Programmazione avanzata

a.a. 2023-24

Introduzione a Python
(I lezione)

1

Informazioni utili

- Il sito Web del corso:
  - http://intranet.di.unisa.it/~debonis/progAv2023-24
- Il mio studio: numero 44, quarto piano, stecca 7
- L'orario di ricevimento: su appuntamento anche se sara` indicato un orario preferenziale sul sito web di unisa.

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 2023 :

Rank	Change	Language	Share	1-year trend
1		Python	27.99 %	+0.1 %
2		Java	15.9 %	-1.1 %
3		JavaScript	9.36 %	-0.1 %
4		C#	6.67 %	-0.4 %
5		C/C++	6.54 %	+0.3 %
6		PHP	4.91 %	-0.4 %
7		R	4.4 %	+0.2 %
8		TypeScript	3.04 %	+0.2 %
9	↑↑	Swift	2.64 %	+0.6 %
10		Objective-C	2.15 %	+0.1 %
11	↑↑	Rust	2.12 %	+0.5 %
12	↓↓↓	Go	2.0 %	-0.1 %
13	↓	Kotlin	1.78 %	-0.0 %
14		Matlab	1.58 %	+0.1 %
15		Ruby	1.05 %	-0.1 %

5

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

6

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)**

7

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

8

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 .

9

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**

10

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.

11

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.

12

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.

13

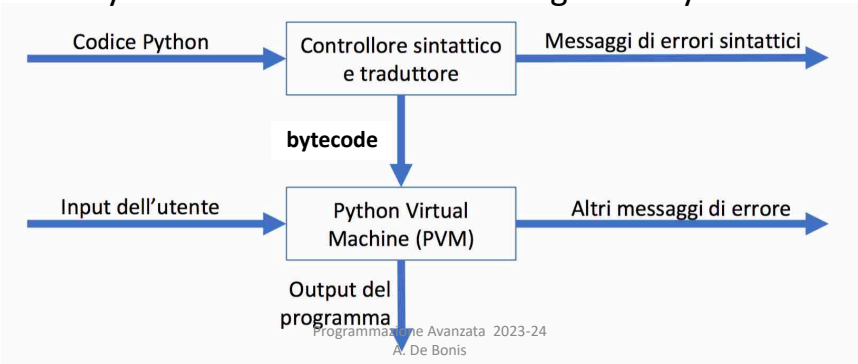
Il linguaggio Python

- 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

14

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



15

Versione Python da utilizzare

- Ultima versione Python **3.10.7**
- 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

16

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

17

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

18

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
```


19

Funzione main

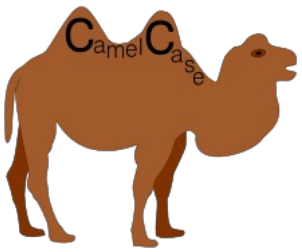
- Non è necessaria introdurla
  - Non succede come in C o Java dove la funzione main è invocata quando il programma è eseguito

20

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

21

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	

22

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

23

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'>
```


24

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)

25

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

26

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
```


27

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

28

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`

29

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)`

30

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

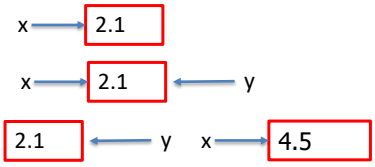
```
operatore +:  7.8
metodo __add__:  7.8
```

output

31

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

32

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

33

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

34

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]
```


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

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

37

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

38

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

39

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

40

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

41

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()`

42

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
```


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]
```

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']
```


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
```

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'>
```


47

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)
```


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.
```


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

51

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

52

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')].

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().

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'

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

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()
  - 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.

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
```