Oggetti

- Python manipola oggetti.
- Ad ogni oggetto è associato un tipo.
- Python fornisce una serie di tipi di oggetti predefiniti. I principali sono:

  numeri 1.2345

  stringhe ”Benvenuti al corso Python"

  liste [“paul", “john", “ringo”, “george”, 4]

  dizionari { "UNO" : 1, "DUE" : 2}

  tuple (”low","medium",”high")

  file myfile = fopen(”dati.txt", ‘"r")

  None indica l’oggetto “nullo”
- Gli oggetti possono essere manipolati tramite operatori (1.2 + 3.4) e funzioni (print ”Benvenuti al corso Python")

Oggetti

- Un oggetto si crea “assegnando” un valore ad una variabile che rappresenterà un riferimento all’oggetto stesso:

```
>>> a = 7 * 8
>>> b = ”una stringa”
```

- Il tipo dell’oggetto viene stabilito al momento della creazione, e dipende dal valore che gli viene assegnato:

```
>>> a = 7 * 8                    # number
>>> b = ”una stringa"             # string
```

Oggetti

- Le operazioni che possono essere fatte (ed il loro risultato) dipendono dal tipo degli oggetti coinvolti:

```
>>> a = 4
>>> b = a * 3
>>> print b
12
>>> c = "sono"
>>> d = " una stringa"
>>> e = c + d
>>> print e
sono una stringa
```

Variabili

- Una variabile rappresenta un riferimento ad un oggetto in un programma Python
- Una variabile è una sequenza arbitraria di:
  - lettere maiuscole ([A..Z])
  - lettere minuscole ([a..z])
  - cifre ([0..9])
  - underscore ( )

  in cui il primo carattere non sia una cifra

Python ha una serie di keyword riservate che non possono essere usate come identificatori (vedremo):

```
and       del       from      not     while
as        elif      global    or      with
assert    else      if        pass    yield
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
```

Variabili – dynamic typing

- Python utilizza una forma di tipizzazione dinamica:
  - solo gli oggetti hanno tipi
  - il tipo di una variabile è il tipo dell'oggetto a cui si riferisce
- se ad una variabile si assegna un nuovo oggetto, il suo tipo diventa quello del nuovo oggetto a cui si riferisce (mentre il vecchio oggetto rimane invariato)

```
>>> a = 4
>>> a = "stringa"
>>> b = a
>>> print b
stringa
>>> b = 3
>>> print b
3
>>> print a
stringa
```

Variabili – strong typing

- In ogni istante, una variabile ha il tipo dell'oggetto a cui si riferisce
- Non e` possibile eseguire operazioni sulla variabile che non siano compatibili con il suo tipo (non c'è conversione implicita di tipo)

```
>>> a = 4            # number
>>> b = "stringa" # string
>>> c = a + b
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +:
'int' and 'str'
```

Commenti

- Un commento comincia con il carattere '#'
- Tutto ciò che segue fino al termine della riga viene ignorato dall'interprete:

```
>>> a = "ciao"
>>> a = a * 2      # replica due volte a
>>> print a        # stampa a "ciaociao"
```

- I commenti su più righe vanno scritti così:

```
"""                 '''
Commento            Commento
Su                  su
piu                 piu
linee               linee
"""                 '''
```

Statement

- Uno statement semplice è un comando contenuto in una unica riga
- Esistono vari tipi di statement. Finora abbiamo visto:
  - assegnazione `x = a + b`
  - espressione `a + b` (principalmente per uso interattivo)
  - stampa `print a + b`
- Esistono anche statement composti che operano su più righe (tipicamente per uso non-interattivo, esempio):

```
if a > 0:
    b=a*2
    print b
```

Funzioni

- Una funzione è una sequenza di comandi raggruppati in modo da poter essere chiamati più volte
- Una funzione ha un nome con il quale viene chiamata
- Una funzione può avere uno o più oggetti in ingresso
- Una funzione può produrre un oggetto in uscita
- Python fornisce una serie di funzioni di utilità predefinite
- Gli oggetti (a parte i numeri) hanno funzioni proprie per manipolarli
- E’ possibile scrivere le proprie funzioni

Moduli

- Programmi complessi vengono suddivisi in componenti chiamati moduli
- Un modulo corrisponde ad un file di codice Python (estensione ".py")
- I moduli forniscono una vasta gamma di funzioni di utilitàche possono essere impiegate nei propri programmi
- Per poter utilizzare il contenuto di un modulo, deve essere importato (comando `import`):

```
>>> import math   # modulo funzioni matematiche
>>> print math.cos(math.pi)
-1.0
```

Tipi numerici

- **interi normali**
  - interi con una gamma prefissata di possibili valori (che dipende dall'architettura, [-2147483648,2147483647] con parole a 32-bit):

    1234, 0, -100

- **Interi lunghi**
  - interi con gamma illimitata (dipende dalla quantita` di memoria della macchina): e.g.420000000000000000L

- **reali in virgola mobile**
  - numeri reali, rappresentabili sia in notazione scientifica che non:

    -1.234, 15e5, -2E-3, 40.4E+22

- **booleani**
  - assumono solo due valori: True (corrispondente a 1) e False (corrispondente a 0). Si usano nelle espressioni logiche.

Operatori aritmetici binari

- + - * /
  - l'interprete esegue operazioni solo su operandi dello stesso tipo.
  - quando gli operandi sono di tipo diverso, l'interprete li converte nel tipo più complesso tra i tipi degli operandi prima di eseguire l'operazione
- // - divisione intera
  - la *divisione intera* restituisce il quoziente della divisione intera a prescindere dal tipo degli operandi numerici
  - se di diverso tipo, gli operandi vengono comunque prima convertiti nel tipo più complesso
- %
  - l'operazione di *modulo* restituisce il resto della divisione intera
- **
  - eleva il primo operando alla potenza del secondo
  - effettua automaticamente la conversione di tipo del risultato se necessario

Operatori relazionali e logici

- `< <= > >= == <> !=`
  - Confrontano due operandi e restituiscono un valore di verità (True o False)
  - Attenzione == è != da =
  - L'operatore di disuguaglianza si scrive in due modi equivalenti:

    `<>` e `!=`
- `and or not`
  - Permettono di costruire espressioni logiche

```
>>> 3 == 3 and 4 < 3
False
```

  - `False`
    - E' falso False, None, lo zero (di qualunque tipo), un oggetto vuoto (stringa, tupla, etc)
  - True
    - Tutto il resto

Variabili

- Una variabile viene creata nel momento in cui le si assegna un valore.
- Non è possibile utilizzare una variabile prima che venga creata.
- Una successiva istruzione di assegnazione modificherà il contenuto di una variabile con il risultato dell'espressione a destra dell'assegnazione.

Stringhe

- Una stringa è delimitata da apici *singoli* ‘ o *doppi* " "
- I due tipi di rappresentazione sono utili per scrivere stringhe contenenti l’altro tipo di apice.
- Le stringhe possono contenere alcuni caratteri speciali di controllo preceduti dal carattere \ (*sequenza di escape)*.
- Le sequenze di escape si usano anche per poter scrivere caratteri come apici, e la ’\’ stessa.
- Le più comuni sequenze di escape sono:

  \\ backslash (scrive \)
  \’ apice singolo (scrive ’)
  \” apice doppio (scrive ”)
  \n ritorno a capo
  \t tab orizzontale

- Modalità raw: stringa preceduta da r

Stringhe

- E’ possibile scrivere stringhe su più righe, delimitandole con virgolette triple (usando sia singoli che doppi apici)
- Non è necessario proteggere i caratteri di fine riga con escape quando si usano le triple virgolette, questi verranno inclusi nella stringa, ad esempio:

```
>>> print """
>>> Uso: comando [OPZIONI]
>>> -h Visualizza questo messaggio
>>> -H hostname
"""
<<<
```

produrrà il seguente output:

Uso: comando [OPZIONI]

-h Visualizza questo messaggio

-H hostname

Stringhe

- Le stringhe possono essere concatenate tramite + e ripetute con * (operatore overloaded):

```
>>> parola = 'Aiuto' + 'A'
>>> parola
'AiutoA'
>>> '<' + parola*5 + '>'
'<AiutoAAiutoAAiutoAAiutoAAiutoA>'
```

Stringhe

- In Python una stringa è una **sequenza** di caratteri.
- Questa caratteristica fa si che le stringhe supportino una serie di operazioni su sequenza (es. `len`)
- Le stringhe possono essere indicizzate come in C.
- Il primo carattere di una stringa ha indice 0.
- Gli indici possono essere numeri negativi, per iniziare il conteggio da destra. Ad esempio:

```
>>> parola = 'Aiuto' + 'A'
>>> parola[-1]     # L'ultimo carattere 'A'
>>> parola[-2]     # Il penultimo carattere 'o'
```

- L'operatore di confronto `in` restituisce `True` se un certo carattere (o sottostringa) si trova in una stringa, `False` altrimenti:

```
>>> s = "abcd"
>>> "a" in s
True
```

Stringhe

- Possono essere specificate sottostringhe con la notazione a 'slice': due indici separati dal carattere due punti.

```
>>> parola = 'Aiuto' + 'A'
>>> parola[4]
'o'
>>> parola[0:2]
'Ai'
>>> parola[2:4]
'ut'
```

- Il primo indice, se omesso, viene impostato al valore predefinito 0. Se viene tralasciato il secondo, viene impostato alla dimensione della stringa.

```
>>> parola = 'Aiuto' + 'A'
>>> parola[:2] # I primi due caratteri
'Ai'
>>> parola[2:] # Tutti eccetto i primi due caratteri
'utoA
```

Stringhe

- Le stringhe sono un tipo di oggetti **immutabili.**
- Non è possibile modificare una stringa una volta creata.

```
>>> Saluto = "Ciao!“
>>> Saluto[0] = 'M'      # ERRORE!
>>>print Saluto
```

- Invece di ottenere Miao! questo codice produce un errore perché le stringhe sono immutabili. L'unica cosa che si può fare è creare una nuova stringa come variante di quella originale:

```
>>>Saluto = "Ciao!“
>>>NuovoSaluto = 'M' + Saluto[1:]
>>>print NuovoSaluto
'Miao!'
```

- Abbiamo concatenato la nuova prima lettera ad una porzione di Saluto, e questa operazione non ha avuto alcun effetto sulla stringa originale.

Stringhe

- Come già detto, non è possibile eseguire operazioni tra tipi non compatibili (e.g. sommare una stringa e un numero)
- E' possibile però convertire esplicitamente un oggetto di un tipo in uno di un altro tipo (dove tale conversione è definita)

```
>>> "x=" + str(10) # str da numeri a stringhe
'x=10'
>>> int("2") + 6 # int da stringhe a interi
8
>>> float("4.3") / 1.5 # float da stringhe a reali
2.866666666666667
```

Stringhe

- L'operatore `%` serve a formattare una stringa a partire da una tupla di argomenti.

```
>>> "x=%s,y=%s,z=%s" % (5,10,"basso")
'x=5,y=10,z=basso'
```

- La stringa a sinistra contiene degli specificatori di conversione (%s), la tupla a destra i valori da sostituire
- %s indica una stringa, in pratica utilizza la funzione str per convertire l'elemento corrispondente della tupla in una stringa.

- Numerose sono le funzioni disponibili per le stringhe → help(str).

Liste

- Una lista è una **sequenza** di oggetti qualunque ed è disponibile come tipo predefinito:

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
```

- La lista è un tipo **mutabile**

```
>>> a[0] = 1000
```

- Come per le stringhe, la lunghezza di una lista si ottiene:

```
>>> len(a)
4
```

Liste

- Operazioni

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a[0]
'spam'
>>> a[-2]
100
>> a + ['bacon', 2*2]
['spam', 'eggs', 100, 1234, 'bacon', 4]
>>> a * 2
['spam', 'eggs', 100, 1234, 'spam', 'eggs', 100, 1234]
>> 'eggs' in a
True
```

Liste

- Indicizzazione

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a[1:-1]
['eggs', 100]
>>> a[:2] + ['bacon', 2*2]
['spam', 'eggs', 'bacon', 4]
>>> 3*a[:3] + ['Boe!']
['spam', 'eggs', 100, 'spam', 'eggs', 100,
'spam', 'eggs', 100, 'Boe!‘]
```

Lista

```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a[0:2] = [1, 12]                    #Rimpiazza alcuni elementi
>>> a
[1, 12, 100, 1234]
>>> a[0:2] = []                         #Rimuove alcuni elementi
>>> a
[100, 1234]
>>>a[1:1] = ['bletch', 'xyzzy’]        #Inserisce alcuni elementi
>>> a
[100, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a          #Inserisce (una copia di) se stesso all'inizio
>>> a
[100, 'bletch', 'xyzzy', 1234, 100, 'bletch', ‘xyzzy’, 1234]
```

Liste annidate

- Una lista i cui elementi siano tutte liste implementa una *matrice.* In generale, si parla di liste di liste o liste annidate.

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> matrix[2]
[7, 8, 9]
>>> matrix[2][0]
7
>>> matrix[2][0:2]
[7, 8]
```

Liste annidate

- Esempi

```
>>> q = [2, 3]
>>> p = [1, q, 4]
>>> p
[1, [2, 3], 4]
>>> len(p)
3
>>> p[1]
[2, 3]
>>> p[1][0]
2
```

Liste annidate

- Esempi

```
>>> p[1].append('xtra')
>>> p
[1, [2, 3, 'xtra'], 4]
>>> q
[2, 3, 'xtra']
```

Metodi sulle liste

```
>>> a = [66.6, 333, 333, 1, 1234.5]
>>> print a.count(333), a.count(66.6), a.count('x')
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.6, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.6, -1, 333, 1, 1234.5, 333]
```

Metodi sulle liste

```
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.6]
>>> a.sort()
>>> a
[-1, 1, 66.6, 333, 333, 1234.5]
```

Implementazione del tipo pila

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Implementazione del tipo coda

```
>>> queue = ["Eric", "John", "Michael"]
>>> queue.append("Terry") # Aggiunge Terry
>>> queue.append("Graham") # Aggiunge Graham
>>> queue
['Eric', 'John', 'Michael', 'Terry', 'Graham']
>>> queue.pop(0)
'Eric'
>>> queue.pop(0)
'John'
>>> queue
['Michael', 'Terry', 'Graham']
```

List comprehensions

- La *list comprehension* è un tipo di espressione Python che permette di creare una lista come risultato di un'operazione sugli elementi di un'altra lista

```
>>> L = [0,1,2,3,4]
>>> [i+1 for i in L]
[1, 2, 3, 4, 5]
```

- Il costrutto è composto da:
  - Un oggetto di tipo sequenza su cui iterare.
  - Una variabile (o più) che raccolga di volta in volta gli elementi
  - Una espressione che faccia qualche operazione che coinvolga l'elemento cui la variabile si riferisce.
  - Le parole riservate `for` ed `in`
  - Le parentesi quadre a delimitazione

List comprehensions

- Esempi.
- Estrazione di una colonna da una matrice:

```
>>> matrix = [[1,2,3],[4,5,6],[7,8,9]]
>>> [ row[2] for row in matrix]
[3, 6, 9]
```

- Invocazione di una funzione:

```
>>> [ str.upper() for str in ["a","b","c"]]
['A', 'B', 'C']
```

- Selezione elementi in accoppiata con la parola chiave `if`:

```
>>> [i for i in [1,3,4,5,6,2,8,9] if i % 2 != 0]
[1, 3, 5, 9]
```

List comprehensions

- Il costrutto può essere usato con qualunque oggetto di tipo *sequenza*:

```
>>> [ r for r in "AHHDCCEDGGTA" if r in "HC"]
['H', 'H', 'C', 'C']
```

- Può essere combinato con metodi o funzioni che processano liste:

```
>>> "".join([ r for r in "AHHDCCEDGGTA" if r in "HC"])
'HHCC'
```

Tuple

- Si è visto come stringhe e liste abbiano molte proprietà in comune, p.e. le operazioni di indicizzazione e slice.
- Si tratta di due esempi di tipi di dato del genere sequenza. Esiste anche un altro tipo di dato standard del genere sequenza: la tupla.
- Una tupla è composta da un certo numero di oggetti separati da virgole e racchiusi da parantesi tonde.
- Esempio:

```
>>> t = (12345, 54321, 'hello!')
>>> t[0]
12345
>>> u = (t, (1, 2, 3, 4, 5)) #Le tuple possono essere annidate
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```

Tuple

- Le tuple hanno molti usi, per esempio coppie di coordinate (x,y), record di un database ecc.
- Le tuple, come le stringhe, sono immutabili.

```
>>> t[2] = 3
Traceback (most recent call last):
File "<console>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

- È però possibile creare tuple che contengano oggetti mutabili, come liste:

```
>>> t = (100, "ciccio", [1,2,3,4])
(100, 'ciccio', [1, 2, 3, 4])
>>> t[2][0] = 1000
>>> t
(100, 'ciccio', [1000, 2, 3, 4])
```

- Se si desidera modificare i contenuti di una tupla, è possibile:
  - create una lista che abbia come elementi gli elementi della tupla (funzione list(t))
  - fare le necessarie modifiche sulla lista
  - creare una tupla che abbia come elementi gli elementi della lista (funzione tuple(l))
  - eventualmente assegnare la nuova tupla alla variabile che si riferiva alla tupla originaria

Tuple

- L'istruzione:

```
t = 12345, 54321, 'hello!'
```

è un esempio di impacchettamento (packing) in tupla: i valori 12345, 54321 e 'hello!' sono impacchettati in una tupla.

- È anche possibile l'operazione inversa, ad esempio:

```
>>> x, y, z = t
```

- È chiamata “unpacking” di sequenza. Lo spacchettamento di sequenza richiede che la lista di variabili a sinistra abbia un numero di elementi pari alla lunghezza della sequenza.
- L'impacchettamento di valori multipli crea sempre una tupla, mentre lo spacchettamento funziona per qualsiasi sequenza.

Tuple

- Le tuple sono spesso usate per semplificare le operazioni di assegnazione a più variabili

```
>>> (x, y, z) = (1.1, 2, 7.8)
>>> y
2
```

- In alternativa:

```
>>> x, y, z = 1.1, 2, 7.8
>>> y, z
(2, 7.8)
```

Numeri complessi

- Python supporta i numeri complessi.
- Per contrassegnare i numeri immaginari si usa il suffisso "j" o "J".
- I numeri complessi sono creati con "(real+imagj)", o con la funzione complex(real, imag)

```
>>> 1j * 1J
(-1+0j)
>>> 1j * complex(0,1)
(-1+0j)
>>> (3+1j)*3
(9+3j)
>>> (1+2j)/(1+1j)
(1.5+0.5j)
```

Numeri complessi

- I numeri complessi vengono rappresentati come due float, la parte reale e quella immaginaria. Per estrarre queste parti si usano z.real e z.imag. .

```
>>> a=3.0+4.0j
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a)          # sqrt(a.real**2 + a.imag**2)
5.0
```

Dizionari

- Un **dizionario** è un insieme non ordinato di coppie chiave valore, con il requisito che ogni chiave deve essere unica all'interno di un dizionario (tabella hash).
- Una coppia di parentesi graffe crea un dizionario vuoto: {}.
- Mettendo tra parentesi graffe una lista di coppie chiave: valore separate da virgole si ottengono le coppie iniziali del dizionario.

```
>>> tel = {'jack': 4098, 'sape': 4139}
```

- Essendo una collezione non ordinata, gli oggetti non manterranno l'ordine con cui sono stati inseriti, ma verranno riarrangiati in maniera casuale per velocizzare le operazioni di ricerca e recupero.

Dizionari

- Le chiavi di un dizionario non devono necessariamente essere stringhe, basta che siano oggetti immutabili. Quindi, stringhe e numeri possono essere usati come chiavi in ogni caso, le tuple possono esserlo se contengono solo stringhe, numeri o tuple; se una tupla contiene un qualsivoglia oggetto mutabile, sia direttamente che indirettamente, non può essere usata come chiave. Non si possono usare come chiavi le liste, dato che possono essere modificate.
- Le operazioni principali su un dizionario sono la memorizzazione di un valore con una qualche chiave e l'estrazione del valore corrispondente a una data chiave.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
```

Dizionari

- I Dizionari sono mutabili. Se si memorizza un valore usando una chiave già in uso, il vecchio valore associato alla chiave viene sovrascritto.

```
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel[‘jack’] = 3027
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 3027}
```

- È anche possibile cancellare una coppia chiave-valore con del.

```
>>> del tel['sape']
>>> tel
{'guido': 4127, 'jack': 3027}
```

Dizionari

- Verifica della presenza di una certa chiave (operatore `in` o `metodo has_key`)

```
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 2037}
>>> ‘guido’ in tel
True
>>> ‘andrea’ in tel
False
>>> tel.has_key(‘guido’)
True
```

- Il metodo `get` consente di recuperare un oggetto specificando chiave e valore di default da restituire se non presente (`None` se non specificato)

```
>>> tel.get(‘guido’)
4127
>>> tel.get(‘andrea’, -1)
-1
```

Dizionari

- Alcuni metodi.
  - `len` - restituisce il numero di elementi (coppie chiave, valore) contenuti nel dizionario:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> len(tel)
2
```

  - `keys` - restituisce la lista delle chiavi contenute nel dizionario

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel.keys()
['jack', 'sape']
```

  - `values` - restituisce la lista dei valori contenuti nel dizionario

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel.valus()
[4098, 4139]
```

  - `items` - restituisce la lista delle coppie chiave,valore (come tuple) contenute nel dizionario

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel.items()
[('jack', 4098), ('sape', 4139)]
```

Dizionari

- Le operazioni che presuppongono un ordinamento degli elementi non funzionano con i dizionari.
  - concatenazione o ripetizione

```
>>> d1 = {"a" : 0, "b" : 1, "c" : 2}
>>> d2 = {"d" : 3, "e" : 4}
>>> d1 + d2
Traceback (most recent call last):
        File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

  - Estrazione o modifica di sottodizionari

```
>>> d1["a":"b"] = {}
Traceback (most recent call last):
       File "<stdin>", line 1, in <module>
TypeError: unhashable type
```

Insiemi

- Python include anche tipi di dati per insiemi (sets).
- Un insieme è una collezione non ordinata che non contiene elementi duplicati al suo interno.
- Solitamente viene usato per verificare l'appartenenza dei membri ed eliminare gli elementi duplicati.
- Gli oggetti insieme supportano anche le operazioni matematiche come l'unione, l'intersezione, la differenza e la differenza simmetrica.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', ‘orange', 'banana’]
>>> fruits = set(basket) # crea un insieme con frutti
>>> fruits
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruits
True
>>> ’strawberry' in fruits
False
```

Insiemi

```
>>> a = set('abracadabra’)
>>> a
set(['a', 'r', 'b', 'c', 'd'])
>>> b = set('alacazam')
>>> b
set(['a', 'c', 'z', 'm', 'l'])
>>> a - b                    # lettera in a ma non in b
set(['r', 'd', 'b'])
>>> a | b                    # lettera in a o in b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                    # lettera comune in a ed in b
set(['a', 'c'])
>>> a ^ b              # lettera in a o b ma non in comune
set(['r', 'd', 'b', 'm', 'z', 'l'])
```