Programmazione avanzata
a.a. 2021-22

Introduzione a Python
(I e II lezione)

1

Informazioni utili

- Il sito Web del corso:
  - http://intranet.di.unisa.it/~debonis/progAv2021-22
  - non ancora pronto per problemi tecnici
- Il mio studio: numero 44, quarto piano, stecca 7
  - per il momento gli studenti non possono accedervi
- L'orario di ricevimento: on-line su appuntamento

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

3

Indice PYPL

Creato analizzando quanto spesso tutorial sul linguaggio sono cercati su Google

Worldwide, Sept 2021 compared to a year ago:

Rank	Change	Language	Share	Trend
1		Python	29.48 %	-2.4 %
2		Java	17.18 %	+0.7 %
3		JavaScript	9.14 %	+0.8 %
4		C#	6.94 %	+0.6 %
5		PHP	6.49 %	+0.4 %
6		C/C++	6.49 %	+0.9 %
7		R	3.59 %	-0.5 %
8	↑↑↑	TypeScript	2.18 %	+0.3 %
9		Swift	2.1 %	-0.4 %
10	↓↓	Objective-C	2.06 %	-0.6 %

4

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 %

5

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 %

6

Contenuti

- Design pattern
- Reflection e introspection
- System programming
- Generatori e coroutine
- Programmazione concorrente
  - Multithreading e multiprocessing
- Programmazione funzionale
- Network programming
- Vediamo qualche cenno su alcuni di questi argomenti

7

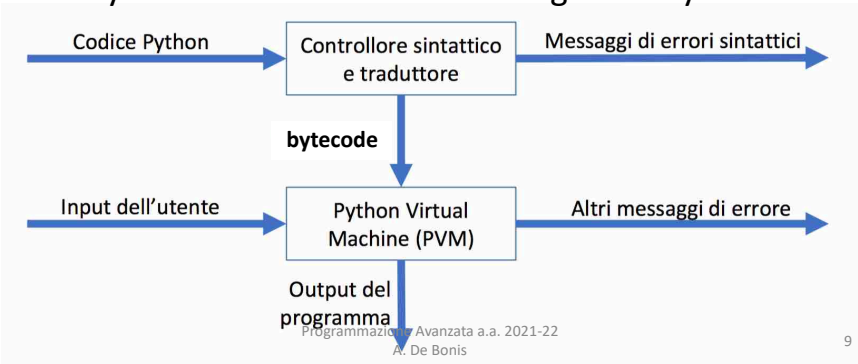
Il linguaggio Python

- 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

8

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



9

Versione Python da utilizzare

- Ultima versione Python **3.9.7**
- https://www.python.org/downloads/

- python -V oppure python --version
  - Per sapere quale versione e` installata
  - Se sono installate piu` versioni ci dice quale viene lanciata con il comando python
- Shell
- Idle, LiClipse, PyCharm
  - Ambienti di sviluppo integrati in Python

10

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

11

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

12

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))
```

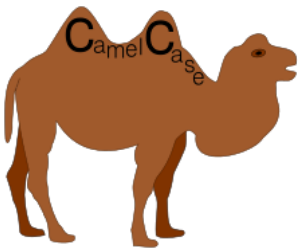

Funzione main

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

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

15

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	

16

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

17

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'>
```


18

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)

19

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

20

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
```


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

Classe **int**

```
i = int(75982347985724957923752437502354375O3)
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 immutable 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))
```

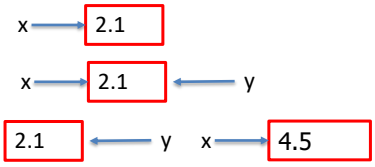
script

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

output

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

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

27

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

28

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]
```


29

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

30

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

31

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

32

Concatenazione di liste

La funzione id() fornisce l'identità di un oggetto, cioe` un intero che identifica univocamente l'oggetto per la sua intera vita. In molte implementazioni del linguaggio Python, l'identità dell'oggetto e` 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

33

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

34

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

35

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()`

36

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
```


37

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]
```


38

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']
```


39

Classe **tuple**

- Fornisce una versione immutable 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
```


40

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'>
```


41

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)
```


42

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.
```


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 (piu` 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

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

45

Classe **frozenset**

- È una classe immutable 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

46

Classe **dict**

- La classe **dict** rappresenta un dizionario
  - Un insieme di coppie (chiave, valore)
  - Le chiavi devono essere distinte e di un tipo immutable (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')].

47

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 immutable sono hashable; contenitori mutable (come liste e dizionari) non lo sono; contenitori immutable (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().

48

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'

49

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)

50

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}
```


51

Alcuni metodi classe **dict**

- diz.keys()
  - resituisce un nuovo oggetto *view* delle chiavi del dizionario
- diz.values()
  - resituisce un nuovo oggetto *view* dei valori del dizionario
- diz.items(k)
  - resituisce un nuovo oggetto *view* delle coppie (chiave, valore) del dizionario
- Gli oggetti view forniscono una ”vista” dinamica delle entrate del dizionari. 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.

52

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

Introduzione a Python (II parte)

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 oggetto 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'orginale*

55

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.

56

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.

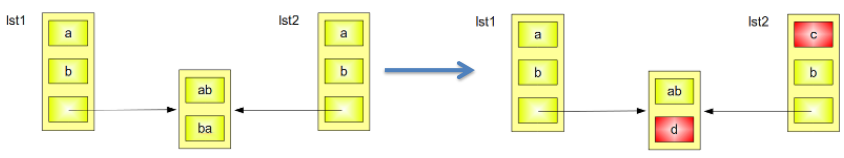
- La funzione deepcopy() evita i suddetti problemi in questo modo:
  - mantenendo un "memo" (dizionario) degli oggetti gia` copiati
  - permettendo alle classi definite-dall'utente di fare l'override delloperazione di copia.
    - per la copia deep, la classe deve definire __deepcopy__()

57

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']]
```



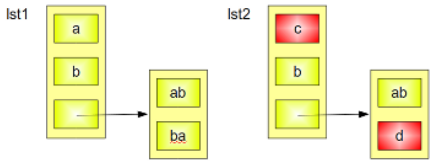
58

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']]
```



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
```

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

61

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

62

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 riferisco ad oggetti che possono essere considerati equivalenti
  - Due oggetti dello stesso tipo che *contengono* gli stessi valori

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

Operatori di confronto

- Python supporta i seguenti operatori di confronto

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

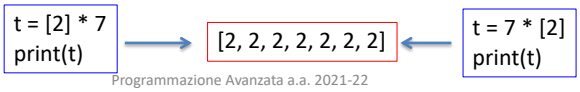
- Per gli interi hanno il significato atteso
- Per le stringhe sono case-sensitive e considerano l'ordinamento lessicografico
- Per sequenze ed insiemi assumono un significato particolare (dettagli in seguito)

65

Operatori per sequenze
list, tuple e str

- I tipi sequenza predefiniti in Python supportano i seguenti operatori

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



66

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

67

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

68

Operatori per insiemi

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

key **in** s	containment check
key **not in** s	non-containment check
s1 == s2	s1 is equivalent to s2
s1 != s2	s1 is not equivalent to s2
s1 <= s2	s1 is subset of s2
s1 < s2	s1 is proper subset of s2
s1 >= s2	s1 is superset of s2
s1 > s2	s1 is proper superset of s2
s1 | s2	the union of s1 and s2
s1 & s2	the intersection of s1 and s2
s1 − s2	the set of elements in s1 but not s2
s1 ^ s2	the set of elements in precisely one of s1 or s2

69

Operatori per dizionari

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

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

70

Precedenza degli operatori

priorità

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

71

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 +=

72

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

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

75

Controllo del flusso in Python

76

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

77

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

78

if elif … else

```
if first_condition:
    first_body
elif second_condition:
    second_body
elif third_condition:
    third_body
else:
    fourth_body
```

codice indentato

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')
```

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

79

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

80

while

while *condition*:
  body

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

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


81

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
```


82

range()

- range(n) genera una lista di interi compresi tra 0 ed n-1
  - range(start, stop, step)

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

- Utile quando vogliamo iterare in una sequenza di dati utilizzando un indice
  - **for** i **in range**(n)

```
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
```


Esempi

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

```
ciao
ciao
ciao
ciao
ciao
```

```
# 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
```

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

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

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

Crea una lista infinita

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
```


85

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
```


86

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

87

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 **]**

88

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

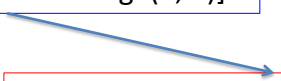
Lista dei quadrati dei numeri compresi tra 1 ed n

```
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']
```

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')]
```


91

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(trasposed)
```

```
[ [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]]
```


92

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}

93