

# Programmazione avanzata a.a. 2024-25

## Introduzione a Python (I lezione)

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

1

1

## Informazioni utili

- Il sito Web del corso:  
– [http://intranet.di.unisa.it/~debonis/progAv24\\_25](http://intranet.di.unisa.it/~debonis/progAv24_25)
- Il mio studio: numero 44, quarto piano, stecca 7
- L'orario di ricevimento: su appuntamento anche se sarà indicato un orario preferenziale sul sito web di unisa.

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

2

2

# Python

- Python è il linguaggio utilizzato nel corso per approfondire la conoscenza e applicare i concetti di programmazione avanzata.
- Perché Python?
  - Potente ed espressivo
  - Facile da apprendere
  - Codice facile da leggere e scrivere
  - Interpretato (ottimo per scrivere script)
  - Fornisce strumenti quali strutture dati di alto livello, tipizzazione dinamica, il dynamic binding che lo rendono adatto a sviluppare rapidamente diverse applicazioni
  - Ricchissima libreria standard e migliaia di librerie di terze parti.

3

## Origini



- Linguaggio di programmazione sviluppato agli inizi degli anni 90 presso il Centrum Wiskunde & Informatica (CWI)
- Ideato da Guido van Rossum nel 1989
- Il nome "Python" deriva dalla passione di Guido van Rossum per la serie televisiva



4

# Indice PYPL

Creato analizzando quanto spesso tutorial sul linguaggio sono cercati su Google

Worldwide, Sept. 2024 :

Rank	Change	Language	Share	1-year trend
1		Python	29.66 %	+1.6 %
2		Java	15.64 %	-0.2 %
3		JavaScript	8.3 %	-1.0 %
4		C#	6.64 %	-0.1 %
5		C/C++	6.46 %	-0.2 %
6	↑	R	4.66 %	+0.2 %
7	↓	PHP	4.35 %	-0.5 %
8		TypeScript	2.96 %	-0.0 %
9		Swift	2.69 %	+0.0 %
10	↑	Rust	2.65 %	+0.6 %
11	↓	Objective-C	2.45 %	+0.2 %
12		Go	2.08 %	+0.2 %
13		Kotlin	1.95 %	+0.2 %

Programmazione Avanzata 2024-25  
A. De Bonis

5

## Design Pattern

- Forniscono schemi generali per la soluzione di problematiche ricorrenti che si incontrano durante lo sviluppo del software.
- Favoriscono il riutilizzo di tecniche di design di successo nello sviluppo di nuove soluzioni.
- Evitano al progettista di riscoprire ogni volta le stesse cose.
- Permettono di sviluppare un linguaggio comune che semplifica la comunicazione tra le persone coinvolte nello sviluppo del software.
- Approfondiremo molti dei design pattern introdotti nel 1994 dal celebre gruppo di quattro autori noto con il nome di **Gang of Four (GOF)**

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

6

6

## Introspection e Reflection

- **Introspection** è la capacità di esaminare un oggetto durante l'esecuzione del programma.
- Ad esempio Python fornisce gli strumenti per conoscere tutti gli attributi di un oggetto e il tipo di un oggetto.
  - Ricordiamo che queste caratteristiche sono dinamiche.
- **Reflection** è uno strumento più potente in quanto non solo permette di esaminare gli oggetti ma anche di modificarne la struttura e il comportamento durante l'esecuzione del programma.
  - Fondamentale per implementare alcuni design pattern

7

## Programmazione funzionale

- Nei linguaggi orientati agli oggetti tutto ruota intorno agli oggetti
  - Le funzioni sono sempre collegate alle classi e non è possibile invocarle indipendentemente dalle classi o dagli oggetti.
- La programmazione funzionale in Python
  - consente una separazione tra oggetti e funzioni
    - il risultato delle funzioni dipende solo dall' input (non dallo stato dell'oggetto)
  - fornisce le modalità con cui combinare le funzioni
  - permette di trattare le funzioni come un qualsiasi altro oggetto
    - le funzioni possono essere assegnate a variabili, passate come argomenti ad altre funzioni, usate come valori di ritorno di altre funzioni .

8

## Generatori

- I generatori sono funzioni
  - la cui esecuzione può essere sospesa e ripresa
  - restituiscono oggetti su cui si può iterare.
  - al contrario di altri oggetti iterabili, come le liste o altre collezioni, i generatori producono gli elementi uno alla volta e solo quando sono richiesti
- **uso molto efficiente della memoria quando si ha a che fare con grandi quantità di dati**

## Programmazione concorrente

- Python supporta sia la concorrenza a thread (libreria `multithreading`) che quella basata su processi multipli (libreria `multiprocessing`)
- La concorrenza a thread consiste nell'avere thread concorrenti separati che operano all'interno di uno stesso processo. I thread richiedono meno tempo e risorse per essere creati. Il context switching tra thread è meno costoso rispetto a quello tra processi. Questi thread tipicamente accedono i dati condivisi attraverso un accesso serializzato alla memoria. In Python porta a performance migliori in caso di computazioni I/O-bound.
- La concorrenza basata sui processi si ha quando processi separati vengono eseguiti indipendentemente. I processi concorrenti tipicamente condividono i dati mediante IPC anche se possono usare anche la memoria condivisa se il linguaggio o la sua libreria la supportano.

## System programming

- Nella programmazione di sistema i programmi utente che interagiscono strettamente con il sistema operativo e utilizzano i servizi (system call) forniti dal sistema operativo.
- In Python la programmazione di sistema è resa molto semplice dal modulo os.
- Il modulo os serve come un livello astratto tra il programma python e il sistema operativo.
  - Fa in modo che la maggior parte dei comandi siano indipendenti dal sistema operativo.

11

## Network programming

- Python fornisce due livelli di accesso ai servizi di rete.
  - A basso livello si può accedere al supporto di base per i socket fornito dal sistema operativo sottostante.
  - Ad alto livello è possibile utilizzare librerie per accedere a protocolli Internet quali FTP, HTTP, ecc.
- Impareremo ad implementare Client e Server utilizzando un approccio di programmazione ad alto livello.

12

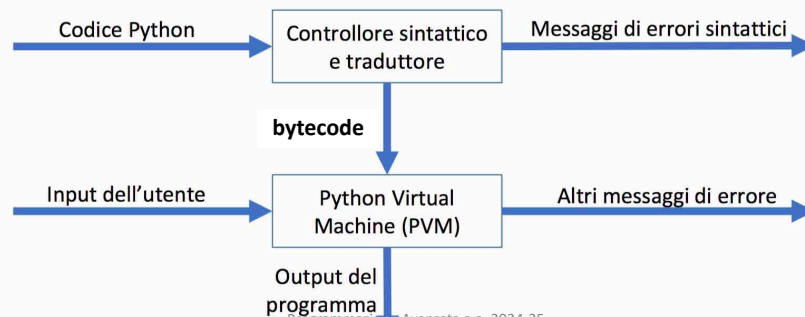
## Il linguaggio Python

- Python è un linguaggio interpretato
- I comandi sono eseguiti da un interprete
  - L'interprete riceve un comando, valuta il comando e restituisce il risultato del comando
- Un programmatore memorizza una serie di comandi in un file di testo a cui faremo riferimento con il termine codice sorgente o script (modulo)
- Convenzionalmente il codice sorgente è memorizzato in un file con estensione `.py`
  - file.py

13

## Come funziona Python?

- L'interprete svolge il ruolo di controllore sintattico e di traduttore
- Il bytecode è la traduzione del codice Python in un linguaggio di basso livello
- È la Python Virtual Machine ad eseguire il bytecode



14

## Versione Python da utilizzare

- Ultima versione Python **3.12.6**
- <https://www.python.org/downloads/>
- **python -V** oppure **python --version**
  - Per sapere quale versione è installata
  - Se sono installate più versioni ci dice quale viene lanciata con il comando **python**
- Shell
- Idle, LiClipse, PyCharm
  - Ambienti di sviluppo integrati in Python

15

## Documentazione Python

- Sito ufficiale Python
  - <https://docs.python.org/3/>
- Tutorial Python
  - <https://docs.python.org/3/tutorial/>
- **Assicuratevi che la documentazione sia per Python 3**

16



## Come scrivere il codice

- Commento introduttivo
- Import dei moduli richiesti dal programma
  - Subito dopo il commento introduttivo
- Inizializzazione di eventuali variabili del modulo
- Definizione delle funzioni
  - Tra cui la funzione main (**non è necessaria**)
- Docstring per ogni funzione definita nel modulo
- Uso di nomi significativi

17

## Esempio di modulo

```
# esempio di modulo: file fact.py

""" questa è la docstring del modulo """

def factorial(n):      # funzione che computa il fattoriale
    result=1          # inizializza la variabile che contiene il risultato
    for k in range(1,n+1):
        result=result*k
    return result     # restituisce il risultato

print("fattoriale di 3:",factorial(3))
print("fattoriale di 1:",factorial(1))
print(__doc__)
```

lo script stampa:

```
6
1
Questa e` la docstring del modulo
```

18

## Funzione main

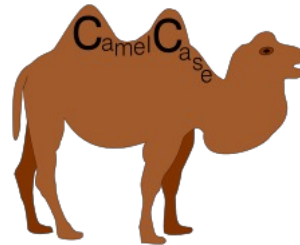
- Non è necessaria introdurla
  - Non succede come in C o Java dove la funzione main è invocata quando il programma è eseguito

19

## Convenzioni

- Nomi di funzioni, metodi e di variabili iniziano sempre con la lettera minuscola
- Nomi di classi iniziano con la lettera maiuscola
- Usare in entrambi i casi la notazione CamelCase

```
userId  
testDomain  
PriorityQueue  
BinaryTree
```



- Nel caso di costanti scrivere il nome tutto in maiuscolo

20

## Identificatori

- Sono case sensitive
- Possono essere composti da lettere, numeri e underscore (\_)
- Un identificatore **non** può iniziare con un numero e **non** può essere una delle seguenti parole riservate

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

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

21

21

## Identificatori in Python 3

- Gli identificatori possono contenere caratteri unicode
  - Ma solo caratteri che somigliano a lettere
- `résumé = "knows Python"`
- `π = math.pi`
- Non funziona il seguente assegnamento
  - `□ = 5.8`

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

22

22

## Tipi delle variabili

- Il **tipo** di una variabile (intero, carattere, virgola mobile, ...) è basato sull'utilizzo della variabile e non deve essere specificato prima dell'utilizzo
- La variabile può essere riutilizzata nel programma e il suo tipo può cambiare in base alla necessità corrente

script	<pre>a = 3 print(a, type(a)) a = "casa" print(a, type(a)) a = 4.5 print(a, type(a))</pre>	output	<pre>3 &lt;class 'int'&gt; casa &lt;class 'str'&gt; 4.5 &lt;class 'float'&gt;</pre>
--------	---	--------	---

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

23

23

## Oggetti in Python

- Python è un linguaggio orientato agli oggetti e le classi sono alla base di tutti i tipi di dati
- Alcune classi predefinite in Python
  - La classe per i numeri interi **int**
  - La classe per i numeri in virgola mobile **float**
  - La classe per le stringhe **str**

`t = 3.8` crea una nuova istanza della classe **float**  
In alternativa possiamo invocare il costruttore `float()`: `t=float(3.8)`

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

24

24

## Oggetti mutable/immutable

- Oggetti il cui valore può cambiare sono chiamati *mutable*
- Una classe è *immutable* se un oggetto della classe una volta inizializzato non può essere modificato in seguito
- Un oggetto contenitore *immutable* che contiene un *riferimento* ad un oggetto *mutable*, può cambiare quando l'oggetto contenuto cambia
  - esempio:
 

```
>>> L=[1,2,3]
>>> t=('a',L)
>>> t
('a', [1, 2, 3])
>>> L.append(4)
>>> t
('a', [1, 2, 3, 4])
```
  - Il contenitore è comunque considerato *immutable* perché la collezione di oggetti che contiene non può cambiare

25

## Classi built-in

Class	Description	Immutable?
<b>bool</b>	Boolean value	✓
<b>int</b>	integer (arbitrary magnitude)	✓
<b>float</b>	floating-point number	✓
<b>list</b>	mutable sequence of objects	
<b>tuple</b>	immutable sequence of objects	✓
<b>str</b>	character string	✓
<b>set</b>	unordered set of distinct objects	
<b>frozenset</b>	immutable form of set class	✓
<b>dict</b>	associative mapping (aka dictionary)	

```
i = int(3)
print(i)
print(i.bit_length())
```

→

```
3
2
```

26

## Classe **bool**

- La classe **bool** è usata per rappresentare i valori booleani **True** e **False**
- Il costruttore **bool()** restituisce **False** di default
- Python permette la creazione di valori booleani a partire da valori non-booleani
  - **bool(foo)**
    - L'interpretazione dipende dal valore di foo
  - Un numero è interpretato come **False** se uguale a 0, **True** altrimenti
  - Sequenze ed altri tipi di contenitori sono valutati **False** se sono vuoti, **True** altrimenti

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

27

27

## Classe **int**

```
i = int(7598234798572495792375243750235437503)
print('numero di bit: ', i.bit_length())
```

output **numero di bit: 123**

- La classe **int** è usata per rappresentare i valori interi di grandezza arbitraria
- Il costruttore **int()** restituisce **0** di default
- È possibile creare interi a partire da **stringhe** che rappresentano numeri in qualsiasi base tra 2 e 35 (2, 3, ..., 9, A, ..., Z)

```
i = int("23", base=4)
print('la variabile vale: ', i)
```

output **la variabile vale: 11**

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

28

28

## Classe **float**

- La classe **float** è usata per rappresentare i valori floating-point in doppia precisione
- Il costruttore **float()** restituisce **0.0** di default
- La classe float ha vari metodi, ad esempio possiamo rappresentare il valore come rapporto di interi

```
f= 0.321123
print(f, '=', f.as_integer_ratio())
```

```
0.321123 = (5784837692560383, 18014398509481984)
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

29

29

## Classe **float**

- L'istruzione `t = 23.7` crea una nuova istanza immutabile della classe **float**
- Lo stesso succede con l'istruzione `t = float(23.7)`
- `t + 4` automaticamente invoca `t.__add__(4)`  
– overloading dell'operatore +

```
f1 = float(3.8)
print('operatore +:', f1+4)
print('metodo __add__:', f1.__add__(4))
```

script

```
operatore +: 7.8
metodo __add__: 7.8
```

output

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

30

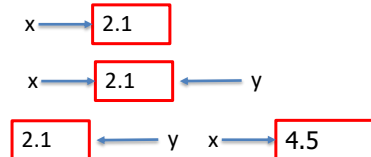
30

## Oggetti immutabili

```
x = 2.1
```

```
y=x
```

```
x=4.5
```



- L'assegnamento `x = 4.5` non modifica il valore di `x`, ma crea una nuova istanza di **float** che contiene il valore 4.5. La variabile `x` fa quindi riferimento a questa nuova istanza di float

31

## Sequenze

- Python le classi **list**, **tuple** e **str** sono tipi **sequenza**
  - Una sequenza rappresenta una collezione di valori in cui l'ordine è rilevante (non significa che gli elementi sono ordinati in modo crescente o decrescente)
  - Ogni elemento della sequenza ha una posizione
    - Se ci sono `n` elementi, il primo elemento è in posizione 0, mentre l'ultimo è in posizione `n-1`

32



## Oggetti iterable

- Un oggetto è **iterable** se
  - Contiene *alcuni elementi*
  - È in grado di *restituire* i suoi elementi uno alla volta
- Stesso concetto di **Iterable** in Java

```
List list = new ArrayList();
//inseriamo qualcosa in list
for(Object o : list){
    //Utilizza o
}
```

Java

```
lst = list([1, 2, 3])
for o in lst:
    //Utilizza o
```

Python

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

33

33

## Oggetti iterable

```
>>> list=[1,2,3,4,10,23,43,5,22,7,9]
>>> list1=[x for x in list if x>4]
>>> list1
[10, 23, 43, 5, 22, 7, 9]
```

equivalente a

```
>>> list=[1,2,3,4,10,23,43,5,22,7,9]
>>> list1=[]
>>> for x in list:
...     if x>4:
...         list1.append(x)
...
>>> list1
[10, 23, 43, 5, 22, 7, 9]
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

34

34

## Classe **list**

- Un'istanza dell'oggetto lista memorizza una sequenza di oggetti
  - Una sequenza di riferimenti (puntatori) ad oggetti nella lista
- Gli elementi di una lista possono essere oggetti arbitrari (incluso l'oggetto **None**)
- Python usa i caratteri **[ ]** come delimitatori di una lista
  - `[]` lista vuota
  - `['red', 'green', 'blue']` lista con tre elementi
  - `[3, 4.9, 'casa']` lista con tre elementi

35

## Classe **list**

- Il costruttore **list()** restituisce una lista vuota di default
- Il costruttore **list()** accetta un qualsiasi parametro iterabile
  - `list('ciao')` produce una lista di singoli caratteri `['c', 'i', 'a', 'o']`
- Una lista è una sequenza concettualmente simile ad un array
  - una lista di lunghezza n ha gli elementi indicizzati da 0 ad n-1
- Le liste hanno la capacità di espandersi e contrarsi secondo la necessità corrente

36

## Metodi di **list**

- `list.append(x)`
  - Aggiunge l'elemento `x` alla fine della lista
- `list.extend(iterable)`
  - Estende la lista aggiungendo tutti gli elementi dell'oggetto *iterable*
  - `a.extend(b)` a equivalente `a[len(a):] = b`
- `list.insert(i, x)`
  - Inserisce l'elemento `x` nella posizione `i`
  - `p.insert(0, x)` inserisce `x` all'inizio della lista `p`
  - `p.insert(len(p), x)` inserisce `x` alla fine della lista `p` (equivalente a `p.append(x)`)

`len(a)` restituisce il numero degli elementi in `a`

37

## Concatenazione di liste

La funzione `id()` fornisce l'identità di un oggetto, cioè un intero che identifica univocamente l'oggetto per la sua intera vita. In molte implementazioni del linguaggio Python, l'identità dell'oggetto è il suo indirizzo in memoria.

```
a = list([1, 2, 3])
print('id =', id(a), ' a =', a)
b = list([4, 5])
print('id =', id(b), ' b =', b)
a.extend(b)
print('id =', id(a), ' a =', a)
a += b    #non crea un nuovo oggetto
print('id =', id(a), ' a =', a)
a = a + b #crea un nuovo oggetto
print('id =', id(a), ' a =', a)
```

```
id = 4321719112 a = [1, 2, 3]
id = 4321719176 b = [4, 5]
id = 4321719112 a = [1, 2, 3, 4, 5]
id = 4321719112 a = [1, 2, 3, 4, 5, 4, 5]
id = 4321697160 a = [1, 2, 3, 4, 5, 4, 5, 4, 5]
```

`a += b`

≠

`a = a + b`

38

## Metodi di **list**

- **list.remove(x)**
  - Rimuove la prima occorrenza dell'elemento x dalla lista. Genera un errore se x non c'è nella lista
- **list.pop(i)**
  - Rimuove l'elemento in posizione i e lo restituisce
  - **a.pop()** rimuove l'ultimo elemento della lista
- **list.clear()**
  - Rimuove tutti gli elementi dalla lista

## Metodi di **list**

- **list.index(x, start, end)**
  - Restituisce l'indice della prima occorrenza di x compreso tra start ed end (opzionali)
  - L'indice è calcolato a partire dall'inizio (indice 0) della lista
- **list.count(x)**
  - Restituisce il numero di volte che x è presente nella lista
- **list.reverse()**
  - Inverte l'ordine degli elementi della lista
- **list.copy()**
  - Restituisce una copia della lista

## Metodi di **list**

Syntax	Description
L.append(x)	Appends item x to the end of list L
L.count(x)	Returns the number of times item x occurs in list L
L.extend(m) L += m	Appends all of iterable m's items to the end of list L; the operator += does the same thing
L.index(x, start, end)	Returns the index position of the leftmost occurrence of item x in list L (or in the start:end slice of L); otherwise, raises a ValueError exception
L.insert(i, x)	Inserts item x into list L at index position int i
L.pop()	Returns and removes the rightmost item of list L
L.pop(i)	Returns and removes the item at index position int i in L
L.remove(x)	Removes the leftmost occurrence of item x from list L, or raises a ValueError exception if x is not found
L.reverse()	Reverses list L in-place
L.sort(...)	Sorts list L in-place; this method accepts the same key and reverse optional arguments as the built-in sorted()

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

41

41

## Esempio

codice

```
l = [3, '4', 'casa']
l.append(12)
print('l =', l)
d = l
print('d =', d)
d[3] = 90
print('d =', d)
print('l =', l)
```

stampa

```
l = [3, '4', 'casa', 12]
d = [3, '4', 'casa', 12]
d = [3, '4', 'casa', 90]
l = [3, '4', 'casa', 90]
```

d ed l fanno riferimento  
allo stesso oggetto

```
a = [3, 4, 5, 4, 4, 6]
print('a =', a)
print('Indice di 4 in a:', a.index(4))
print('Indice di 4 in a tra 3 e 6:', a.index(4, 3, 6))
```

```
a = [3, 4, 5, 4, 4, 6]
Indice di 4 in a: 1
Indice di 4 in a tra 3 e 6: 3
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

42

42

## Ordinare una lista

- `list.sort(key=None, reverse=False)`
  - Ordina gli elementi della lista, `key` e `reverse` sono opzionali
  - A `key` si assegna il nome di una funzione con un solo argomento che è usata per estrarre da ogni elemento la chiave con cui eseguire il confronto
  - A `reverse` si può assegnare il valore `True` se si vuole che gli elementi siano in ordine decrescente

```
a = [3, 4, 5, 4, 4, 6]
a.sort(reverse=True)
print(a)
[6, 5, 4, 4, 4, 3]
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

43

43

## Ordinare una lista

```
>>> x=["anna","michele","carla","antonio","fabio"]
>>> x
['anna', 'michele', 'carla', 'antonio', 'fabio']
>>> x.sort()
>>> x
['anna', 'antonio', 'carla', 'fabio', 'michele']
>>> x.sort(reverse=True)
>>> x
['michele', 'fabio', 'carla', 'antonio', 'anna']
>>> x.sort(key=len)
>>> x
['anna', 'fabio', 'carla', 'michele', 'antonio']
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

44

44

## Classe **tuple**

- Fornisce una versione immutabile di una lista
- Python usa i caratteri **( )** come delimitatori di una tupla
- L'accesso agli elementi della tupla avviene come per le liste
- La tupla vuota è **()**
- La tupla **(12,)** contiene solo l'elemento 12

```
t = (3, 4, 5, '4', 4, '6')
print('t =', t)
print('Lunghezza t =', len(t))
```



```
t = (3, 4, 5, '4', 4, '6')
Lunghezza t = 6
```

45

## **tuple** packing/unpacking

- Il packing è la creazione di una tupla
- L'**unpacking** è la creazione di variabili a partire da una tupla

```
t = (1, 's', 4)
x, y, z = t
print('t =', t, type(t))
print('x =', x, type(x))
print('y =', y, type(y))
print('z =', z, type(z))
```



```
t = (1, 's', 4) <class 'tuple'>
x = 1 <class 'int'>
y = s <class 'str'>
z = 4 <class 'int'>
```

46

## Ancora su mutable/immutable

```
lst = ['a', 1, 'casa']
tpl = (lst, 1234)
print('list =', lst)
print('tuple =', tpl)
try:
    tpl[0] = 0
except Exception as e:
    print(e)
print(tpl[0])
lst.append('nuovo')
print('list =', lst)
print('tuple =', tpl)
```

```
list = ['a', 1, 'casa']
tuple = (['a', 1, 'casa'], 1234)
```

**'tuple' object does not support item assignment'**  
 ['a', 1, 'casa']

```
list = ['a', 1, 'casa', 'nuovo']
tuple = (['a', 1, 'casa', 'nuovo'], 1234)
```

47

Programmazione avanzata a.a. 2024-25

A. De Bonis

## Introduzione a Python (II parte)

48



## Classe **str**

- Le stringhe (sequenze di caratteri) possono essere racchiuse da apici singoli o apici doppi
- Si usano tre apici singoli o doppi per stringhe che contengono newline (sono su più righe)
- Nei manuali dettagli sui metodi di **str**

```
s = """Il Principe dell'Alba
si mette in cammino venti
minuti prima delle quattro."""
print(s)
```

```
Il Principe dell'Alba
si mette in cammino venti
minuti prima delle quattro.
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

49

49

## Classe **set**

- La classe **set** rappresenta la nozione matematica dell'insieme
  - Una collezione di elementi **senza duplicati** e senza un particolare ordine
- Può contenere **solo** istanze di oggetti **immutable** (più precisamente hashable)
- Si usano le parentesi graffe per indicare l'insieme { }
- L'insieme vuoto è creato con **set()**

```
ins = {2, 4, '4'}
print(ins)
```

→ {2, '4', 4}

L'ordine dell'output dipende dalla rappresentazione interna di set

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

50

50

## Classe **set**

- Il costruttore **set()** accetta un qualsiasi parametro iterabile
  - `a=set('buongiorno')` → `a={'o', 'u', 'i', 'b', 'r', 'g', 'n'}`
- `len(a)` restituisce il numero di elementi di `a`
- `a.add(x)`
  - Aggiunge l'elemento `x` all'insieme `a`
- `a.remove(x)`
  - Rimuove l'elemento `x` dall'insieme `a`
- Altri metodi li vediamo in seguito
  - Dettagli sul manuale

## Classe **frozenset**

- È una classe immutabile del tipo **set**
  - Si può avere un set di frozenset
- Stessi metodi ed operatori di **set**
  - Si possono eseguire facilmente test di (non) appartenenza, operazioni di unione, intersezione, differenza, ...
- Dettagli maggiori quando analizzeremo gli operatori
  - Per ogni operatore esiste anche la *versione* metodo

## Classe **dict**

- La classe **dict** rappresenta un dizionario
  - Un insieme di coppie (chiave, valore)
  - Le chiavi devono essere distinte e di un tipo immutabile (piu' precisamente hashable)
  - Implementazione in Python simile a quella di set
- Il dizionario vuoto è rappresentato da **{ }**
  - **d={ }** crea un dizionario vuoto
- Un dizionario si crea inserendo nelle **{ }** una serie di coppie **chiave:valore** separate da virgola
  - `d = {'ga' : 'Irish', 'de' : 'German'}`
  - Alla chiave **de** è associato il valore **German**
- Il costruttore accetta una sequenza di coppie (chiave, valore) come parametro
  - `d = dict(pairs)` dove `pairs = [('ga', 'Irish'), ('de', 'German')]`.

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

53

53

## Oggetti hashable

- La seguente definizione è una traduzione di quella presente nella documentazione.
- Un oggetto è *hashable* se ha un valore hash che non cambia mai durante il suo tempo di vita (ha bisogno di un metodo `__hash__()`) e puo' essere confrontato con altri oggetti (ha bisogno del metodo `__eq__()`). Oggetti Hashable uguali devono avere lo stesso valore hash.
- Un oggetto hashable è utilizzabile come chiave di un dizionario o come elemento di un oggetto set perche' queste strutture dati usano il valore hash internamente.
- La maggior parte degli oggetti built-in immutabile sono hashable; contenitori mutable (come liste e dizionari) non lo sono; contenitori immutabile (quali le tuple e i frozenset) sono hashable solo se i loro elementi lo sono. Oggetti che sono istanze di classi definite dall'utente sono hashable per default. Questi oggetti, se confrontati tra di loro, risultano tutti diversi (a meno che non vengano confrontati con se stessi) e il loro valore hash è derivato dal loro `id()`.

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

54

54

## Esempi classe **dict**

```

>>> d={"k1":250,"k2":340}
>>> d['k1']
250
>>> d
{'k1': 250, 'k2': 340}
>>> d={"k1":250,"k2":340}
>>> d
{'k1': 250, 'k2': 340}
>>> d['k1']
250
>>> d['k3']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'k3'
>>> d['k3']=34
>>> d['k3']
34
>>> d['k2']
340
>>> d['k2']=12000
>>> d['k2']
12000

```

chiave 'k3' non  
presente nel  
dizionario

inserisco l'entrata  
con chiave 'k3' e  
valore 34

modifico valore  
associato alla chiave  
'k2'

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

55

55

## Alcuni metodi classe **dict**

- **diz.clear()**
  - Rimuove tutti gli elementi da **diz**
- **diz.copy()**
  - Restituisce una copia superficiale (shallow) di **diz**
- **diz.get(k)**
  - Restituisce il valore associato alla chiave **k** (None se **k** non in **diz**)
- **diz.pop(k)** (KeyError se **k** non è in **diz**)
  - Rimuove la chiave **k** da **diz** e restituisce il valore ad essa associato
- **diz.update([other])**
  - Aggiorna **diz** con le coppie chiave/valore in **other**, sovrascrive i valori associati a chiavi già esistenti
  - update accetta come input o un dizionario o un oggetto iterabile di coppie chiave/valore (le coppie possono essere tuple o un altro oggetto iterabile di lunghezza due)

**pop e get possono prendere in input un secondo argomento il cui valore viene restituito nel caso in cui **k** non sia nel dizionario**

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

56

56

## Esempio di update

```
tel = {'irv': 4127, 'guido': 4127, 'jack': 4098}
print('tel =', tel)
tel2 = {'guido': 1111, 'john': 666}
print('tel2 =', tel2)
tel.update(tel2)
print('tel =', tel)
tel.update([('mary', 1256)])
print('tel =', tel)
```

```
tel = {'irv': 4127, 'guido': 4127, 'jack': 4098}
tel2 = {'guido': 1111, 'john': 666}
tel = {'guido': 1111, 'john': 666, 'irv': 4127, 'jack': 4098}
tel = {'guido': 1111, 'mary': 1256, 'john': 666, 'irv': 4127, 'jack': 4098}
```

57

## Alcuni metodi classe **dict**

- **diz.keys()**
  - restituisce un nuovo oggetto *view* delle chiavi del dizionario
- **diz.values()**
  - restituisce un nuovo oggetto *view* dei valori del dizionario
- **diz.items(k)**
  - restituisce un nuovo oggetto *view* delle coppie (chiave, valore) del dizionario
- Gli oggetti *view* forniscono una "vista" dinamica delle entrate del dizionario. Se il dizionario viene modificato, le modifiche si riflettono negli oggetti *view*.
- Gli oggetti *view* possono essere iterati e permettono di effettuare test di appartenenza.

58

## Esempi classe dict

```
tel = {'jack': 4098, 'sape': 4139}
tel['guido'] = 4127
print('tel =', tel)
tel['irv'] = 4127
print('tel =', tel)
del tel['sape']
print('tel =', tel)
```

```
tel = {'jack': 4098, 'guido': 4127, 'sape': 4139}
tel = {'jack': 4098, 'irv': 4127, 'guido': 4127, 'sape': 4139}
tel = {'jack': 4098, 'irv': 4127, 'guido': 4127}
```

```
chiavi = tel.keys()
print('chiavi =', chiavi)
valori = tel.values()
print('valori =', valori)
for i in chiavi:
    print(i)
```

```
chiavi = dict_keys(['guido', 'irv', 'jack'])
valori = dict_values([4127, 4127, 4098])
guido
irv
jack
```

```
for i in tel.keys():
    print(i)
```

```
elementi = tel.items()
for k,v in elementi:
    print(k,v)
```

```
irv 4127
guido 4127
jack 4098
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

59

## shallow vs deep copy

- Dal manuale Python
  - A **shallow** copy constructs a new compound object and then inserts references into it to the objects found in the original
  - *Costruisce un nuovo oggetto composto e inserisce in esso i riferimenti agli oggetti presenti nell'originale*
  - A **deep** copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original
  - *Costruisce un nuovo oggetto composto e ricorsivamente inserisce in esso le copie degli oggetti presenti nell'originale*

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

60

## Problemi con deep copy

- Problem
  1. Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop
    - Ad esempio, se un oggetto a contiene un riferimento a se stesso allora una copia deep di a causa un loop
  2. Because deep copy copies everything it may copy too much, e.g., even administrative data structures that should be shared even between copies
    - The [deepcopy\(\)](#) function avoids these problems by:
      - keeping a memo dictionary of objects already copied during the current copying pass; and
      - letting user-defined classes override the copying operation or the set of components copied.

61

## Problemi con deep copy

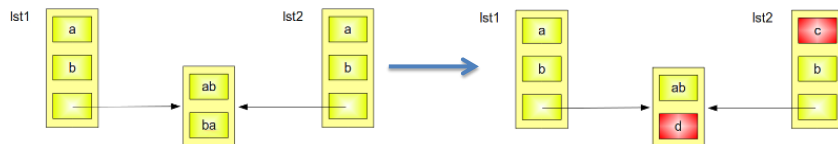
- Problemi con deepcopy
  1. A causa di oggetti ricorsivi: oggetti che direttamente o indirettamente contengono riferimenti a se stessi possono causare un loop
  2. A causa del fatto che possono essere copiati anche dati che dovrebbero essere condivisi tra le varie copie.
- La funzione [deepcopy\(\)](#) evita i suddetti problemi in questo modo:
  - mantenendo un “memo” (dizionario) degli oggetti già copiati
  - permettendo alle classi definite dall’utente di fare l’override dell’operazione di copia.
    - per la copia deep, la classe deve definire `__deepcopy__()`

62

## Esempio shallow/deep copy

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1.copy() #metodo della classe list
print('lista1 =', lst1)
print('lista2 =', lst2)
lst2[0] = 'c'
lst2[2][1] = 'd'
print('lista1 =', lst1)
print('lista2 =', lst2)
```

```
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['a', 'b', ['ab', 'ba']]
lista1 = ['a', 'b', ['ab', 'd']]
lista2 = ['c', 'b', ['ab', 'd']]
```



Programmazione Avanzata a.a. 2024-25  
A. De Bonis

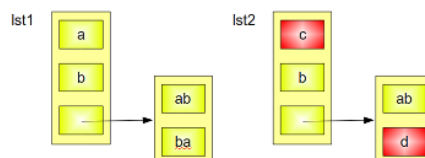
63

63

## Esempio shallow/deep copy

```
from copy import deepcopy
lst1 = ['a','b',['ab','ba']]
lst2 = deepcopy(lst1)
print('lista1 =', lst1)
print('lista2 =', lst2)
lst2[0] = 'c'
lst2[2][1] = 'd'
print('lista1 =', lst1)
print('lista2 =', lst2)
```

```
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['a', 'b', ['ab', 'ba']]
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['c', 'b', ['ab', 'd']]
```



Programmazione Avanzata a.a. 2024-25  
A. De Bonis

64

64



## Espressioni ed operatori

- Espressioni esistenti possono essere combinate con simboli speciali o parole chiave (operatori)
- La semantica dell'operatore dipende dal tipo dei suoi operandi

```

a=3
b=4
c=a+b
print('a+b =', c)
a='ciao '
b='mondo'
c=a+b
print('a+b =', c)

```

→

```

a+b = 7
a+b = ciao mondo

```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

65

65

## Operatori aritmetici

```

+ addition
- subtraction
* multiplication
/ true division
// integer division
% the modulo operator

```

```
// e % definiti anche per
numeratore o denominatore
negativo. Dettagli sul manuale
```

- Per gli operatori +, -, \*
  - Se entrambi gli operandi sono `int`, il risultato è `int`
  - Se uno degli operandi è `float`, il risultato è `float`
- Per la divisione *vera* /
  - Il risultato è sempre float
- Per la divisione intera // (**floor division**)
  - Il risultato (`int`) è la parte intera della divisione

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

66

66

## Operatori logici

### Operatori di uguaglianza

- Python supporta i seguenti operatori logici

**not** unary negation  
**and** conditional and  
**or** conditional or

- Python supporta i seguenti operatori di uguaglianza

**is** same identity  
**is not** different identity  
**==** equivalent  
**!=** not equivalent

67

## Operatori di uguaglianza

- L'espressione **a is b** risulta vera solo se a e b sono alias dello stesso oggetto
- L'espressione **a == b** risulta vera anche quando gli identificatori a e b si riferiscono ad oggetti che possono essere considerati equivalenti
  - Due oggetti che **contengono** gli stessi valori

68

## Esempio

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1
if lst1 is lst2:
    print('Oggetti identici')
else:
    print('Oggetti distinti')

if lst1 == lst2:
    print('Oggetti equivalenti')
else:
    print('Oggetti non equivalenti')
```

Oggetti identici  
Oggetti equivalenti

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1.copy()
if lst1 is lst2:
    print('Oggetti identici')
else:
    print('Oggetti distinti')

if lst1 == lst2:
    print('Oggetti equivalenti')
else:
    print('Oggetti non equivalenti')
```

Oggetti distinti  
Oggetti equivalenti

69

## Operatori di confronto

- Python supporta i seguenti operatori di confronto

< less than  
<= less than or equal to  
> greater than  
>= greater than or equal to

- Per gli interi hanno il significato atteso
- Per le stringhe sono case-sensitive e considerano l'ordinamento lessicografico
- Per sequenze ed insiemi assumono un significato particolare (dettagli in seguito)

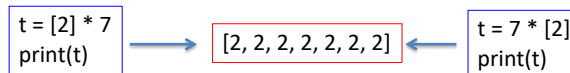
70

## Operatori per sequenze

### list, tuple e str

- I tipi sequenza predefiniti in Python supportano i seguenti operatori

<code>s[j]</code>	element at index $j$
<code>s[start:stop]</code>	slice including indices $[start, stop)$
<code>s[start:stop:step]</code>	slice including indices $start, start + step, start + 2*step, \dots$ , up to but not equalling or stop
<code>s + t</code>	concatenation of sequences
<code>k * s</code>	shorthand for $s + s + s + \dots$ (k times)
<code>val in s</code>	containment check
<code>val not in s</code>	non-containment check



Programmazione Avanzata a.a. 2024-25  
A. De Bonis

71

71

## Indici negativi

- Le sequenze supportano anche indici negativi
- `s[-1]` si riferisce all'ultimo elemento di `s`
- `s[-2]` si riferisce al penultimo elemento di `s`
- `s[-3]` ...
- `s[j] = val` sostituisce il valore in posizione `j`
- `del s[j]` rimuove l'elemento in posizione `j`

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

72

72

## Confronto di sequenze

- Le sequenze possono essere confrontate in base all'ordine lessicografico
  - Il confronto è fatto elemento per elemento
  - Ad esempio,  $[5, 6, 9] < [5, 7]$  (**True**)
  - $s == t$  equivalent (element by element)
  - $s != t$  not equivalent
  - $s < t$  lexicographically less than
  - $s <= t$  lexicographically less than or equal to
  - $s > t$  lexicographically greater than
  - $s >= t$  lexicographically greater than or equal to

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

73

73

## Operatori per insiemi

- Le classi **set** e **frozenset** supportano i seguenti operatori

<code>key in s</code>	containment check
<code>key not in s</code>	non-containment check
<code>s1 == s2</code>	s1 is equivalent to s2
<code>s1 != s2</code>	s1 is not equivalent to s2
<code>s1 &lt;= s2</code>	s1 is subset of s2
<code>s1 &lt; s2</code>	s1 is proper subset of s2
<code>s1 &gt;= s2</code>	s1 is superset of s2
<code>s1 &gt; s2</code>	s1 is proper superset of s2
<code>s1   s2</code>	the union of s1 and s2
<code>s1 &amp; s2</code>	the intersection of s1 and s2
<code>s1 - s2</code>	the set of elements in s1 but not s2
<code>s1 ^ s2</code>	the set of elements in precisely one of s1 or s2

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

74

74

## Operatori per dizionari

- La classe **dict** supporta i seguenti operatori

<code>d[key]</code>	value associated with given key
<code>d[key] = value</code>	set (or reset) the value associated with given key
<code>del d[key]</code>	remove key and its associated value from dictionary
<code>key in d</code>	containment check
<code>key not in d</code>	non-containment check
<code>d1 == d2</code>	d1 is equivalent to d2
<code>d1 != d2</code>	d1 is not equivalent to d2

## Precedenza degli operatori

priorità

Operator Precedence		
	Type	Symbols
1	member access	<code>expr.member</code>
2	function/method calls container subscripts/slices	<code>expr(...)</code> <code>expr[...]</code>
3	exponentiation	<code>**</code>
4	unary operators	<code>+expr</code> , <code>-expr</code> , <code>~expr</code>
5	multiplication, division	<code>*</code> , <code>/</code> , <code>//</code> , <code>%</code>
6	addition, subtraction	<code>+</code> , <code>-</code>
7	bitwise shifting	<code>&lt;&lt;</code> , <code>&gt;&gt;</code>
8	bitwise-and	<code>&amp;</code>
9	bitwise-xor	<code>^</code>
10	bitwise-or	<code> </code>
11	comparisons containment	<code>is</code> , <code>is not</code> , <code>==</code> , <code>!=</code> , <code>&lt;</code> , <code>&lt;=</code> , <code>&gt;</code> , <code>&gt;=</code> <code>in</code> , <code>not in</code>
12	logical-not	<code>not expr</code>
13	logical-and	<code>and</code>
14	logical-or	<code>or</code>
15	conditional	<code>val1 if cond else val2</code>
16	assignments	<code>=</code> , <code>+=</code> , <code>-=</code> , <code>*=</code> , etc.

## Assegnamento esteso

- In C o Java gli operatori binari ammettono una versione *contratta*
  - $i += 3$  è equivalente a  $i = i + 3$
- Tale caratteristica esiste anche in Python
  - Per i tipi immutable si crea un nuovo oggetto a cui si assegna un nuovo valore
  - Alcuni tipi di dato (e.g., list) ridefiniscono la semantica dell'operatore  $+=$

## Chaining

- Assegnamento
  - In Python è permesso l'assegnamento concatenato
  - $x = y = z = 0$
- Operatori di confronto
  - In Python è permesso  $1 < x + y <= 9$
  - Equivalente a  $(1 < x+y)$  **and**  $(x + y <= 9)$ , ma l'espressione  $x+y$  è calcolata una sola volta

```
x=y=5
if 3 < x+y <= 10:
    print('interno')
else:
    print('esterno')
```

## Esempio += per **list**

```
alpha = [1, 2, 3]
beta = alpha
print('alpha =', alpha)
print('beta =', beta)
beta += [4, 5]
print('beta =', beta)
beta = beta + [6, 7]
print('beta =', beta)
print('alpha =', alpha)
```



```
alpha = [1, 2, 3]
beta = [1, 2, 3]
beta = [1, 2, 3, 4, 5]
beta = [1, 2, 3, 4, 5, 6, 7]
alpha = [1, 2, 3, 4, 5]
```

beta += [4, 5] estende la lista originale

Equivalente a  
beta.extend([4,5])

beta = beta + [6, 7] riassegna beta ad una nuova lista

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

79

79

## Riferimenti

- M. Summerfield, "Programming in Python 3. A Complete Introduction to the Python Language", Addison-Wesley
- M. Lutz, "Learning Python», 5th Edition, O'Reilly
- Altro materiale sarà indicato in seguito

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

80

80



## Controllo del flusso in Python

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

81

81

## Blocchi di codice

- In Python i blocchi di codice non sono racchiusi tra parentesi graffe come in C o Java
- In Python per definire i blocchi di codice o il contenuto dei cicli si utilizza l'indentazione
  - Ciò migliora la leggibilità del codice, ma all'inizio può confondere il programmatore

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

82

82

## Indentazione del codice: Spazi o tab

- Il metodo preferito è indentare utilizzando spazi (di norma 4)
- Il tab può essere diverso tra editor differenti
- In Python 3 non si possono mischiare nello stesso blocco spazi e tab
  - In Python 2 era permesso

Stile per Codice Python

<https://www.python.org/dev/peps/pep-0008/#tabs-or-spaces>

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

83

83

## if elif ... else

```
if first_condition:           codice indentato
    first_body
```

```
elif second_condition:
    second_body
```

```
elif third_condition:
    third_body
```

```
else:
    fourth_body
```

Se il blocco è costituito da una sola istruzione, allora può andare subito dopo i due punti

**elif** ed **else** sono opzionali

```
if x < y and x < z:
    print('x è il minimo')
elif y < z:
    print('y è il minimo')
else:
    print('z è il minimo')
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

84

84

## Esempi

```
print('inizio')
if 5>3:
    print(1)
    print(2)
    print(3)
    print(4)
else:
    print(5)
    print(6)
    print(7)
print('fine')
```



```
inizio
1
2
3
4
fine
```

```
print('inizio')
if 5>3:
    print(1)
    print(2)
    print(3)
    print(4)
else:
    print(5)
    print(6)
    print(7)
print('fine')
```

**ERRORE**

85

## while

**while** *condition*:  
*body*

```
j=0
while j < len(data) and data[j] != X:
    j += 1
```

```
while a<b:
    print(a)
    a=a+1
```

86

## for ... in

```
for element in iterable:
    body
```

# body may refer to 'element' as an identifier

```
total = 0
for val in data:
    total += val
```

```
biggest = 0
for val in data:
    if val > biggest:
        biggest = val
```

87

## range()

- range(n) genera una lista di interi compresi tra 0 ed n-1
  - range(start, stop, step)
- Utile quando vogliamo iterare in una sequenza di dati utilizzando un indice
  - for i in range(n)

```
>>> list(range(1,10,3))
[1, 4, 7]
```

```
for i in range(0, -10, -2): print(i)
```

```
0
-2
-4
-6
-8
```

```
big_index = 0
for j in range(len(data)):
    if data[j] > data[big_index]:
        big_index = j
```

88

## Esempi

```
# Stampa la lunghezza delle
# parole in una lista
words = ['cat', 'window', 'look']
for w in words:
    print(w, len(w))
```

```
cat 3
window 6
look 4
```

```
for _ in range(1,6):
    print('ciao')
```

```
ciao
ciao
ciao
ciao
ciao
```

```
# Cicla su una copia della lista
for w in words[:]:
    if len(w) == 4:
        words.insert(0, w)
    print(words)
```

```
# Cicla su sulla stessa lista
for w in words:
    if len(w) == 4:
        words.insert(0, w)
    print(words)
```

Crea una lista infinita

```
['look', 'cat', 'window', 'look']
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

89

## break e continue

- **break** termina immediatamente un ciclo **for** o **while**, l'esecuzione continua dall'istruzione successiva al **while/for**
- **continue** interrompe l'iterazione corrente di un ciclo **for** o **while** e continua verificando la condizione del ciclo

```
found = False
for item in data:
    if item == target:
        found = True
        break
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

90

## Clausola **else** e cicli

- Utilizzata con cicli che prevedono un **break**
- La clausola **else** è eseguita quando si esce dal ciclo ma non a causa del **break**

```
n=3
for x in [4, 5, 7, 8, 10]:
    if x % n == 0:
        print(x, 'è un multiplo di ', n)
        break
else:
    print('non ci sono multipli di', n, 'nella lista')
```

Con n=2 invece

4 è un multiplo di 2

non ci sono multipli di 3 nella lista

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

91

91

## Python: if *abbreviato*

- In C/Java/C++ esiste la forma abbreviata dell'if  
massimo = a > b ? a : b
- Anche Python supporta questa forma, ma la sintassi è differente

massimo = a if (a > b) else b

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

92

92

## List Comprehension

- *Comprensione di lista*
- Costrutto sintattico di Python che agevola il programmatore nella creazione di una lista a partire dall'elaborazione di un'altra lista
  - Si possono generare tramite comprehension anche
    - Insiemi
    - Dizionari

[ expression for value in iterable if condition ]

## List Comprehension

- *expression* e *condition* possono dipendere da *value*
- La parte **if** è opzionale
  - In sua assenza, si considerano tutti i *value* in iterable
  - Se *condition* è vera, il risultato di *expression* è aggiunto alla lista
- [ *expression* for *value* in iterable if *condition* ] è equivalente a

```
result = []
for value in iterable:
    if condition:
        result.append(expression)
```

## Esempi

Lista dei quadrati dei numeri compresi tra 1 ed n

```
squares = [k*k for k in range(1, n+1)]
```

Lista dei divisori del numero n

```
factors = [k for k in range(1,n+1) if n % k == 0]
```

```
[str(round(pi, i)) for i in range(1, 6)]
```

```
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

95

95

## Doppia comprehension

```
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

```
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

```
combs = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            combs.append((x, y))
```

```
a = [(x, y) for x in [1,2,3] for y in ['a', 'b', 'c']]
print(a)
```

```
[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (3, 'a'), (3, 'b'), (3, 'c')]
```

Programmazione Avanzata a.a. 2024-25  
A. De Bonis

96

96



## Doppia comprehension

```
matrix = [ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ]
print(matrix)
transposed = [[row[i] for row in matrix] for i in range(4)]
print(transposed)
```

```
[ [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12]]
```

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

## Altri tipi di comprehension

- list comprehension  
[ k\*k for k in range(1, n+1) ]
- set comprehension  
{ k\*k for k in range(1, n+1) }
- dictionary comprehension  
{ k : k\*k for k in range(1, n+1) }

esempio:

```
{ k : k*k for k in range(1, 10) }
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```