

Programmazione avanzata a.a. 2020-21

A. De Bonis

Introduzione a Python (I parte)

Programmazione Avanzata a.a. 2020-21
A. De Bonis

1

1

Informazioni utili

- Il sito Web del corso:
 - www.di-srv.unisa.it/professori/debonis/debonis1/progAv2020-21/
- Il mio studio: **numero 44, quarto piano, stecca 7**
 - per il momento gli studenti non possono accedervi
- L'orario di ricevimento: on-line

Programmazione Avanzata a.a. 2020-21
A. De Bonis

2

2

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



Programmazione Avanzata a.a. 2020-21
A. De Bonis

3

3

Indice PYPL

[Creato analizzando quanto spesso tutorial sul linguaggio sono cercati su Google](#)

Worldwide, Sept 2020 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	31.56 %	+2.9 %
2		Java	16.4 %	-3.1 %
3		Javascript	8.38 %	+0.3 %
4		C#	6.5 %	-0.8 %
5		PHP	5.85 %	-0.5 %
6		C/C++	5.8 %	+0.0 %
7		R	4.08 %	+0.3 %
8		Objective-C	2.79 %	+0.2 %
9		Swift	2.35 %	-0.1 %
10		TypeScript	1.92 %	+0.1 %

4

4

Indice PYPL

Creato analizzando quanto spesso tutorial sul linguaggio sono cercati su Google

Worldwide, Sept 2019 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	29.21 %	+4.6 %
2		Java	19.9 %	-2.2 %
3		Javascript	8.39 %	+0.0 %
4		C#	7.23 %	-0.6 %
5		PHP	6.69 %	-1.0 %
6		C/C++	5.8 %	-0.4 %
7		R	3.91 %	-0.2 %
8		Objective-C	2.63 %	-0.7 %
9		Swift	2.46 %	-0.3 %
10		Matlab	1.82 %	-0.2 %

Programmazione Avanzata a.a. 2020-21
A. De Bonis

5

5

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

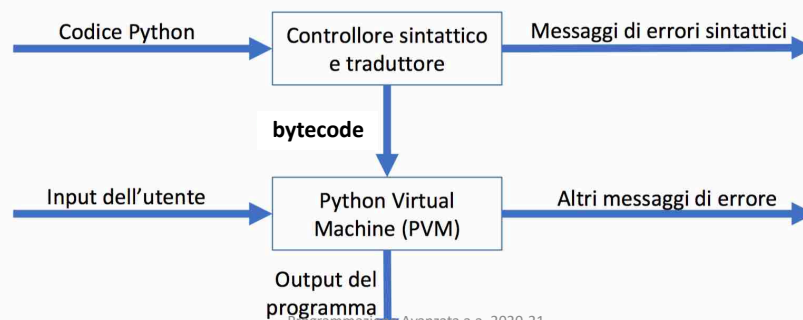
Programmazione Avanzata a.a. 2020-21
A. De Bonis

6

6

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



Programmazione Avanzata a.a. 2020-21
A. De Bonis

7

7

Versione Python da utilizzare

- Ultima versione Python 3.8.5
<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

Programmazione Avanzata a.a. 2020-21
A. De Bonis

8

8

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**

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

Esempio di modulo

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

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("fattoriale di 0:",factorial(0))
```

Programmazione Avanzata a.a. 2020-21
A. De Bonis

11

11

Funzione main

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

Programmazione Avanzata a.a. 2020-21
A. De Bonis

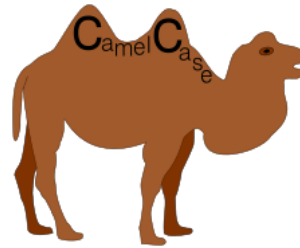
12

12

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

Programmazione Avanzata a.a. 2020-21
A. De Bonis

13

13

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. 2020-21
A. De Bonis

14

14

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. 2020-21
A. De Bonis

15

15

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

```
a = 3
print(a, type(a))
a = "casa"
print(a, type(a))
a = 4.5
print(a, type(a))
```

output

```
3 <class 'int'>
casa <class 'str'>
4.5 <class 'float'>
```

Programmazione Avanzata a.a. 2020-21
A. De Bonis

16

16

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)`

17

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
 - Il contenitore è comunque considerato *immutable* perché la collezione di oggetti che contiene non può cambiare

18

Classi built-in

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

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

→

```
3
2
```

Programmazione Avanzata a.a. 2020-21
A. De Bonis

19

19

Classe **bool**

```
b = bool(False)
if b == False:
    print('La variabile b è ', b)
else:
    print('La variabile b è True')
```

- 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
 - I numeri sono interpretati come **False** se uguali a 0, **True** altrimenti
 - Sequenze ed altri tipi di contenitori sono valutati **False** se sono vuoti, **True** altrimenti

Programmazione Avanzata a.a. 2020-21
A. De Bonis

20

20

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**

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)

Classe **float**

- L'istruzione `t = 23.7` crea una nuova istanza immutabile della classe **float**
- Lo stesso succede con l'istruzione `t = float(3.8)`
- `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

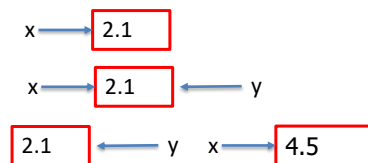
Programmazione Avanzata a.a. 2020-21
A. De Bonis

23

23

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`

Programmazione Avanzata a.a. 2020-21
A. De Bonis

24

24

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

25

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

26

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]
```

27

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

28

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

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)** è equivalente a $a[\text{len}(a):] = b$
- **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

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

a += b
≠
a = a + b

Programmazione Avanzata a.a. 2020-21
A. De Bonis

31

31

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

Programmazione Avanzata a.a. 2020-21
A. De Bonis

32

32

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

33

Metodi di **list**

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

34

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. 2020-21
A. De Bonis

35

35

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. 2020-21
A. De Bonis

36

36

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. 2020-21
A. De Bonis

37

37

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
```

oppure
t = 3,4,5,'4',4,'6'

Programmazione Avanzata a.a. 2020-21
A. De Bonis

38

38

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'>
```

Programmazione Avanzata a.a. 2020-21
A. De Bonis

39

39

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)
```

Programmazione Avanzata a.a. 2020-21
A. De Bonis

40

40

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. 2020-21
A. De Bonis

41

41

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**
- 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. 2020-21
A. De Bonis

42

42

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
 - 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. 2020-21
A. De Bonis

45

45

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. 2020-21
A. De Bonis

46

46

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**
- **diz.pop(k)**
 - 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)

Programmazione Avanzata a.a. 2020-21
A. De Bonis

47

47

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}
```

Programmazione Avanzata a.a. 2020-21
A. De Bonis

48

48

Programmazione avanzata a.a. 2020-21

A. De Bonis

Introduzione a Python (II parte)

Programmazione Avanzata a.a. 2020-21
A. De Bonis

49

49

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. 2020-21
A. De Bonis

50

50

Problemi con deep copy

- Dal manuale Python
 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

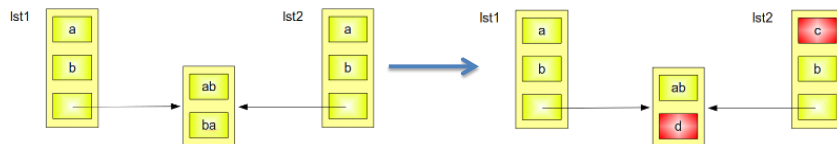
Problemi con deep copy

- Dal manuale Python
 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.

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. 2020-21
A. De Bonis

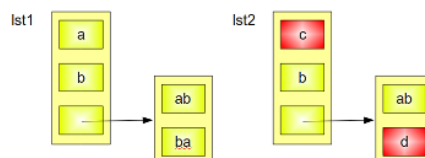
53

53

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. 2020-21
A. De Bonis

54

54

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. 2020-21
A. De Bonis

55

55

Operatori aritmetici

- Gli operatori aritmetici sono quelli a destra

```

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

```

- 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 //
 - Il risultato (**int**) è la parte intera della divisione

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

```
// adesso si chiama floor division
```

Programmazione Avanzata a.a. 2020-21
A. De Bonis

56

56

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

57

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 dello stesso tipo che **contengono** gli stessi valori

58

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

59

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)

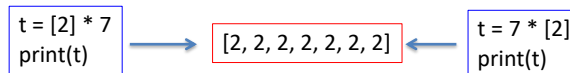
60

Operatori per sequenze

list, tuple e str

- I tipi sequenza predefiniti in Python supportano i seguenti operatori

<code>s[j]</code>	element at index <i>j</i>
<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, ..., up to but not equalling or stop
<code>s + t</code>	concatenation of sequences
<code>k * s</code>	shorthand for <code>s + s + s + ...</code> (k times)
<code>val in s</code>	containment check
<code>val not in s</code>	non-containment check



Programmazione Avanzata a.a. 2020-21
A. De Bonis

61

61

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. 2020-21
A. De Bonis

62

62

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. 2020-21
A. De Bonis

63

63

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 <= s2</code>	s1 is subset of s2
<code>s1 < s2</code>	s1 is proper subset of s2
<code>s1 >= s2</code>	s1 is superset of s2
<code>s1 > s2</code>	s1 is proper superset of s2
<code>s1 s2</code>	the union of s1 and s2
<code>s1 & 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. 2020-21
A. De Bonis

64

64

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

Programmazione Avanzata a.a. 2020-21
A. De Bonis

65

65

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><<</code> , <code>>></code>
8	bitwise-and	<code>&</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><</code> , <code><=</code> , <code>></code> , <code>>=</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.

Programmazione Avanzata a.a. 2020-21
A. De Bonis

66

66

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 e l'identificatore è riassegnato al nuovo oggetto
 - Alcuni tipi di dato (e.g., list) ridefiniscono la semantica dell'operatore +=

Programmazione Avanzata a.a. 2020-21
A. De Bonis

67

67

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')
```

Programmazione Avanzata a.a. 2020-21
A. De Bonis

68

68

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. 2020-21
A. De Bonis

69

69

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. 2020-21
A. De Bonis

70

70

Controllo del flusso in Python

Programmazione Avanzata a.a. 2020-21
A. De Bonis

71

71

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. 2020-21
A. De Bonis

72

72

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. 2020-21
A. De Bonis

73

73

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. 2020-21
A. De Bonis

74

74

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

75

while

while *condition*:
body

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

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

76

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
```

77

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
```

78

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. 2020-21
A. De Bonis

79

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. 2020-21
A. De Bonis

80

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. 2020-21
A. De Bonis

81

81

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. 2020-21
A. De Bonis

82

82

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]

83

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)
```

84

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. 2020-21
A. De Bonis

85

85

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. 2020-21
A. De Bonis

86

86

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]]
```

87

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) }

88