicato ad una qualsiasi istanza di una classe iterabile
  - liste: il **for** itera sugli elementi della lista
  - stringhe: il **for** itera sui caratteri della stringa
  - tuple: il **for** itera sugli elementi della tupla
  - dizionari: il **for** itera sulle chiavi del dizionario
- La sintassi generica è la seguente:

for var **in** sequenza:
    Istruzione1
    Istruzione2
    …

- *var* è la variabile del ciclo e viene assegnata iterativamente ad ognuno degli elementi della sequenze
- *sequenza* è la sequenza del ciclo
- Le istruzioni all'interno del ciclo **for** sono il corpo del ciclo (blocco)

Istruzioni Iterative: ciclo for

Codice

```
#for su un oggetto lista
for i in [4, 5, 6, 7, 8]:
    print(i, end=', ')
print("")
#for su un oggetto tupla
for i in (2, 'ciao', 3.0, "Hello", 1):
    print(i, end=', ')
print("")
#for su un oggetto stringa
for i in "Un programma python":
    print(i, end=', ')
print("")
```

Console output

```
4, 5, 6, 7, 8,
2, ciao, 3.0, Hello, 1,
U, n,  , p, r, o, g, r, a, m, m, a,  , p, y, t, h, o, n,
```

Istruzioni Iterative: ciclo for- la funzione range()

- La funzione *range(start, stop, increment)* restituisce un oggetto iterabile costituito da interi nell'intervallo specificato.
- La funzione prende 3 parametri:
    - stop: definisce il numero massimo dell'intervallo di interi da inserire nella lista
    - start: definisce il numero di partenza della sequenza di interi (opzionale, di default è pari a 0)
    - increment: definisce l'incremento tra un intero e l'altro nella sequenza (opzionale, di default è pari a 1)

Istruzioni Iterative: ciclo for

Codice

```
1  for j in range(2, 10, 2):
2      print(j)
3  print("")
4
5  for j in range(10):
6      print(j)
```

Console output

```
2
4
6
8

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

Istruzioni Iterative: ciclo for- cicli annidati

- Come negli altri linguaggi di programmazione, è possibile creare cicli for annidati.
- I blocchi di codice in questo caso sono 2: il primo associato al primo ciclo **for** (1° livello di indentazione) e il secondo associato al secondo ciclo **for** che è nel blocco del primo ciclo (2° livello di indentazione, dove è presente la funzione print()).

Codice

```
1  for i in range(4):
2      for j in range(3):
3          print("i: ", i, " j: ", j)
```

Console output

```
i:  0  j:  0
i:  0  j:  1
i:  0  j:  2
i:  1  j:  0
i:  1  j:  1
i:  1  j:  2
i:  2  j:  0
i:  2  j:  1
i:  2  j:  2
i:  3  j:  0
i:  3  j:  1
i:  3  j:  2
```

Istruzioni Iterative: ciclo while

- Il ciclo **while** consente di ripetere istruzioni fino a che una certa condizione resta vera e si interrompe appena la condizione non è più vera
- La sintassi in Python è la seguente:

```
while condizione:
    istruzione1
    istruzione2
```

Codice

```
1  i = 4
2  while i !=0:
3      print(i)
4      i = i-1
```

Console output

```
4
3
2
1
```

Istruzioni Iterative: break e continue

- L'istruzione **break** interrompe l'esecuzione di un ciclo (sia **for** che **while**)
- L'struzione **continue** interrompe l'iterazione corrente del ciclo (sia **for** che **while**) e inizia la successiva

Codice

```
1   sum = 0
2   for i in range(2, 20, 2):
3       sum += i
4       if sum > 10:
5           break
6   print("Somma finale: ", sum)
7
8   product = 1
9   num_list = [1, 4, 6, 9.0, 0.0, 1]
10  num_element = len(num_list)
11  i = 0
12  while i < num_element:
13      current_element = num_list[i]
14      i += 1
15      if current_element==0.0:
16          continue
17      product *= current_element
18
19  print("Prodotto finale: ", product)
```

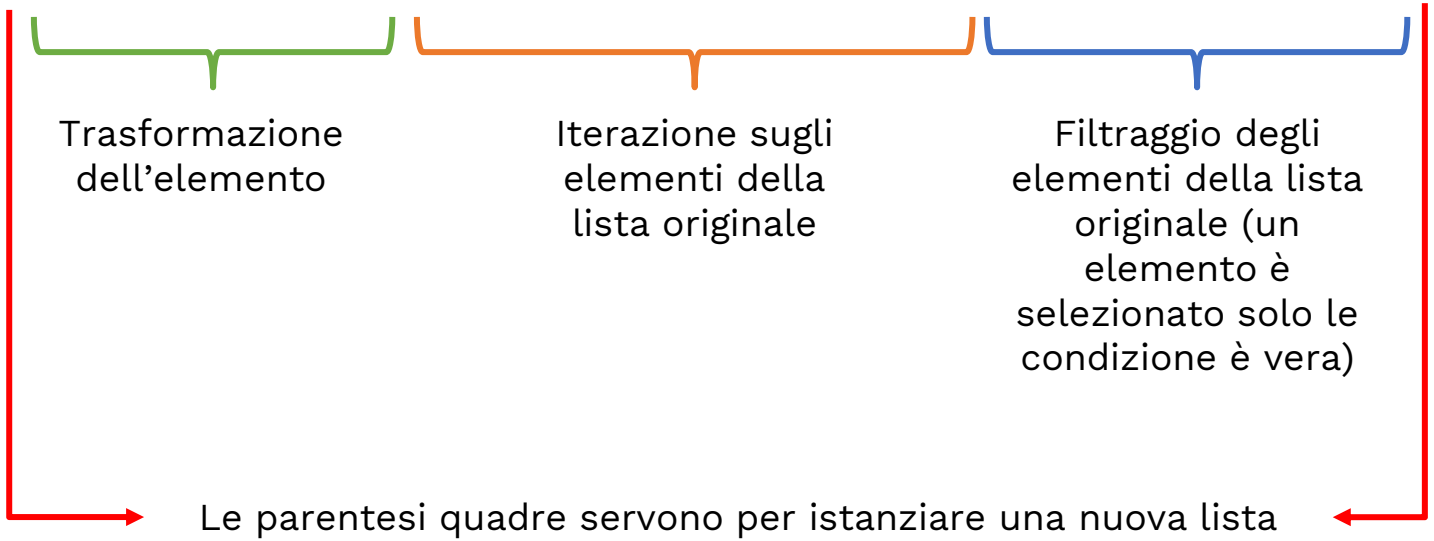
Console output

```
Somma finale:  12
Prodotto finale:  216.0
```

Istruzioni Iterative: list comprehension

- La **list comprehension** (comprensione di lista) è un istruzione basata sul ciclo **for** molto utile e diffusa in Python
- Questa istruzione offre un metodo compatto e facilmente leggibile per creare una nuova lista filtrando e trasformando gli elementi di un'altra
- La sintassi generale è la seguente:

transformed_and filtered_list = [function(element) **for** element **in** original_list **if** condition(element)]



Istruzioni Iterative: list comprehension

- In un'unica istruzione iteriamo attraverso gli elementi della lista originale, li selezioniamo solo se una certa condizione è verificata (**filtraggio**) e li trasformiamo prima di inserirli nella nuova lista (**trasformazione**)
- È chiaramente possibile anche usare l'operatore solo come filtro o come trasformatore di elementi:

```
transformed_list = [function(element) for element in original]

filtered_list = [ element for element in original if condition(element)]
```

Istruzioni Iterative: list comprehension

Codice

```
1   num_list = [1, 2, 5, 7, 9]
2
3   filtered_list = [n for n in num_list if n>2]
4   print(filtered_list)
5
6   transformed_list = [n**2 for n in num_list]
7   print(transformed_list)
8
9   filtered_and_transformed_list = [
10      n**2 for n in num_list if n > 1
11  ]
12  print(filtered_and_transformed_list)
```

Console output

```
[5, 7, 9]
[1, 4, 25, 49, 81]
[4, 25, 49, 81]
```

Istruzioni complesse: esempi

- Scrivere un codice Python che restituisce la somma dei numeri contenuti in una lista (utilizzando il ciclo **for**)

Codice

```
1  number_list = [3, 6, 7.0, -4, 100, 25, 3.0]
2  somma = 0
3  for n in number_list:
4      somma += n
5  print("Somma: ", somma)
```

Console output

```
Somma:  140.0
```

Istruzioni complesse: esempi

- Scrivere un codice Python che stampa True se in una lista data esistono due numeri il cui prodotto è uguale a un certo numero (es. 20)

Codice

```
number_list = [3, 6, 7.0, -4, 100, 25, 3.0]
target_value = 300
found = False
for i in range(len(number_list)):
    for j in range(i+1, len(number_list)):
        if (number_list[i]*number_list[j] == target_value):
            found = True
        if found:
            break
if found:
    print("Trovati")
else:
    print("Non ci sono numeri il cui prodotto è", target_value)
```

Console output

```
Il prodotto tra i numeri  3  e  100  è  300
Trovati
```

Istruzioni complesse: esempi

- Scrivere un codice Python che chiede all'utente il risultato di un operazione e ripete la domanda per 3 volte se l'input ricevuto è scorretto. Se dopo la terza volta non riceve il risultato corretto lo stampa

Codice

```
correct_answer = False
count = 0
while (not correct_answer and count < 3):
    user_answer = int(input("Quale è il risultato del prodotto tra 7 e 8?\n"))
    if user_answer == 56:
        correct_answer = True
        print("Risposta esatta!")
    else:
        print("Risposta errata, prova ancora")
        count +=1
if not correct_answer:
    print("Hai sbagliato per 3 volta, la risposta corretta è 56")
```

Console output

```
Quale è il risultato del prodotto tra 7 e 8?
56
Risposta esatta!
```

Istruzioni complesse: esempi

- Scrivere un codice Python che, data una lista di numeri, genera una nuova lista che contiene i valori che appaiono una sola volta nella lista iniziale e ne calcola il valore assoluto.

Codice

```
1  number_list = [3, 3, 5, 8.0, -3.5, 5.8, 100, 100, 400, 8.0, 5]
2  new_list = [abs(n) for n in number_list if number_list.count(n)==1]
3  print(new_list)
```

Console output

```
[3.5, 5.8, 400]
```

A.D. 1308
unipg

DIPARTIMENTO
DI INGEGNERIA

Funzioni in Python

www.unipg.it

Funzioni

- Le funzioni in Python, come negli altri linguaggi, sono blocchi di codice a cui viene assegnato un nome arbitrario, ovvero un insieme di istruzioni che possono essere eseguite come se fossero un'unica istruzione unitaria richiamando la funzione in qualsiasi parte del programma
- In Python, la definizione di una funzione segue al seguente sintassi:

```
def f(arg1, arg2, ....):
        istruzione1
        istruzione2
        return variabili_restituite
```

- Una funzione in Python può essere chiamata come segue:

```
result = f(val1, val2, ....)
```

Funzioni

Codice

```
1  def squared_sum (a, b):
2      return (a+b)**2
3
4  x = 2
5  y = 4
6  print(squared_sum(x, y))
```

Console output

```
36
```

Funzioni

- Come negli altri linguaggi le funzioni possono avere dei valori di **return** oppure essere **void.** In Python, una funzione **void** è semplicemente una funzione che non termina con un'istruzione **return**
- Nello script, le funzioni possono essere chiamato solo dopo essere state definite, altrimenti Python solleverà un eccezione di tipo *«Name not defined»*

Codice

```
1  x = 2
2  y = 4
3  print(squared_sum(x, y))
4
5  def squared_sum (a, b):
6      return (a+b)**2
```

Console output

```
Traceback (most recent call last):
  File
    print(squared_sum(x, y))
NameError: name 'squared_sum' is not defined
```

Funzioni

- A differenza di altri linguaggi, **def** è un istruzione che crea un **oggetto** della classe **function** e assegna l'oggetto ad una variabile il cui nome è quello della funzione appena definita.
- Questo significa che è possibile assegnare l'oggetto funzione ad un'altra variabile o anche passarlo come argomento ad un'altra funzione

Codice

```
1  def squared_sum (a, b):
2      return (a+b)**2
3
4  x = 2
5  y = 4
6  another_variable = squared_sum
7  print(another_variable(x, y))
```

Console output

```
36
```

Funzioni

- In Python, le funzioni possono restituire un numero arbitrario di valori.
- L'interprete ritorna tutti i risultati inserendoli all'interno di un oggetto della classe tupla (la creazione dell'oggetto è gestita automaticamente dall'interprete)

Codice

```
1   def sum_diff_and_squared_sum (a, b):
2       sum_value = a+b
3       diff_value = a-b
4       squared_sum_value = (a+b)**2
5       # La seguente istruzione è equivalente a
6       # return (sum_value, diff_value, squared_sum_value)
7       return sum_value, diff_value, squared_sum_value
8
9   x = 2
10  y = 4
11  result = sum_diff_and_squared_sum(x, y)
12  print(result)
13  print(type(result))
```

Console output

```
(6, -2, 36)
<class 'tuple'>
```

Funzioni

- Nel caso di funzioni multioutput, è possibile assegnare il risultato su una variabile e poi accedere ai suoi elementi con l'operatore di indicizzazione delle tuple, oppure usare l'**automatic unpacking** di Python

Codice

```
1   def sum_diff_and_squared_sum (a, b):
2       sum_value = a+b
3       diff_value = a-b
4       squared_sum_value = (a+b)**2
5       # La seguente istruzione è equivalente a
6       # return (sum_value, diff_value, squared_sum_value)
7       return sum_value, diff_value, squared_sum_value
8
9   x = 2
10  y = 4
11  sum_res, diff_res, squared_res = sum_diff_and_squared_sum(x, y)
12  print(sum_res)
13  print(diff_res)
14  print(squared_res)
```

Console output

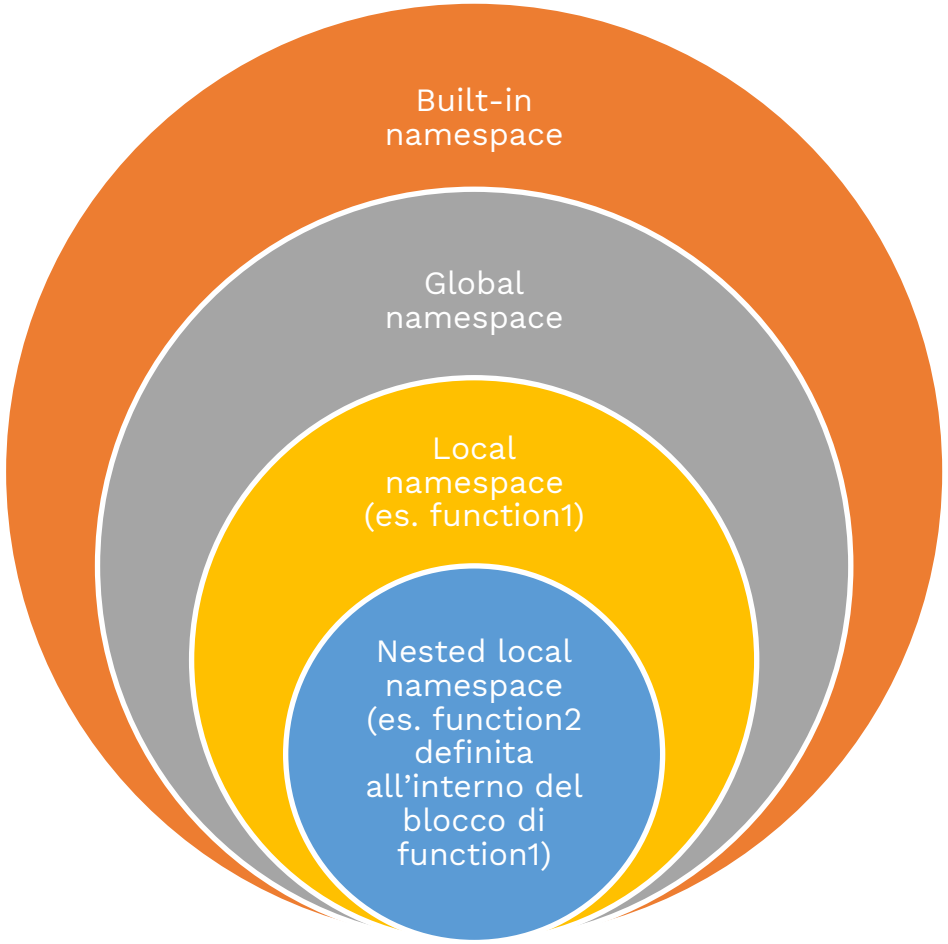
```
6
-2
36
```

Funzioni: namespace e scope

- Il **namespace** o **contesto** è, in generale, il sistema di nomenclatura delle variabili necessario a garantire l'unicità dei nomi ed evitare ambiguità
- Lo **scope** di un nome di una variabile è la regione del programma dalla quale quel nome può essere accessibile.
- In Python il nome di una variabile viene associato al namespace relativo all'area del programma in cui viene creata la variabile, definendo quindi anche il suo scope.
- In Python esistono tre namespace fondamentali:
  - **built-in namespace:** il namespace delle variabili built-in di Python dove, ad esempio, sono definite le funzioni *type(), id(), abs(), sum()*. Questo namespace è il più «esterno» e i nomi delle sue variabili hanno lo scope più ampio nel programma Python (sono accessibili da ogni parte del codice)
  - **global namespace:** il namespace principale del programma, associato al codice **main**, fuori dalle funzioni (senza livelli di indentazione). Lo scope delle variabili definite nel global namespace include la regione del global namespace e di tutti i local namespace generati dalle varie funzioni definite
  - **local namespace:** il namespace all'interno dei blocchi delle funzioni. I nomi delle variabili definite nei local namespace non sono visibili dai namespace più esterni (built-in, global e namespace locali più esterni). Se una funzione viene definita all'interno di un'altra funzione, si parla di **nested local namespace** (annidato)

Funzioni: namespace e scope

- Lo scope delle variabili del namespace built-in è costituito da tutti i programmi Python (sono sempre visibili)
- Lo scope delle variabili del namespace global è costituito dalla regione del codice del global namespace stesso e da tutti le regioni del local namespace delle varie funzioni/classi
- Lo scope delle variabili dei local namespaces è costituita dal local namespace stesso in cui è definita e da tutti i local namspace «figli» (nested) associati a funzioni/classi definite all'interno del namespace locale più esterno
- Lo scope di una variabile può anche essere interpretato considerando il ciclo di vita di un namespace. Un namespace è attivo fintanto che il blocco di codice associato è in esecuzione e termina con esso. Ad esempio, il local namespace di una funzione nasce alla prima istruzione della funzione e muore dopo il **return**. Per questo motivo le variabili del namespace della funzione non sono visibili esternamente (il namespace non esisto più dopo la chiamata della funzione)



Funzioni: namespace e scope

Codice

```
var1 = 5
print(var1)
def some_func():
    print("Var1 è accessibile a anche nel local namespace di some_func: ", var1)
    # var2 è nel local namespace associato a some_func
    var2 = 6

    def some_inner_func():
        print("Var1 è accessibile anche nel local namespace di some_inner_func: ", var1)
        print("Var2 è accessibile anche nel local namespace di some_inner_func: ", var2)
        # var3 è nel local namespace annidato associato a some_inner_func
        # namespace
        var3 = 7

    print("Var3 non è accessibile nel local namespace di some_func: ", var3)

print("Var2 non è accessibile nel local namespace di some_func: ", var2)
print("Var3 non è accessibile nel local namespace di some_func: ", var3)
```

Funzioni: namespace e scope

- Se una variabile ha lo stesso nome in due namespace diversi (global e local ad esempio), vengono trattate come due copie distinte

Codice

```
1   sum_res = 10
2
3   def sum (a, b):
4       # La variabile const è visibile nel local namespace perchè è
5       # definita nel global namespace che è più esterno
6       sum_res = a+b
7       print("sum_res all'interno della funzione: ", sum_res)
8       return sum_res
9
10  sum(4, 5)
11  #La variabile sum non è visibile nel global namespace
12  print("sum_res nel global namespace: ", sum_res)
```

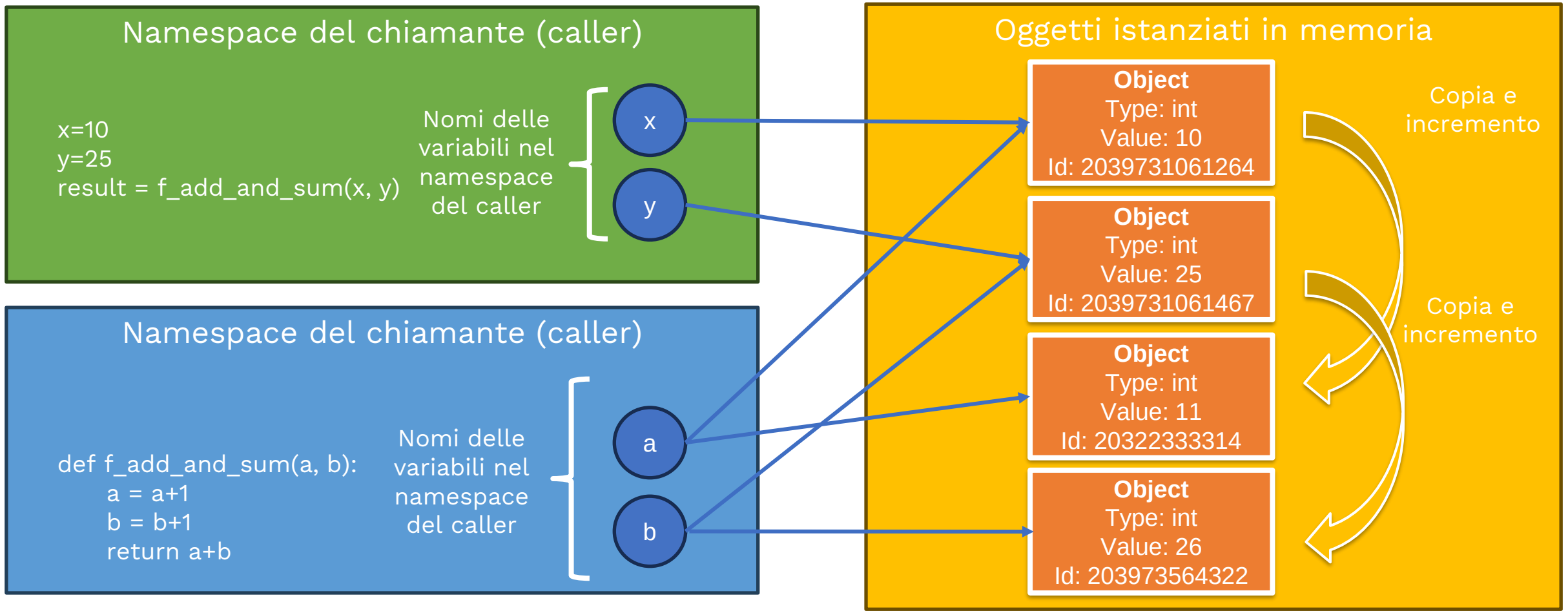
Console output

```
sum_res all'interno della funzione:  9
sum_res nel global namespace:  10
```

Funzioni: passaggio degli argomenti

- Quando viene chiamata una funzione, gli oggetti che vengono passati come argomento, tramite delle variabili del namespace del **caller** (ovvero il main del programma **(global namespace)** o un'altra funzione (namespace locale della funzione più «esterna»)), vengono assegnati a delle variabili locali del namespace della funzione chiamata.
- Il passaggio degli argomenti durante la chiamata di funzioni in Python è **sempre** per **riferimento (reference)**. Tuttavia, è importante sottolineare una differenza tra oggetti **immutabili** e oggetti **mutabili:**
    - Abbiamo visto che è possibile avere due variabili che puntano allo stesso oggetto immutabile. È quindi possibile, nel caso di un'istanza di una classe immutabile, assegnare la variabile locale della funzione chiamata allo stesso oggetto referenziato dalla variabile passata come argomento. Tuttavia, poiché di un oggetto immutabile, per definizione, non si possono cambiare le caratteristiche, qualsiasi manipolazione di esso tramite la sua variabile locale comporta la creazione di una nuova istanza. Questa, avendo uno scope locale alla funzione in cui è definita, non avrà impatto sul valore dell'oggetto referenziato dalla variabile globale che era stata passata alla funzione. Per questo motivo, in pratica, spesso in maniera informale, si parla di passaggio per **valore** nel caso di oggetti immutabili, in quanto la loro manipolazione necessariamente genera una **copia** dell'oggetto originale.
    - Nel caso di oggetti mutabili, invece, la variabile locale della funzione chiamata punta sempre all'oggetto referenziato dalla variabile passata alla funzione e le modifiche effettuate all'oggetto saranno mantenute anche dopo il termine della funzione e visibili nel codice del chiamante

Funzioni: passaggio degli argomenti – oggetti immutabili



Funzioni: passaggio degli argomenti – oggetti immutabili

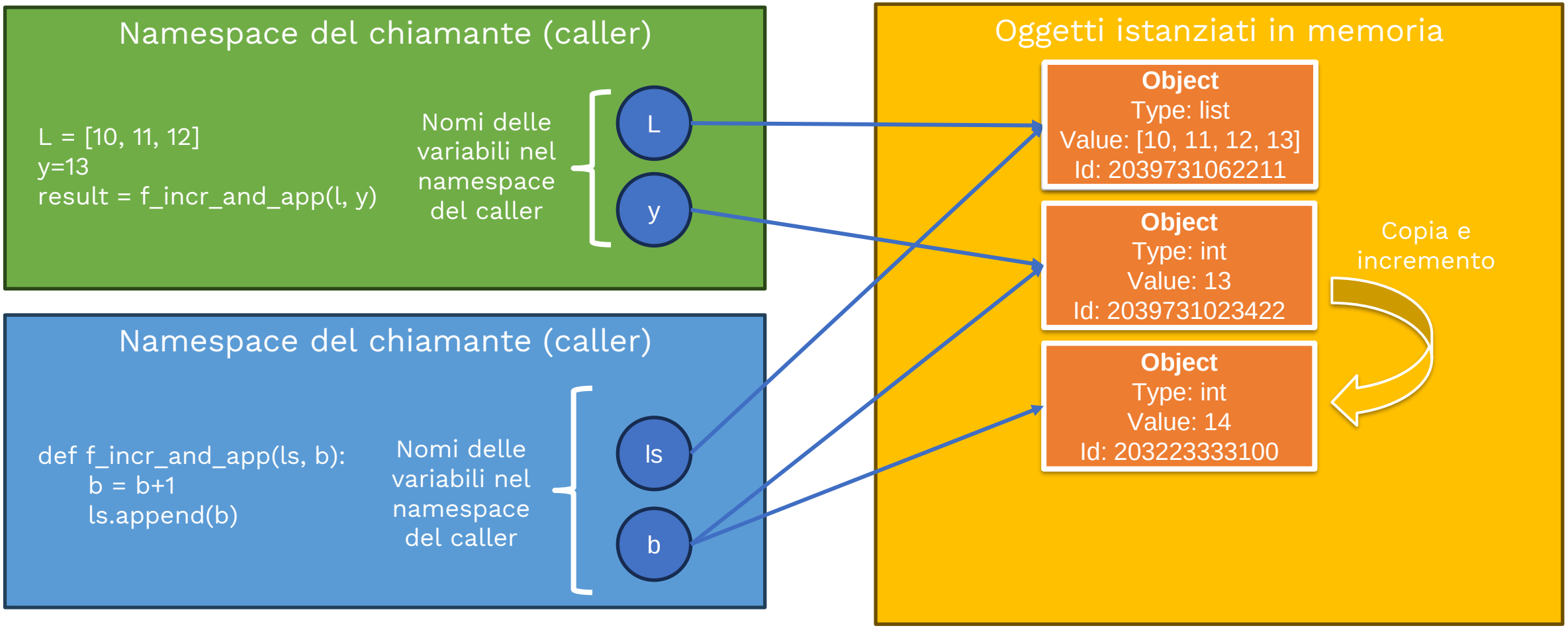
Codice

```
1   def f_increment_and_sum(a, b):
2       a = a+1
3       b = b+1
4       print(a)
5       print(b)
6       return a+b
7
8   x = 10
9   y = 25
10  f_increment_and_sum(x, y)
11  print(x)
12  print(y)
```

Console output

```
11
26
10
25
```

Funzioni: passaggio degli argomenti – oggetti mutabili



Funzioni: passaggio degli argomenti – oggetti immutabili

Codice

```
1   def f_incr_and_app(ls, b):
2       print("id di b nel local namespace: ", id(b))
3       b = b+1
4       print("id di b nel local namespace dopo l'incremento: ", id(b))
5       ls.append(b)
6       print("id di ls nel local namespace:", id(ls))
7
8   x = 13
9   L = [10, 11, 12]
10  print("id di x nel global namespace: ", id(x))
11  print("id di L nel global namespace: ", id(L))
12  print("Valore di L: ", L)
13  f_incr_and_app(L, x)
14  print("valore di x dopo la funzione: ", x)
15  print("id di L nel global namespace dopo la funzione: ", id(L))
16  print("Valore di L dopo la funzione: ", L)
```

Console output

```
id di x nel global namespace:  2082466955888
id di L nel global namespace:  2082472180096
id di b nel local namespace:  2082466955888
id di b nel local namespace dopo l'incremento:  2082466955920
id di ls nel local namespace: 2082472180096
valore di x dopo la funzione:  13
Valore di L:  [10, 11, 12, 14]
id di L nel global namespace dopo la funzione:  2082472180096
Valore di L dopo la funzione:  [10, 11, 12, 14]
```

Funzioni: passaggio degli argomenti – oggetti immutabili

- Se si vuole evitare che una funzioni alteri i valori di un oggetto mutabile referenziato da una variabile globale (o di un namespace più esterno a quello della funzione) è necessario creare esplicitamente una copia dell'oggetto mutabile. Ad esempio, per le liste, è sufficiente utilizzare l'operatore di slicing per creare una copia dell'oggetto: **L[:].** In questo modo la variabile locale verrà assegnata ad un clone della lista referenziata dalla variabile esterna alla funzione, e modifiche alla lista tramite la variabile locale non avranno impatto sulla lista originale.

Codice

```
1  def f_app(ls):
2      ls.append(13)
3      print("valore di ls nella funzione: ", ls)
4      print("id di ls nel local namespace:", id(ls))
5
6  L = [10, 11, 12]
7  print("id di L nel global namespace: ", id(L))
8  print("Valore di L: ", L)
9  f_app(L[:])
10 print("id di L nel global namespace dopo la funzione: ", id(L))
11 print("Valore di L dopo la funzione: ", L)
```

Console output

```
id di L nel global namespace:  2480347330944
Valore di L:  [10, 11, 12]
valore di ls nella funzione:  [10, 11, 12, 13]
id di ls nel local namespace: 2480352728512
id di L nel global namespace dopo la funzione:  2480347330944
Valore di L dopo la funzione:  [10, 11, 12]
```

Funzioni: passaggio di parametri per nome

- In Python è possibile esplicitare il nome del parametro della funzione durante la sua chiamata.
- Questa pratica è molto conveniente in quando aumenta significativamente la leggibilità del codice e, inoltre, consente di elencare gli argomenti della funzione in ordine arbitrario, senza necessità di rispettare quello determinato in fase di definizione della funzione. Sarà infatti l'interprete che, grazie all'esplicitazione dei nomi dei parametri della funzione, effettuerà l'associazione.

Funzioni: passaggio di parametri per nome

Codice

```python
def custom_function (first, second):
    first = first**2
    return first+second

x = 2
y = 4

# Non usando il passaggio dei parametri per nome,
# passare le variabili in ordine diverso alla funzione
# può generare comportamenti diversi o errori
print("Passaggio senza nome, ordine x, y: ", custom_function(x, y))
print("Passaggio senza nome, ordine y, x: ",custom_function(y, x))

# Utilizzando invece il passaggio per nome,
# l'ordine dei parametri può essere cambiato
# arbitrariamente. Sarà l'inteprete che, sfruttando
# l'esplicitazione dei nomi in fase di chiamara,
# riuscirà ad associare i parametri alle variabili locali
print("Passaggio con nome, ordine x, y: ", custom_function(first=x, second=y))
print("Passaggio con nome, ordine x, y: ", custom_function(second=y, first=x))
```

Console output

```
Passaggio senza nome, ordine x, y:  8
Passaggio senza nome, ordine y, x:  18
Passaggio con nome, ordine x, y:  8
Passaggio con nome, ordine x, y:  8
```

Funzioni: valori di default

- È possibile in Python specificare dei valori di default per i parametri di una funzione in fase di definizione di quest'ultima.
- Durante la chiamata della funzione, se un parametro con un valore di default non viene specificato, la funzione assegnerà alla variabile locale un oggetto con il valore di default
- Chiaramente, se un parametro non ha un valore di default e, in fase di chiamata, non viene specificato, l'interprete solleverà un eccezione runtime.

Funzioni: valori di default

Codice

```
1   def custom_function (first, second=10):
2       first = first**2
3       return first+second
4
5   x = 2
6   y = 4
7
8   print("Chiamata usando il valore di default: ", custom_function(first=x))
9   print("Chiamata usando un valore custom: ",custom_function(first=x, second=y))
10  print("Chiamata senza specificare first: ",custom_function(second=y))
```

Console output

```
Traceback (most recent call last):
  File
    print("Chiamata senza specificare first: ",custom_function(second=y))
TypeError: custom_function() missing 1 required positional argument: 'first'
Chiamata usando il valore di default:  14
Chiamata usando un valore custom:  8
```

A.D. 1308
unipg

DIPARTIMENTO
DI INGEGNERIA

Classi in Python

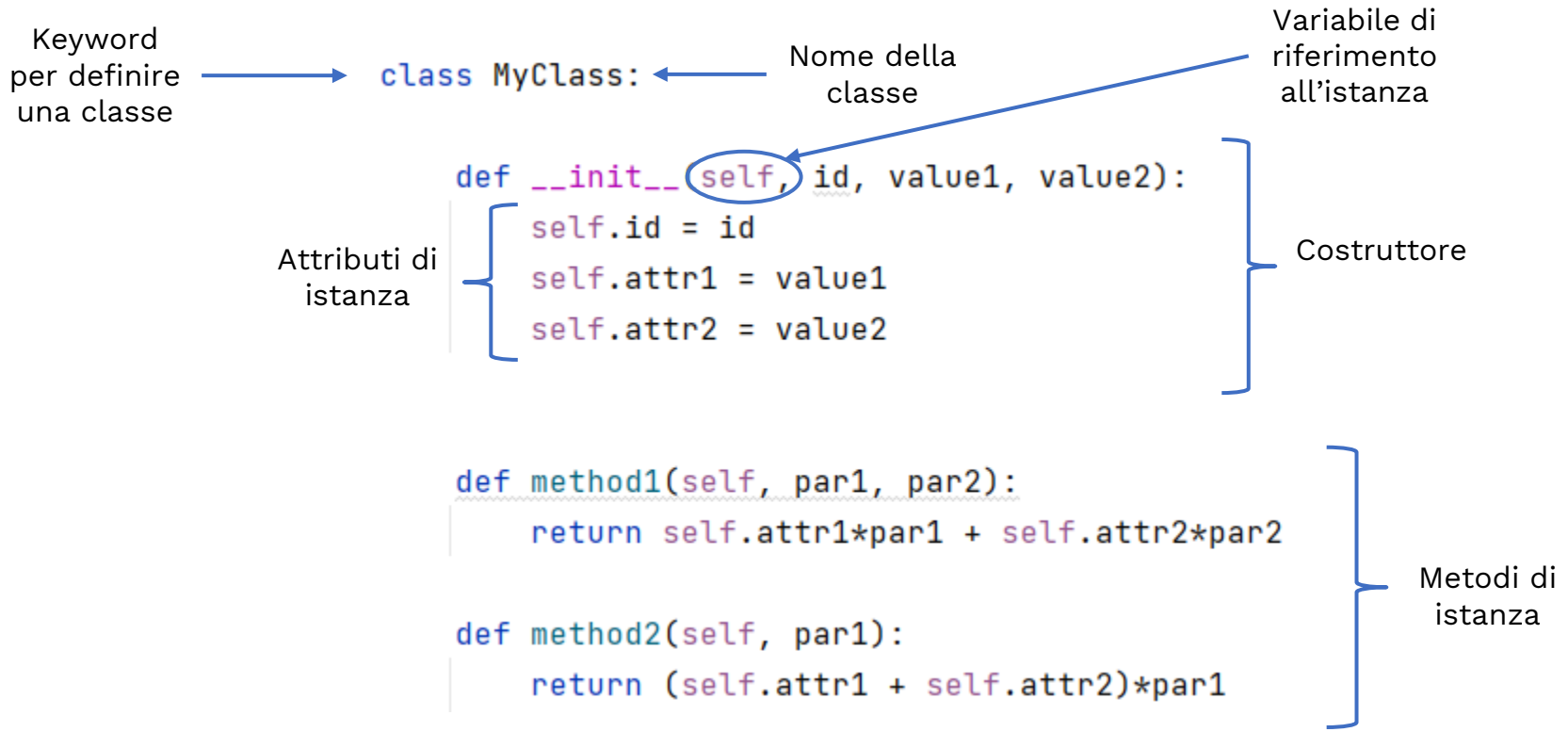
www.unipg.it

Classi

- Python è un linguaggio OOP (abbiamo visto come qualsiasi tipologia di dato in Python sia rappresentato da un'istanza di una certa classe)
- Fino ad ora abbiamo utilizzato classi **built-in**, come int, float, string, bool, list.... Queste classi built-in possono essere viste come i **tipi primitivi** degli altri linguaggi di programmazione
- Come possiamo creare delle nuove classi per le esigenze dello specifico programma che stiamo realizzando?
- La struttura delle classi Python, da un punto di vista generale, è molto simile a quella degli altri linguaggi di programmazione:
    - Utilizza costruttori per istanziare oggetti
    - Una classe può avere attributi
    - Una classe può avere metodi (di istanza, di classe o statici)

Classi: definizione

- Una classe in Python è definita in un blocco di codice (come le funzioni) e utilizza la seguente sintassi:



Classi: definizione

- Il costruttore *__init__* viene chiamato durante la creazione dell'oggetto e nel suo corpo è possibile inizializzare gli attributi di istanza (anche utilizzando gli argomenti passati al costruttore
- Se il costruttore non viene specificato, Python utilizza quello di default (che non prende parametri e non definisce alcun attributo)
- La keyword **self** svolge un ruolo analogo al **this** dei linguaggi C++ e Java, ovvero è la variabile che referenzia l'istanza dell'oggetto.
- Tutti i metodi di istanza prendono come parametro **self** (riferimento all'istanza) per accedere agli attributi di istanza e agli altri metodi

Classi: Instanziare un oggetto

Codice

```
class MyClass:

    def __init__(self, value1, value2):
        # attributo pubblico
        self.attr1 = value1
        #attributo protetto
        self._attr2 = value2
        #attributo privato
        self.__attr3 = 20

new_instance = MyClass(value1=1, value2=3)
# L'accesso ad un attributo pubblico è possibile attraverso la
# variabile che referenzia l'istanza
print(new_instance.attr1)

# Anche se non è una buona pratica,è possibile anche accedere ad un
# attributo privato sempre attraverso la variabile che referenzia l'istanza
print(new_instance._attr2)

# Non è invece possibile accedere all'attributo privato
# con al variabile che referenzia l'istanza
print(new_instance.__attr3)
```

Console output

```
25
```

Classi: attributi pubblici, protetti e privati

- In Python, l'accessibilità di un attributo di istanza non è gestito attraverso delle keyword esplicite (come in Java o C++), ma attraverso una convenzione sui nomi degli attributi.
- In particolare, viene usato il carattere underscore «_» all'inizio del nome dell'attributo per indicarne l'accessibilità
    - Se il nome di un attributo non è preceduto da underscore è **pubblico**
    - Se il nome di un attributo è preceduto da un underscore è **protetto**
    - Se il nome di un attributo è preceduto da due underscore è **privato**
- Se un attributo è pubblico è possibile accedervi direttamente attraverso la variabile che referenzia l'oggetto (attraverso l'operatore «.»)
- In realtà, è possibile accedere anche ad un attributo protetto sempre attraverso la variabile che referenzia l'oggetto, ma questa è considerata una cattiva pratica
- Invece, nel casi di un tentativo di accesso diretto ad un attributo privato con la variabile che referenzia l'oggetto, l'inteprete Python solleverà un'eccezione

Classi: attributi pubblici, protetti e privati

- Come è possibile accedere e modificare correttamente gli attributi protetti e privati di istanza?
- Utilizzando i metodi **getter** e **setter**
- In realtà è buona pratica usare questi metodi anche per accedere agli attributi pubblici per migliorare la leggibilità e ridurre il rischio di bug

Classi: Instanziare un oggetto

Codice

```
class MyClass:

    def __init__(self, value1, value2):
        self.attr1 = value1
        self._attr2 = value2
        self.__attr3 = 20
    def get_attr1(self):
        return self.attr1
    def set_attr1(self, value):
        self.attr1 = value
    def get_attr2(self):
        return self._attr2
    def set_attr2(self, value):
        self._attr2 = value
    def get_attr3(self):
        return self.__attr3
    def set_attr3(self, value):
        self.__attr3 = value

new_instance = MyClass(value1=1, value2=3)
new_instance.set_attr3(value=100)
print(new_instance.get_attr3())
```

Console output

```
100
```

Classi: metodi di classe e metodi statici

- I metodi di classe sono metodi che non hanno accesso all'istanza specifica della classe, ma ad attributi della classe in generale. A differenza dei metodi di istanza non prendono come parametro la variabile **self**, ma la variabile **cls,** ovvero un riferimento alla classe
- I metodi di classe sono tipicamente utilizzati per implementare un design pattern noto come **Factory Method,** (non sarà oggetto di questo corso)
- I metodi statici, invece, non prendono né un riferimento alla classe ne all'istanza e sono tipicamente usati per implementare funzionalità di utilità che sono concettualmente legate alla classe ma non dipendono dall'istanza specifica

- In Python, è possibile definire metodi di classe e di istanza usando i **decorator**
- I decorator sono uno strumento che consente di estendere o modificare il comportamento di una funzione senza riscriverne il codice
- I decorator vengono anteposti alla definizione di una funzione e sono caratterizzati dal carattere @ che precede il loro nome
- Esistono molti tipi di decorator in Python: vedremo i decorator *@classmethod* e *@staticmethod*

Classi: metodi di classe e metodi statici

Codice

```python
class Pizza:

    def __init__(self, ingredient1, ingredient2):
        self.attr1 = ingredient1
        self.attr2 = ingredient2

    #Factory method per creare istanze con parametri specificati del costruttore
    @classmethod
    def margherita(cls):
        return cls(ingredient1="mozzarella", ingredient2="pomodoro")

    @classmethod
    def prosciutto(cls):
        return cls(ingredient1="prosciutto", ingredient2="pomodoro")

    @staticmethod
    def pizza_size(raggio):
        return 3.14*(raggio**2)

margherita_object = Pizza.margherita()
prosciutto_object = Pizza.prosciutto()
print(Pizza.pizza_size(raggio=0.1))
```

Console output

```
0.031400000000000004
```

Classi: ereditarietà

- In Python, per implementare l'ereditarietà è necessario seguire la seguente procedura:
  - Specificare la classe *parent* da cui si eredita tra parentesi dopo il nome della classe *child* che stiamo definendo
  - Nel costruttore della classe *child* è necessario chiamare il costruttore della classe *parent* attraverso la funzione **super()**
  - Nel caso in cui la classe base (*parent)* sia astratta e abbia dei metodi non implementati, è necessario implementare quei metodi (altrimenti non sarà possibile istanziare oggetti della classe *child*)

Codice

```
1  class BaseClass:
2      def __init__(self, value1):
3          self.base_attr = value1
4
5
6  class ChildClass(BaseClass):
7      def __init__(self, par1, par2):
8          super().__init__(value1=par2)
9          self.child_attr= par2
10
11
12 child_instance = ChildClass(par1=10, par2=20)
```