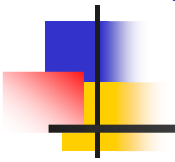
Linguaggi

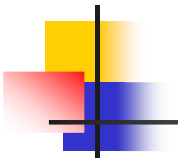
Corso M-Z - Laurea in Ingegneria Informatica
A.A. 2007-2008

Alessandro Longheu
http://www.diit.unict.it/users/alongheu
alessandro.longheu@diit.unict.it



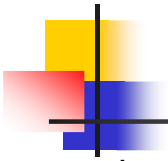
- lezione 12 -

Il linguaggio Python



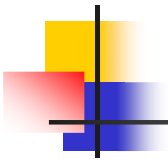
Python

- Python è stato sviluppato più di dieci anni fa da Guido van Rossum che ne ha derivato semplicità di sintassi e facilità d'uso in gran parte da ABC, un linguaggio dedicato all'insegnamento sviluppato negli anni '80.
- Oltre che per questo specifico contesto, Python è stato creato per risolvere problemi reali, dimostrando di possedere un'ampia varietà di caratteristiche tipiche di linguaggi di programmazione quali C++, Java, Modula-3 e Scheme
- "Perché Python?" : Python permette un ottimo equilibrio tra l'aspetto pratico e quello concettuale
  - Python è interpretato
  - Python è fornito di un'ampia libreria di moduli
  - Python offre caratteristiche dei linguaggi object oriented, ma anche dei linguaggi di funzionali



Python - interprete

- Python è un linguaggio interpretato; Ci sono due modi di usare l'interprete: a linea di comando o in modo script (estensione .py)
- Le funzioni di editing di riga dell'interprete di solito non sono molto sofisticate. L'interprete opera all'incirca come una shell Unix: quando viene lanciato con lo standard input connesso ad un terminale legge ed esegue interattivamente dei comandi; quando viene invocato con il nome di un file come argomento o con un file come standard input legge ed esegue uno *script* da quel file.
- Quando noti all'interprete, il nome dello script e gli argomenti addizionali sono passati allo script tramite la variabile sys.argv, che è una lista di stringhe. La sua lunghezza minima è uno; quando non vengono forniti né script né argomenti, sys.argv[0] è una stringa vuota. Quando il nome dello script è fornito come '-' (che identifica lo standard input), sys.argv[0] viene impostato a '-'. Allorché viene usato **-c** *comando*, sys.argv[0] viene impostato a -c. Le opzioni trovate dopo **-c** *comando* non vengono consumate nell'elaborazione delle opzioni da parte dell'interprete Python, ma lasciate in sys.argv e gestite dal comando.

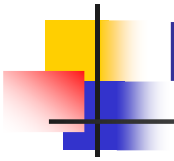


Python - interprete

- Le righe di continuazione sono necessarie quando si introduce un costrutto multiriga, ad esempio:

```
>>> il_mondo_è_piatto = 1
>>> if il_mondo_è_piatto:
... print "Occhio a non caderne fuori!"
>>> Occhio a non caderne fuori!
```

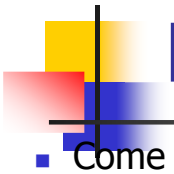
- Quando sopravviene un errore, l'interprete stampa un messaggio di errore e una traccia dello stack. Se si trova in modalità interattiva ritorna poi al prompt primario; se l'input proveniva da un file, esce con uno stato diverso da zero dopo aver stampato la traccia dello stack. Alcuni errori sono incondizionatamente fatali e provocano un'uscita con uno stato diverso da zero; ciò accade quando si tratta di problemi interni dell'interprete e di alcuni casi di esaurimento della memoria. Tutti i messaggi di errore vengono scritti nel flusso dello standard error; l'output normale dei comandi eseguiti viene scritto sullo standard output.



Python – tipi di dato numerici

- L'interprete si può comportare come una semplice calcolatrice: si può digitare un'espressione ed esso fornirà il valore risultante. La sintassi delle espressioni è chiara: gli operatori +, -, * e / funzionano come nella maggior parte degli altri linguaggi (p.e. Pascal o C); le parentesi possono essere usate per raggruppare operatori e operandi. Ad esempio:

```
>>> 2+2
4
>>> # Questo è un commento ...
2+2
4
>>> 2+2 # e un commento sulla stessa riga del codice
4
>>> (50-5*6)/4
5
>>> # Una divisione tra interi restituisce solo il quoziente:
... 7/3
2
>>> 7/-3
-3
```

Python – tipi di dato numerici

- Come in C, il segno di uguale ("=") è usato per assegnare un valore ad una variabile. Il valore di un assegnamento non viene stampato:

```
>>> larghezza = 20
>>> altezza = 5*9
>>> larghezza * altezza
900
```

- Un valore può essere assegnato simultaneamente a variabili diverse:

```
>>> x = y = z = 0 # Zero x, y e z
>>> x
0
>>> y
0
```

- Le operazioni in virgola mobile sono pienamente supportate; in presenza di operandi di tipo misto gli interi vengono convertiti in virgola mobile:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

Python – tipi di dato numerici

- I numeri complessi vengono supportati; per contrassegnare i numeri immaginari si usa il suffisso "j" o "J". I numeri complessi con una componente reale non nulla vengono indicati come "(real+imagj)", o possono essere creati 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)
```

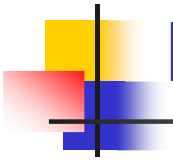
- I numeri complessi vengono sempre rappresentati come due numeri in virgola mobile, la parte reale e quella immaginaria. Per estrarre queste parti da un numero complesso z si usano z.real e z.imag.

```
>>> a=1.5+0.5j
>>> a.real
1.5
>>> a.imag
0.5
```


Python – tipi di dato numerici

- Le funzioni di conversione in virgola mobile e intero (float(), int() e long()) non funzionano con i numeri complessi: non c'è alcun modo corretto per convertire un numero complesso in numero reale. Usare abs(*z*) per ottenere la sua grandezza (in virgola mobila) o z.real per ottenere la sua parte reale.

```
>>> a=3.0+4.0j
>>> float(a)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: can't convert complex to float; use abs(z)
>>> a.real
3.0
>>> a.imag
4.0
>>> abs(a) # sqrt(a.real**2 + a.imag**2)
5.0
```

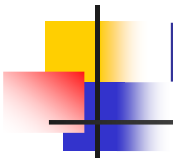


Python – tipi di dato numerici

- In modo interattivo, l'ultima espressione stampata è assegnata alla variabile _. Questo facilita i calcoli in successione quando si sta usando Python come calcolatrice, ad esempio:

```
>>> tassa = 12.5 / 100
>>> prezzo = 100.50
>>> prezzo * tassa
12.5625
>>> prezzo + _
113.0625
>>> round(_, 2)
113.06
```

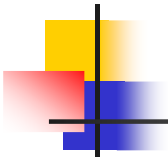
- Questa variabile dev'essere trattata dall'utente come di sola lettura. Non le si deve assegnare esplicitamente un valore, si creerebbe una variabile locale indipendente con lo stesso nome che maschererebbe la variabile built-in ed il suo comportamento particolare.



Python – Stringhe

- Oltre ai numeri, Python può anche manipolare stringhe, che possono essere espresse in diversi modi. Possono essere racchiuse tra apici singoli o virgolette:

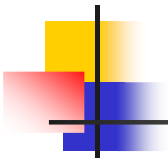
```
>>> 'spam eggs'
'spam eggs'
>>> 'doesn|'t'
"doesn't"
>>> "doesn't"
"doesn't"
>>> '"Yes," he said.'
'"Yes," he said.'
>>> "|"Yes,|" he said."
'"Yes," he said.'
 >>> '"Isn|'t," she said.'
 '"Isn|'t," she said.'
```



Python – Stringhe

- \ e' il carattere di prosecuzione riga, \n quello escape di nuova linea; se si volessero ignorare gli escape, occorre premettere r (raw) alle prime virgolette r"…":

```
>>> ciao = "Questa è una stringa lunga che contiene|n|
>>>parecchie righe di testo proprio come si farebbe in C.|n|
>>>      Si noti che gli spazi bianchi all'inizio della riga sono|
  significativi."
>>>print ciao
Questa è una stringa lunga che contiene
parecchie righe di testo proprio come si farebbe in C.
    Si noti che gli spazi bianchi all'inizio della riga sono significativi.
```

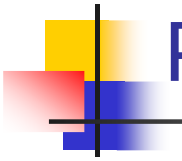
Python – Stringhe

- Le stringhe possono essere circondate da un paio di virgolette o apici tripli corrispondenti: """ o '''. Non è necessario proteggere i caratteri di fine riga quando si usano le triple virgolette, questi verranno inclusi nella stringa, ad esempio:

```
>>>print """
>>>Uso: comando [OPZIONI]
>>>-h Visualizza questo messaggio
>>>-H hostname Hostname per connettersi a
>>>"""
```

- produrrà il seguente output:

```
Uso: comando [OPZIONI]
-h Visualizza questo messaggio
-H hostname Hostname per connettersi a
```



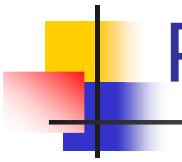
Python – Stringhe

- Le stringhe possono essere concatenate (incollate assieme) tramite l'operatore + e ripetute tramite *:

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

- Le stringhe possono essere indicizzate come in C, il primo carattere di una stringa ha indice 0. Non c'è alcun tipo associato al carattere di separazione; un carattere è semplicemente una stringa di lunghezza uno.
- Gli indici possono essere numeri negativi, per iniziare il conteggio da destra. Ad esempio:

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

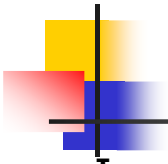
Python – Stringhe

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

```
>>> parola[4]
 'o'
>>> parola[0:2]
'Ai'
>>> parola[2:4] 'ut'
```

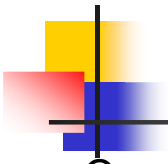
- Il primo indice, se omesso, viene impostato al valore predefinito 0. Se viene tralasciato il secondo, viene impostato alla dimensione della stringa affettata.

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



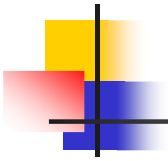
Python – Stringhe

- Immutabilità stringhe:
  >>>Saluto = "Ciao!"
  >>>Saluto[0] = 'M' # ERRORE!
  >>>print Saluto
- Invece di ottenere Miao! questo codice stampa il messaggio d'errore TypeError: object doesn't support item assignment.
- Le stringhe sono infatti 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
- Abbiamo concatenato la nuova prima lettera ad una porzione di Saluto, e questa operazione non ha avuto alcun effetto sulla stringa originale.



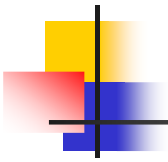
Python – Stringhe

- Questi sono metodi stringa che supportano sia stringhe a 8-bit che oggetti stringa Unicode:
- capitalize()
  Restituisce una copia della stringa con il solo carattere iniziale maiuscolo.
- center(width[, fillchar])
  Restituisce la centratura di una stringa, in un campo di ampiezza width. Il riempimento viene fatto usando il fillchar specificato (il predefinito è uno spazio). Modificato nella versione 2.4: Supporto per l'argomento fillchar.
- count(sub[, start[, end]])
  Restituisce il numero di occorrenze della sotto stringa sub nella stringa S[start:end]. Gli argomenti facoltativi start e end vengono interpretati come nella notazione delle fette.



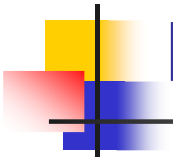
Python – Stringhe

- endswith(suffix[, start[, end]])
  Restituisce True se la stringa finisce con lo specificato suffisso suffix, altrimenti restituisce False. Con il facoltativo start, il test inizia da quella posizione. Con il facoltativo end, blocca il confronto a quella posizione.
- expandtabs([tabsize])
  Restituisce una copia della stringa dove tutti i caratteri tab vengono espansi usando gli spazi. Se tabsize non viene fornito, si assume che sia di 8 caratteri.
- find(sub[, start[, end]])
  Restituisce il più basso indice nella stringa dove viene trovata la sotto stringa sub, così che sub venga contenuta nell'intervallo [start, end]. Gli argomenti facoltativi start e end vengono interpretati come nella notazione relativa alle fette. Restituisce -1 se sub non viene trovata.



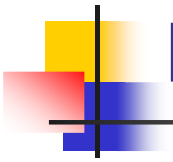
Python – Stringhe

- ndex(sub[, start[, end]])
  Come find(), ma solleva un'eccezione ValueError quando la sotto stringa non viene trovata.
- isalnum()
  Restituisce vero se tutti i caratteri nella stringa sono alfanumerici ed è presente almeno un carattere, falso negli altri casi.
- isalpha()
  Restituisce vero se tutti i caratteri nella stringa sono alfabetici ed è presente almeno un carattere, falso negli altri casi.
- isdigit()
  Restituisce vero se tutti i caratteri nella stringa sono cifre ed è presente almeno un carattere, falso negli altri casi.



Python – Stringhe

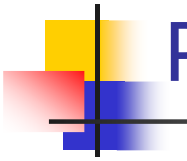
- islower()
  Restituisce vero se tutti i caratteri nella stringa sono minuscoli ed è presente almeno un carattere, falso negli altri casi.
- isspace()
  Restituisce vero se ci sono solo spazi nella stringa ed è presente almeno un carattere, falso negli altri casi.
- isupper()
  Restituisce vero se tutti i caratteri nella stringa sono maiuscoli ed è presente almeno un carattere maiuscolo, falso negli altri casi.
- join(seq)
  Restituisce una stringa che è la concatenazione delle stringhe nella sequenza seq.



Python – Liste

- Esempi di costruzione di variabili lista:

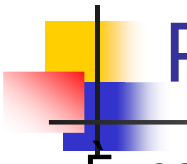
```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
 >>> a[0]
 'spam'
 >>> a[-2]
 100
 >>> a[1:-1]
 ['eggs', 100]
 >>> a[:2] + ['bacon', 2*2]
 ['spam', 'eggs', 'bacon', 4]
 >>> 3*a[:3] + ['Boe!']
 ['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs',
    100, 'Boe!']
```



Python – Liste

- Esempi di costruzione di variabili lista:

```
>>> # Rimpiazza alcuni elementi:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Rimuove alcuni elementi:
 ... a[0:2] = []
>>> a [123, 1234]
>>> # Inserisce alcuni elementi:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a # Inserisce (una copia di) se stesso all'inizio
>>> a
[123, 'bletch', 'xyzzy', 1234, 123, 'bletch', 'xyzzy', 1234]
```

Python – Liste

- È possibile avere delle liste annidate (contenenti cioè altre liste), ad esempio:

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

Python – Liste

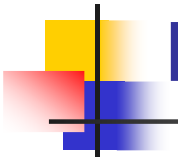
- Un esempio che utilizza buona parte dei metodi delle liste:

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


Python – Liste

- Implementazione del tipo pila:

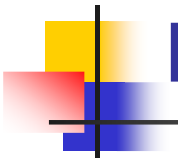
```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```



Python – Liste

- Implementazione del tipo coda:

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

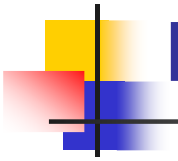
Python – Liste

- Funzioni mutuate dalla programmazione funzionale:
- "**filter** (funzione, sequenza)" restituisce una sequenza (dello stesso tipo, ove possibile) composta dagli elementi della sequenza originale per i quali è vera funzione(elemento). Per esempio, per calcolare alcuni numeri primi:

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

- "**map** (funzione, sequenza)" invoca funzione(elemento) per ciascuno degli elementi della sequenza e restituisce una lista dei valori ottenuti. Per esempio, per calcolare i cubi di alcuni numeri:

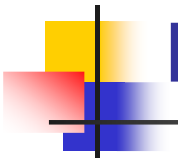
```
>>> def cube(x): return x*x*x
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```



Python – Liste

- Funzioni mutuate dalla programmazione funzionale:
- "**reduce**(*funzione*, *sequenza*)" restituisce un singolo valore ottenuto invocando la funzione a due argomenti sui primi due elementi della *sequenza*, quindi sul risultato dell'operazione e sull'elemento successivo, e così via. Ad esempio, per calcolare la somma dei numeri da 1 a 10:

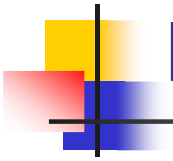
```
>>> def add(x,y): return x+y
>>> reduce(add, range(1, 11))
55
```

Python – Tuple

- Si è visto come stringhe e liste abbiano molte proprietà in comune, p.e. le operazioni di indicizzazione e affettamento. 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 valori separati da virgole, per esempio:

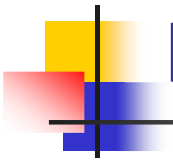
```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Le tuple possono essere annidate:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```



Python – 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. È' però possibile creare tuple che contengano oggetti mutabili, come liste.
- Un problema particolare è la costruzione di tuple contenenti 0 o 1 elementi. Le tuple vuote vengono costruite usando una coppia vuota di parentesi; una tupla con un solo elemento è costruita facendo seguire ad un singolo valore una virgola , ad esempio:

```
>>> empty = ()
>>> singleton = 'hello',
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

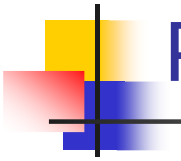



Python – Tuple

- 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
```

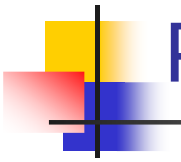
- È chiamata, **unpacking di sequenza**. Lo spacchettamento di sequenza richiede che la lista di variabili a sinistra abbia un numero di elementi pari alla lunghezza della sequenza.
- l'impacchettamento di valori multipli crea sempre una tupla, mentre lo spacchettamento funziona per qualsiasi sequenza.



Python – 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
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruits
True
>>> 'crabgrass' in fruits
False
```

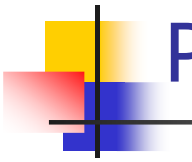



Python – Insiemi

```
>>> a = set('abracadabra')…diventa [a,r,b,c,d]
>>> b = set('alacazam') …diventa [a,l,c,z,m]
>>> a
set(['a', 'r', 'b', 'c', 'd']) (solo lettere non ripetute)
>>> 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'])
```

Python – Dizionari

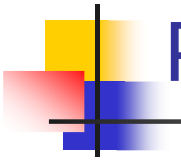
- Un **dizionario** è un insieme non ordinato di coppie chiave: valore, con il requisito che ogni chiave dev'essere unica all'interno di un dizionario (tabella hash).
- Una coppia di parentesi graffe crea un dizionario vuoto: {}. Mettendo tra parentesi graffe una lista di coppie *chiave: valore* separate da virgole si ottengono le coppie iniziali del dizionario.
- 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. E' anche possibile cancellare una coppia *chiave: valore* con del. Se si memorizza un valore usando una chiave già in uso, il vecchio valore associato alla chiave viene sovrascritto.



Python – Dizionari

- Esempio di dizionario:

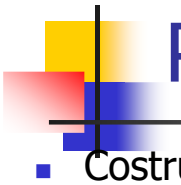
```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
```



Python – Dizionari

- l costruttore dict() crea dizionari direttamente dalla lista di coppie chiave: valore immagazzinate in tuple. Quando la coppia forma un modello, la costruzione di lista può specificare liste chiave: valore in modo compatto.

```
>>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
{'sape': 4139, 'jack': 4098, 'guido': 4127}
>>> dict([(x, x**2) for x in vec])
{2: 4, 4: 16, 6: 36}
```

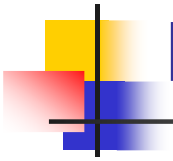



Python – Controllo di flusso

- Costrutto **IF**:

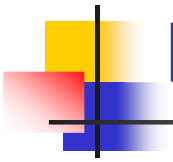
```
>>> x = int(raw_input("Introdurre un numero: "))
>>> if x < 0:
...          x = 0
...          print 'Numero negativo cambiato in zero'
... elif x == 0:
...          print 'Zero'
... elif x == 1:
...          print 'Uno'
... else:
...          print 'Più di uno' ...
```

- Possono essere presenti o meno, una o più parti elif, e la parte else è facoltativa. La parola chiave `elif' è un'abbreviazione di `else if', e serve ad evitare un eccesso di indentazioni. Una sequenza if ... elif ... elif ... sostituisce le istruzioni switch o case che si trovano in altri linguaggi.



Python – Controllo di flusso

- Le condizioni usate nelle istruzioni while e if possono contenere altri operatori oltre a quelli di confronto classici.
- Gli operatori di confronto in e not in verificano se un valore compare (o non compare) in una sequenza. Gli operatori is ed is not confrontano due oggetti per vedere se siano in realtà lo stesso oggetto; questo ha senso solo per oggetti mutabili, come le liste. Tutti gli operatori di confronto hanno la stessa priorità, che è minore di quella di tutti gli operatori matematici.
- I confronti possono essere concatenati. Per esempio, a < b == c verifica se a sia minore di b e inoltre se b eguagli c.
- Sono disponibili gli operatori booleani and, or, not; and e or sono dei cosiddetti operatori *short-circuit*: i loro argomenti vengono valutati da sinistra a destra e la valutazione si ferma non appena viene determinato il risultato. Per esempio se A e C sono veri ma B è falso, A and B and C non valuta l'espressione C.



Python – Controllo di flusso

- È possibile assegnare il risultato di un confronto o di un'altra espressione booleana a una variabile. Ad esempio:

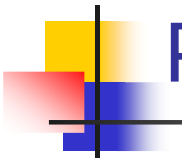
```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

- Si noti che in Python, a differenza che in C, all'interno delle espressioni non può comparire un assegnamento. I programmatori C potrebbero lamentarsi di questa scelta, però essa evita un tipo di problema che è facile incontrare nei programmi C: l'introduzione di = in un'espressione volendo invece intendere ==.

Python – Controllo di flusso

- L'istruzione **FOR** di Python differisce un po' da quella a cui si è abituati in C o Pascal. Piuttosto che iterare sempre su una progressione aritmetica (come in Pascal), o dare all'utente la possibilità di definire sia il passo iterativo che la condizione di arresto (come in C), in Python l'istruzione forcompie un'iterazione sugli elementi di una qualsiasi sequenza (p.e. una lista o una stringa), nell'ordine in cui appaiono nella sequenza. Ad esempio:

```
>>> # Misura la lunghezza di alcune stringhe:
... a = ['gatto', 'finestra', 'defenestrare']
>>> for x in a:
...         print x, len(x)
gatto 5
finestra 8
defenestrare 12
```

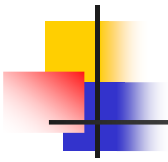
Python – Controllo di flusso

- Non è prudente modificare all'interno del ciclo la sequenza su cui avviene l'iterazione (può essere fatto solo per tipi di sequenze mutabili, come le liste). Se è necessario modificare la lista su cui si effettua l'iterazione, p.e. duplicare elementi scelti, si deve iterare su una copia. La notazione a fette rende questo procedimento particolarmente conveniente:

```
>>> for x in a[:]: # fa una copia della lista tramite slicing
...         if len(x) > 9: a.insert(0, x)
>>> a ['defenestrare', 'gatto', 'finestra', 'defenestrare']
```

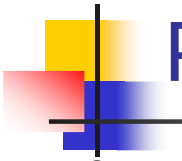
Python – Controllo di flusso

- Se è necessario iterare su una successione di numeri, viene in aiuto la funzione built-in **range**(), che genera liste contenenti progressioni aritmetiche, ad esempio:
  >>> range(10)
  [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
- L'estremo destro passato alla funzione non fa mai parte della lista generata; range(10) genera una lista di 10 valori, esattamente gli indici leciti per gli elementi di una sequenza di lunghezza 10. È possibile far partire l'intervallo da un altro numero, o specificare un incremento diverso:
  >>> range(5, 10)
  [5, 6, 7, 8, 9]
  >>> range(0, 10, 3)
  [0, 3, 6, 9]
  >>> range(-10, -100, -30)
  [-10, -40, -70]



Python – Controllo di flusso

- Per effettuare un'iterazione sugli indici di una sequenza, si possono usare in combinazione range() e len() come segue:
  >>> a = ['Mary', 'had', 'a', 'little', 'lamb']
  >>> for i in range(len(a)):
  ... print i, a[i]
  0 Mary
  1 had
  2 a
  3 little
  4 lamb



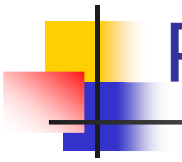
Python – Controllo di flusso

- Quando si usano i cicli sui dizionari, la chiave e il valore corrispondente possono essere richiamati contemporaneamente usando il metodo *iteritems*().

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
... print k, v
...
gallahad the pure
robin the brave
```

- Quando si usa un ciclo su una sequenza, la posizione dell'indice e il valore corrispondente possono essere richiamati contemporaneamente usando la funzione *enumerate*().

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
... print i, v
...
0 tic
1 tac
2 toe
```

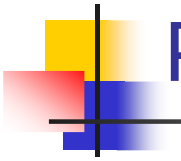
Python – Controllo di flusso

- Utilizzando un ciclo su due o più sequenze contemporaneamente, le voci possono essere accoppiate con la funzione zip().

```
>>> domande = ['nome', 'scopo', 'colore preferito']
>>> risposte = ['lancillotto', 'il santo graal', 'il blu']
>>> for q, a in zip(domande, risposte):
... print 'Qual'e` il tuo %s? E` %s.' % (q, a)
...
Qual'e` il tuo nome? E` lancillotto.
Qual'e` il tuo scopo? E` il santo graal.
Qual'e` il tuo colore preferito? E` il blu.
```

- Per eseguire un ciclo ordinato su di una sequenza, si deve usare la funzione sorted() che restituisce una nuova lista ordinata finché rimane inalterata la sorgente dei dati da elaborare.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
... print f
...
apple banana orange pear
```



Python – Controllo di flusso

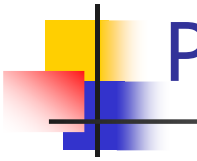
- L'istruzione **break**, come in C, esce immediatamente dal ciclo for o while più interno che la racchiude.
- L'istruzione **continue**, anch'essa presa a prestito dal C, prosegue con l'iterazione seguente del ciclo.
- L'istruzione pass non fa nulla. Può essere usata quando un'istruzione è necessaria per sintassi ma il programma non richiede venga svolta alcuna azione, ad esempio:

```
>>> while True:
...         pass # In attesa di interrupt da tastiera
```


Python – Controllo di flusso

- Le istruzioni di ciclo possono avere una **clausola else** che viene eseguita quando il ciclo termina per esaurimento della lista (con for) o quando la condizione diviene falsa (con while), ma non quando il ciclo è terminato da un'istruzione break, ad esempio:

```
>>> for n in range(2, 10):
...             for x in range(2, n):
...                     if n % x == 0:
...                     print n, 'è uguale a', x, '*', n/x
...                     break
...     else:
...                     # Il ciclo interno scorre senza trovare il fattore
...                     print n, 'è un numero primo'
2 è un numero primo
3 è un numero primo
4 è uguale a 2 * 2
5 è un numero primo
6 è uguale a 2 * 3
7 è un numero primo
8 è uguale a 2 * 4
9 è uguale a 3 * 3
```



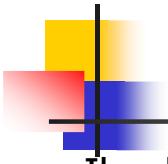
Python – Funzioni

```
>>> def fib(n): # serie di Fibonacci fino a n
... "Restituisce una lista con la serie di Fibonacci fino a n"
... result = []
... a, b = 0, 1
... while b < n:
...         result.append(b) # vedi sotto
...         a, b = b, a+b
... return result
...
 >>> f100 = fib(100) # chiama la funzione
>>> f100 # scrive il risultato
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```


Python – Funzioni

```
def ask_ok(prompt, retries=4, complaint='Si` o no,
  grazie!'):
while True:
  ok = raw_input(prompt)
  if ok in ('s', 'si', 'sì'): return True
  if ok in ('n', 'no', 'nop', 'nope'): return False
  retries = retries - 1
  if retries < 0: raise IOError, 'refusenik user'
  print complaint
```

- Questa funzione può essere chiamata così: ask_ok('Vuoi davvero uscire?') o così: ask_ok('Devo sovrascrivere il file?', 2).



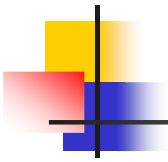
Python – Funzioni

- Il valore predefinito viene valutato una volta sola. Ciò fa sì che le cose siano molto diverse quando si tratta di un oggetto mutabile come una lista, un dizionario o istanze di più classi. A esempio, la seguente funzione accumula gli argomenti ad essa passati in chiamate successive:

```
def f(a, L=[]):
    L.append(a)
    return L
print f(1)
print f(2)
print f(3)
```

- stamperà:

```
[1]
[1, 2]
[1, 2, 3]
```

Python – Funzioni

- Le funzioni possono essere chiamate anche usando argomenti a parola chiave nella forma "parolachiave = valore". Per esempio la funzione seguente:

```
def parrot(voltage, state='a stiff', action='voom',
    type='Norwegian Blue'):
print "-- This parrot wouldn't", action,
print "if you put", voltage, "Volts through it."
print "-- Lovely plumage, the", type
print "-- It's", state, "!"
```

- Potrebbe essere chiamata in uno qualsiasi dei seguenti modi:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

Python – Funzioni

- Le chiamate seguenti non sarebbero valide:

```
parrot() # manca un argomento necessario
parrot(voltage=5.0, 'dead') # argomento non a parola chiave seguito
    da una parola chiave
parrot(110, voltage=220) # valore doppio per un argomento
parrot(actor='John Cleese') # parola chiave sconosciuta
```

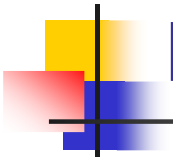
- In generale, una lista di argomenti deve avere un numero qualunque di argomenti posizionali seguiti da zero o più argomenti a parola chiave, ove le parole chiave devono essere scelte tra i nomi dei parametri formali. Nessun argomento deve ricevere un valore più di una volta -- in una medesima invocazione non possono essere usati come parole chiave nomi di parametri formali corrispondenti ad argomenti posizionali. Ecco un esempio di errore dovuto a tale restrizione:

```
>>> def function(a):
... pass
...
>>> function(0, a=0)
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```


Python – Funzioni

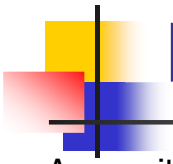
- Quando è presente un parametro formale finale nella forma **nome, esso riceve un dizionario contenente tutti gli argomenti a parola chiave la cui parola chiave non corrisponde a un parametro formale. Ciò può essere combinato con un parametro formale della forma *nome che riceve una tupla contenente gli argomenti posizionali in eccesso rispetto alla lista dei parametri formali (*nome deve trovarsi prima di **nome). Per esempio, se si definisce una funzione come la seguente:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, '?'
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments:
    print arg
    print '-'*40
    keys = keywords.keys()
    keys.sort()
    for kw in keys: print kw, ':', keywords[kw]
```



Python – Funzioni

- La funzione cheeseshop potrà venire invocata così:
  cheeseshop('Limburger', "It's very runny, sir.", "It's really very, VERY runny, sir.", shopkeeper='Michael Palin', client='John Cleese', sketch='Cheese Shop Sketch')
- Naturalmente stamperà:
  -- Do you have any Limburger ?
  -- I'm sorry, we're all out of Limburger
  It's very runny, sir.
  It's really very, VERY runny, sir.
  ----
  client : John Cleese
  shopkeeper : Michael Palin
  sketch : Cheese Shop Sketch

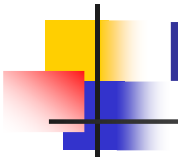


Python – Funzioni

- A seguito di numerose richieste, a Python sono state aggiunte alcune (poche) funzionalità che si trovano comunemente nei linguaggi di programmazione funzionale e in Lisp.
- Con la parola chiave lambda possono essere create piccole funzioni senza nome. Ecco una funzione che ritorna la somma dei suoi due argomenti: "lambda a, b: a+b".
- Le forme lambda possono essere usate ovunque siano richiesti oggetti funzione. Esse sono sintatticamente ristrette ad una singola espressione. Dal punto di vista semantico, sono solo un surrogato di una normale definizione di funzione:

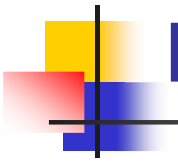
```
>>> def make_incrementor(n):
... return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(4)
46
>>>f1=make_incrementor(38)
>>>f1(4)
>>> print make_incrementor(13)(7)
20
```

- Le forme lambda permettono la creazione di funzioni personalizzate



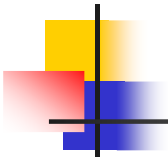
Python – Funzioni

- Altri esempi di utilizzo delle funzioni lambda:
- *>>> foo = [2, 18, 9, 22, 17, 24, 8, 12, 27]*
  >>> print filter(lambda x: x % 3 == 0, foo)
  [18, 9, 24, 12, 27]
  *>>> print map(lambda x: x * 2 + 10, foo)*
  [14, 46, 28, 54, 44, 58, 26, 34, 64]
  >>> print reduce(lambda x, y: x + y, foo)
  139
- *>>> nums = range(2, 50)*
  >>> for i in range(2, 8):
  ... nums = filter(lambda x: x == i or x % i, nums)
  ...
  >>> print nums
  [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]



Python – Funzioni

- *>>> sentence = 'It is raining cats and dogs'*
  >>> words = sentence.split()
  >>> print words
  ['It', 'is', 'raining', 'cats', 'and', 'dogs']
  >>> lengths = map(lambda word: len(word), words)
  >>> print lengths
  [2, 2, 7, 4, 3, 4]
- E' possibile concentrare il programma in un'unica riga:
- *>>> print map(lambda w: len(w), 'It is raining cats and dog s'.split())*
  [2, 2, 7, 4, 3, 4]



Python – Funzioni

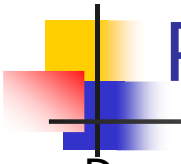
```
hostname# mount -v
/dev/sda7 on / type ext2 (rw)
/dev/md12 on /home type ext2 (rw)
```

- ```
  >>> import commands
  >>>
  >>> mount = commands.getoutput('mount -v')
  >>> lines = mount.split('\n')
  >>> points = map(lambda line: line.split()[2], lines)
  >>>
  >>> print points
  ['/', '/var', '/usr', '/usr/local', '/tmp', '/proc']
  ```
- The getoutput function from the commands module (which is part of the Python standard library) runs the given command and returns its output as a single string. Therefore, we split it up into separate lines first. Finally we use "map" with a lambda function that splits each line (on whitespace, which is the default) and returns just the third element of the result, which is the mountpoint.

Python – Moduli

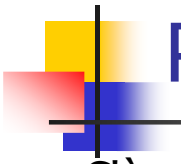
- Python permette di porre le definizioni in un file e usarle in uno script o in una sessione interattiva dell'interprete. Un file di questo tipo si chiama **modulo**; le definizioni presenti in un modulo possono essere **importate** in altri moduli o entro il modulo **main** (la collezione di variabili cui si ha accesso in uno script eseguito al livello più alto).
- Un modulo è un file che contiene definizioni e istruzioni Python. Il nome del file è il nome del modulo con il suffisso .py aggiunto. All'interno di un modulo, il nome del modulo è disponibile (sotto forma di stringa) come valore della variabile globale *__name__*. Ad esempio, si crei il file fibo.py con il seguente contenuto:

```
def fib(n): # restituisce le serie di Fibonacci fino a n
result = []
a, b = 0, 1
while b < n:
    result.append(b)
    a, b = b, a+b
return result
```
```



Python – Moduli

- Dopo avere definito il modulo, si può importare dentro l'interprete:
  >>> import fibo
- Ciò non introduce i nomi delle funzioni definite in fibo direttamente nella tabella dei simboli corrente; vi introduce solo il nome del modulo fibo. Usando il nome del modulo è possibile accedere alle funzioni:
  >>> fibo.fib(100)
  [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
  >>> fibo.__name__
  'fibo'
- Se si intende usare spesso una funzione, si può assegnare ad essa un nome locale:
  >>> fib = fibo.fib
  >>> fib(500)
  1 1 2 3 5 8 13 21 34 55 89 144 233 377

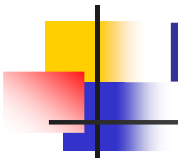


Python – Moduli

- C'è una variante dell'istruzione import che importa nomi da un modulo direttamente nella tabella dei simboli del modulo che li importa. Per esempio:
  >>> from fibo import fib
  >>> fib(500)
  1 1 2 3 5 8 13 21 34 55 89 144 233 377
- In questo modo non si introduce il nome del modulo dal quale vengono importati i nomi nella tabella dei simboli locali (così nell'esempio sopra fibo non è definito).
- C'è anche una variante per importare tutti i nomi definiti in un modulo:
  *>>> from fibo import **
  >>> fib(500)
  1 1 2 3 5 8 13 21 34 55 89 144 233 377
- In questo modo si importano tutti i nomi tranne quelli che iniziano con un `underscore' (_).

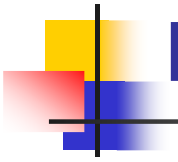
Python – Moduli

- Un modulo può contenere istruzioni eseguibili oltre che definizioni di funzione. Queste istruzioni servono ad inizializzare il modulo. Esse sono eseguite solo la **prima** volta che il modulo viene importato da qualche parte.
- Quando un modulo di nome spam viene importato, l'interprete cerca un file chiamato spam.py nella directory corrente, e quindi nella lista di directory specificata dalla variabile d'ambiente PYTHONPATH. Tale variabile ha la stessa sintassi della variabile di shell PATH, cioè una lista di nomi di directory. Quando PYTHONPATH non è configurata, o quando il file non si trova nelle directory ivi menzionate, la ricerca continua su un percorso predefinito dipendente dall'installazione; su Unix di solito si tratta di .:/usr/local/lib/python.
- Un'accelerazione rilevante dei tempi di avvio di brevi programmi che usano molti moduli standard si ottiene se nella directory dove si trova spam.py esiste un file spam.pyc, ove si assume che questo contenga una versione già ``compilata'' a livello di ``bytecode'' del modulo spam. Normalmente, non c'è bisogno di fare nulla per creare il file spam.pyc. Ogni volta che spam.py è stato compilato con successo, viene fatto un tentativo di scrivere su spam.pyc la versione compilata.



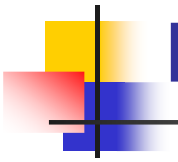
Python – Moduli

- Python viene fornito con una libreria di moduli standard. Alcuni moduli sono interni all'interprete ("built-in"). Forniscono supporto a operazioni che non fanno parte del nucleo del linguaggio ma cionondimeno sono interne, per garantire efficienza o per fornire accesso alle primitive del sistema operativo, come chiamate di sistema.
- L'insieme di tali moduli è un'opzione di configurazione che dipende dalla piattaforma sottostante. Per esempio, il modulo amoeba è fornito solo su sistemi che in qualche modo supportano le primitive Amoeba.



Python – I/O

- Spesso si desidera avere un controllo sulla formattazione dell'output che vada al di là dello stampare semplicemente dei valori separati da spazi. Esistono due modi per formattare l'output:
  - il primo è fare da sé tutto il lavoro di gestione delle stringhe, usando le operazioni di affettamento, concatenamento e riempimento (padding) del modulo standard string
  - Il secondo modo è usare l'operatore % con una stringa come argomento a sinistra. % interpreta l'argomento di sinistra come una stringa di formato nello stile della funzione C sprintf() che dev'essere applicata all'argomento a destra, e restituisce la stringa risultante.



Python – I/O

```
>>> for x in range(1,11):
...         print '%2d %3d %4d' % (x, x*x, x*x*x)
...
1   1    1
2   4    8
3   9   27
4  16   64
5  25  125
6  36  216
7  49  343
8  64  512
9  81  729
10 100 1000
```

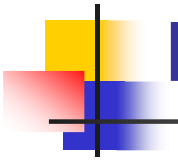
- *zfill(), che aggiunge ad una stringa numerica degli zero a sinistra. Tiene conto dei segni più e meno:*
- *>>> '12'.zfill(5) '*
- *00012'*
- *>>> '-3.14'.zfill(7)*
- *'-003.14'*

Python – I/O

- open() torna un oggetto file, ed ha la sintassi: "open(nomefile, modo)".
  >>> f=open('/tmp/workfile', 'w')
  >>> print f
  <open file '/tmp/workfile', mode 'w' at 80a0960>
- Il primo argomento è una stringa contenente il nome del file, modo è 'r' ('read') quando il file verrà solamente letto, 'w' per la sola scrittura ('write'), in caso esista già un file con lo stesso nome esso verrà cancellato, 'a' aprirà il file in aggiunta ('append'): qualsiasi dato scritto sul file verrà automaticamente aggiunto alla fine dello stesso. 'r+' aprirà il file sia in lettura che in scrittura. L'argomento modo è facoltativo; in caso di omissione verrà assunto essere 'r'.
- Su Windows e Macintosh, 'b' aggiunto al modo apre il file in modo binario, per cui esistono 'rb', 'wb' e 'r+b'. Windows distingue tra file di testo e binari; i caratteri EOF dei file di testo vengono leggermente alterati in automatico quando i dati vengono letti o scritti. Questa modifica che avviene di nascosto ai dati dei file è adatta ai file di testo ASCII, ma corromperà i dati binari presenti ad esempio in file JPEG o .EXE. Si raccomanda cautela nell'uso del modo binario quando si sta leggendo o scrivendo su questi tipi di file. Si noti che l'esatta semantica del modo testo su Macintosh dipende dalla libreria C usata.

Python – I/O

- Per leggere i contenuti di un file, s'invochi f.read(lunghezza), che legge e restituisce al più (come stringa) un numero di byte pari a lunghezza. Se lunghezza è omesso o negativo, verrà letto e restituito l'intero contenuto del file. In caso di EOF, f.read() restituirà una stringa vuota ("").
  >>> f.read()
  'Questo è l'intero file.\n'
  >>> f.read()
  ''
- f.readline() legge una singola riga dal file; un carattere di fine riga (\n) viene lasciato alla fine della stringa, e viene omesso solo nell'ultima riga del file nel caso non finisca con un fine riga. Ciò rende il valore restituito non ambiguo: se f.readline() restituisce una stringa vuota, è stata raggiunta la fine del file, mentre una riga vuota è rappresentata da '\n', stringa che contiene solo un singolo carattere di fine riga.
  >>> f.readline()
  'Questa è la prima riga del file.\n'
  >>> f.readline()
  'Seconda ed ultima riga del file\n'
  >>> f.readline()
  ''



Python – I/O

- f.readlines() restituisce una lista contenente tutte le righe di dati presenti nel file.

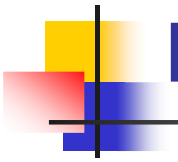
```
>>> f.readlines()
['Questa è la prima riga del file.\n', 'Seconda riga del file\n']
```

- f.write(stringa) scrive il contenuto di stringa nel file, restituendo None.

```
>>> f.write('Questo è un test\n')
```

- f.tell() restituisce un intero che fornisce la posizione nel file, misurata in byte dall'inizio del file. Per variare la posizione si usa "f.seek(offset, da_cosa)". La posizione viene calcolata aggiungendo ad offset l'argomento da_cosa. Un valore pari a 0 effettua la misura dall'inizio del file (default), 1 utilizza la posizione attuale, 2 usa la fine del file.

```
>>> f=open('/tmp/workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5) # Va al sesto byte nel file
>>> f.read(1)
'5'
>>> f.seek(-3, 2) # Va al terzo byte prima della fine del file
>>> f.read(1)
'd'
```

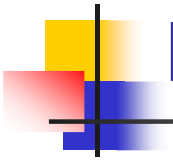



Python – I/O

- Quando si è terminato di lavorare su un file, si chiami f.close() per chiuderlo e liberare tutte le risorse di sistema occupate dal file aperto. Dopo aver invocato f.close(), i tentativi di usare l'oggetto file falliranno automaticamente.

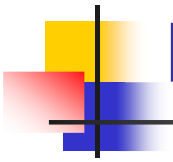
```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

- Gli oggetti file hanno alcuni metodi addizionali, disponibili nella libreria di riferimento.



Python – I/O

- Le stringhe possono essere scritte e lette da un file con facilità. I numeri richiedono uno sforzo un po' maggiore, in quanto il metodo read() restituisce solo stringhe, che dovranno essere passate a una funzione tipo int(), che prende una stringa come '123' e restituisce il corrispondente valore numerico 123. Comunque quando si desidera salvare tipi di dati più complessi, quali liste, dizionari o istanze di classe, le cose si fanno assai più complicate.
- Per non costringere gli utenti a scrivere e correggere in continuazione codice per salvare tipi di dati complessi, Python fornisce un modulo standard chiamato pickle. Si tratta di un modulo che può prendere qualsiasi oggetto Python e convertirlo in una rappresentazione sotto forma di stringa; tale processo è chiamato pickling.

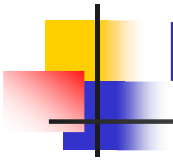


Python – I/O

- La ricostruzione dell'oggetto a partire dalla rappresentazione sotto forma di stringa è chiamata unpickling. Tra la serializzazione e la deserializzazione, la stringa che rappresenta l'oggetto può essere immagazzinata in un file, o come dato, o inviata a una macchina remota tramite una connessione di rete.
- Se si ha un oggetto x, e un oggetto file f aperto in scrittura, il modo più semplice di fare la serializzazione dell'oggetto occupa solo una riga di codice:

  pickle.dump(x, f)
- Per fare la deserializzazione dell'oggetto, se f è un oggetto file aperto in scrittura:

  x = pickle.load(f)
- Ci sono altre varianti del procedimento, usate quando si esegue la serializzazione di molti oggetti o quando non si vuole scrivere i dati ottenuti in un file

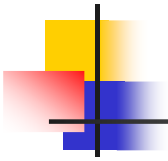


Python – Classi

- Il **meccanismo delle classi** in Python è un miscuglio dei meccanismi delle classi che si trovano in C++ e Modula-3.
- Come in Smalltalk, le classi in sé sono oggetti, nel senso più ampio del termine: in Python, tutti i tipi di dati sono oggetti.
- Diversamente da quanto accade in C++ o Modula-3, i tipi built-in non possono essere usati come classi base per estensioni utente.
- il termine "oggetto" in Python non significa necessariamente un'istanza di classe.

Python – Classi

- Lo **spazio dei nomi** è una mappa che collega i nomi agli oggetti.
- Esempi di spazi dei nomi sono: l'insieme dei nomi built-in (funzioni come abs() ed i nomi delle eccezioni built-in), i nomi globali in un modulo e i nomi locali in una chiamata di funzione.
- Gli spazi dei nomi vengono creati in momenti diversi ed hanno tempi di sopravvivenza diversi.
  - Lo spazio dei nomi che contiene i nomi built-in, chiamato __builtin__, viene creato all'avvio dell'interprete Python e non viene mai cancellato.
  - Le istruzioni eseguite dall'invocazione a livello più alto dell'interprete, lette da un file di script o interattivamente, vengono considerate parte di un modulo chiamato __main__.
  - Lo spazio dei nomi globale di un modulo viene creato quando viene letta la definizione del modulo; normalmente anche lo spazio dei nomi del modulo dura fino al termine della sessione.

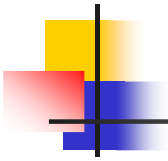


Python – Classi

- Uno **scope** è una regione del codice di un programma Python dove uno spazio dei nomi è accessibile direttamente. "Direttamente accessibile" qui significa che un riferimento non completamente qualificato ad un nome cerca di trovare tale nome nello spazio dei nomi.
- Sebbene gli scope siano determinati staticamente, essi sono usati dinamicamente. In qualunque momento durante l'esecuzione sono in uso esattamente tre scope annidati (cioè, esattamente tre spazi dei nomi sono direttamente accessibili): lo scope più interno, in cui viene effettuata per prima la ricerca, contiene i nomi locali, lo scope mediano, esaminato successivamente, contiene i nomi globali del modulo corrente, e lo scope più esterno (esaminato per ultimo) è lo spazio dei nomi che contiene i nomi built-in.

Python – Classi

- La forma più semplice di **definizione di una classe** è:
- class NomeClasse:
  <istruzione-1>
  . . .
  <istruzione-N>
- Le definizioni di classe possono essere poste in qualsiasi punto di un programma ma solitamente per questioni di leggibilità sono poste all'inizio, subito sotto le istruzioni import.
- Quando viene introdotta una definizione di classe, viene creato un nuovo spazio dei nomi, usato come scope locale.
- Quando una definizione di classe è terminata normalmente (passando per la sua chiusura), viene creato un oggetto classe. Esso è fondamentalmente un involucro (wrapper) per i contenuti dello spazio dei nomi creato dalla definizione della classe

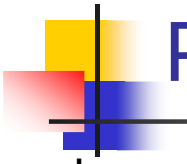


Python – Classi

- **Esempio di creazione di classe**:
  class Punto:
  pass
- Creando la classe Punto abbiamo anche creato un nuovo tipo di dato chiamato con lo stesso nome. I membri di questo tipo sono detti istanze del tipo o oggetti. La creazione di una nuova istanza è detta istanziazione: solo al momento dell'istanziazione parte della memoria è riservata per depositare il valore dell'oggetto. Per creare un oggetto di tipo Punto viene chiamata una funzione chiamata Punto:
  P1 = Punto()
- Alla variabile P1 è assegnato il riferimento ad un nuovo oggetto Punto. Una funzione come Punto, che crea nuovi oggetti e riserva quindi della memoria per depositarne i valori, è detta costruttore.

Python – Classi

- Possiamo aggiungere un nuovo dato ad un'istanza usando la notazione punto:
  >>> P1.x = 3.0
  >>> P1.y = 4.0
- Questa sintassi è simile a quella usata per la selezione di una variabile appartenente ad un modulo, tipo math.pi e string.uppercase. In questo caso stiamo selezionando una voce da un'istanza e queste voci che fanno parte dell'istanza sono dette attributi.
- gli attributi sono aggiunti, e possono essere anche eliminati (tramite del); le operazioni di aggiunta ed eliminazione operano sullo spazio dei nomi della classe, ampliandolo o riducendolo
- la stampa di un oggetto:
  >>> print P1
  <__main__.Punto instance at 80f8e70>
- Il risultato indica che P1 è un'istanza della classe Punto e che è stato definito in __main__. 80f8e70 è l'identificatore univoco dell'oggetto, scritto in base 16 (esadecimale).

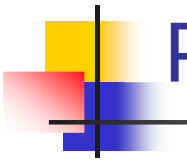


Python – Classi

- La variabile P1 si riferisce ad un oggetto Punto che contiene due attributi ed ogni attributo (una coordinata) si riferisce ad un numero in virgola mobile.
- Possiamo leggere il valore di un attributo con la stessa sintassi:

```
>>> print P1.y
4.0
>>> x = P1.x
>>> print x
3.0
```

- L'espressione P1.x significa "vai all'oggetto puntato da P1 e ottieni il valore del suo attributo x". In questo caso assegniamo il valore ad una variabile chiamata x: non c'è conflitto tra la variabile locale x e l'attributo x di P1: lo scopo della notazione punto è proprio quello di identificare la variabile cui ci si riferisce evitando le ambiguità.

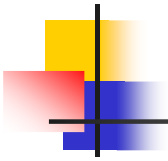


Python – Classi

- Gli **oggetti classe** supportano quindi **due tipi di operazioni: riferimenti ad attributo e istanziazione.**
- I **riferimenti ad attributo** usano la sintassi oggetto.nome. Nomi di attributi validi sono tutti i nomi che si trovavano nello spazio dei nomi della classe al momento della creazione dell'oggetto classe. Così, se la definizione di classe fosse del tipo:

```
class MiaClasse:
    """Una semplice classe d'esempio"""
    i = 12345
    def f(self):
      return 'ciao mondo'
```

- MiaClasse.i e MiaClasse.f sarebbero riferimenti validi ad attributi, che restituirebbero rispettivamente un intero ed un oggetto metodo.
- Sugli attributi di una classe è anche possibile effettuare assegnamenti, quindi è possibile cambiare il valore di MiaClasse.i con un assegnamento.
- Anche __doc__ è un attributo valido, in sola lettura, che restituisce la stringa di documentazione della classe: "Una semplice classe di esempio".



Python – Classi

- **L'istanziazione** di una classe, già vista con *P1 = Punto()* , crea una nuova istanza della classe
- L'instanziazione crea un oggetto vuoto. In molti casi si preferisce che vengano creati oggetti con uno stato iniziale noto. Perciò una classe può definire un metodo speciale chiamato __init__(), ad esempio:

```
def __init__(self):
self.data = []
```

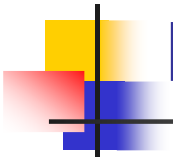
- Quando una classe definisce un metodo __init__(), la sua instanziazione invoca automaticamente __init__() per l'istanza di classe appena creata.
- Naturalmente il metodo __init__() può avere argomenti, ad esempio:

```
>>> class Complesso:
...         def __init__(self, partereale, partimag):
...             self.r = partereale
...             self.i = partimag
...
>>> x = Complesso(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```


Python – Classi

- Ora, cosa possiamo fare con **gli oggetti istanza**? Le sole operazioni che essi conoscono per mezzo dell'istanziazione degli oggetti sono i riferimenti ad attributo. Ci sono **due tipi di nomi di attributo validi: dati e metodi**
- Gli **attributi dato** corrispondono alle "variabili istanza" in Smalltalk, e ai "dati membri" in C++. Gli attributi dato non devono essere dichiarati; come le variabili locali, essi vengono alla luce quando vengono assegnati per la prima volta. Per esempio, se x è l'istanza della MiaClasse precedentemente creata, il seguente pezzo di codice stamperà il valore 16, senza lasciare traccia:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

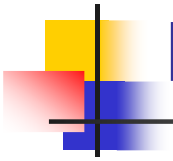


Python – Classi

- Il **secondo tipo** di riferimenti ad attributo conosciuti dagli oggetti istanza sono i **metodi**. Un metodo è una funzione che "appartiene a" un oggetto. In Python, il termine metodo non è prerogativa delle istanze di classi: altri tipi di oggetto possono benissimo essere dotati di metodi, ad esempio gli oggetti lista hanno i metodi append, insert, remove, ecc.
- I nomi dei metodi validi per un oggetto istanza dipendono dalla sua classe. Per definizione, tutti gli attributi di una classe che siano oggetti funzione (definiti dall'utente) definiscono metodi corrispondenti alle sue istanze. Così nel nostro esempio x.f è un riferimento valido ad un metodo, dato che MiaClasse.f è una funzione, ma x.i non lo è, dato che MiaClasse.i non è una funzione. Però x.f non è la stessa cosa di MiaClasse.f: è un oggetto metodo, non un oggetto funzione.

Python – Classi

- x.f() è stato invocato nell'esempio sopra senza argomenti, anche se la definizione di funzione per f specificava un argomento. Che cosa è accaduto all'argomento?
- La particolarità dei metodi è che l'oggetto viene passato come primo argomento della funzione. Nel nostro esempio, la chiamata x.f() è esattamente equivalente a MiaClasse.f(x). In generale, invocare un metodo con una lista di n argomenti è equivalente a invocare la funzione corrispondente con una lista di argomenti creata inserendo l'oggetto cui appartiene il metodo come primo argomento.
- Gli attributi dato prevalgono sugli attributi metodo con lo stesso nome; per evitare accidentali conflitti di nomi, che potrebbero causare bug difficili da scovare in programmi molto grossi, è saggio usare una qualche convenzione che minimizzi le possibilità di conflitti
- Si noti che gli utilizzatori finali possono aggiungere degli attributi dato propri ad un oggetto istanza senza intaccare la validità dei metodi, fino quando vengano evitati conflitti di nomi.

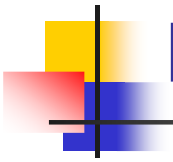


Python – Classi

- Convenzionalmente, il primo argomento dei metodi è chiamato self. Il nome self non ha alcun significato speciale in Python.
- Qualsiasi oggetto funzione che sia attributo di una classe definisce un metodo per le istanze di tale classe. Non è necessario che il codice della definizione di funzione sia racchiuso nella definizione della classe: va bene anche assegnare un oggetto funzione a una variabile locale nella classe. Per esempio:

```
# Funzione definita all'esterno della classe
def f1(self, x, y):
    return min(x, x+y)
class C:
    f = f1
    def g(self):
        return 'ciao mondo'
```

- Ora f e g sono tutti attributi che si riferiscono ad oggetti funzione, di conseguenza sono tutti metodi delle istanze della classe C



Python – Classi

- I metodi possono chiamare altri metodi usando gli attributi metodo dell'argomento self:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Python – Classi

- I **metodi** sono simili alle funzioni, ma sono definiti all'interno della definizione di classe per rendere più esplicita la relazione tra la classe ed i metodi corrispondenti, e la sintassi per invocare un metodo è diversa da quella usata per chiamare una funzione.
- Ad esempio, una classe chiamata Tempo e scritto una funzione StampaTempo:

```
class Tempo:
    pass
def StampaTempo(Orario):
    print str(Orario.Ore) + ":" + str(Orario.Minuti) + ":" +
        str(Orario.Secondi)
```

Per chiamare la funzione abbiamo passato un oggetto Tempo come parametro:

```
>>> OraAttuale = Tempo()
>>> OraAttuale.Ore = 9
>>> OraAttuale.Minuti = 14
>>> OraAttuale.Secondi = 30
>>> StampaTempo(OraAttuale)
```


Python – Classi

- Per rendere StampaTempo un metodo si deve muovere la definizione della funzione all'interno della definizione della classe:

```
class Tempo:
  def StampaTempo(self):
    print str(self.Ore) + ":" +   |
        str(self.Minuti) + ":" +  |
        str(self.Secondi)
```

- Ora possiamo invocare StampaTempo usando la notazione punto.
- >>> OraAttuale.StampaTempo()
- Questo cambio di prospettiva non sembra così utile, ma in realtà lo spostamento della responsabilità dalla funzione all'oggetto rende possibile scrivere funzioni più versatili e rende più immediati il mantenimento ed il riutilizzo del codice

Python – Classi

```
class Tempo:
  ...
  def Dopo(self, Tempo2):
    if self.Ore > Tempo2.Ore:
      return 1
    if self.Ore < Tempo2.Ore:
      return 0
    if self.Minuti > Tempo2.Minuti:
      return 1
    if self.Minuti < Tempo2.Minuti:
      return 0
    if self.Secondi > Tempo2.Secondi:
      return 1
    return 0
```


Python – Classi

- Invochiamo questo metodo su un oggetto e passiamo l'altro come argomento:

```
>>> OraCorrente = Tempo()
>>> OraCorrente.Ore = 9
>>> OraCorrente.Minuti = 14
>>> OraCorrente.Secondi=  30
>>> TempoCottura = Tempo()
>>> TempoCottura.Ore =  3
>>> TempoCottura.Minuti = 35
>>> TempoCottura.Secondi = 0
if TempoCottura.Dopo(OraCorrente):
    print "Il pranzo e' pronto"
```

Python – Classi

- Riscriviamo la classe Punto in uno stile OO:

```
class Punto:
  def __init__(self, x=0, y=0):
    self.x = x
    self.y = y
  def __str__(self):
    return '(' + str(self.x) + ', ' + str(self.y) + ')'
```

- Il metodo di inizializzazione prende x e y come parametri opzionali. Il loro valore di default è 0.
- Il metodo __str__ ritorna una rappresentazione di un oggetto Punto sotto forma di stringa. Se una classe fornisce un metodo chiamato __str__ questo sovrascrive la funzione str di Python.
- >>> P = Punto(3, 4)
- >>> str(P)
- '(3, 4)'
- la definizione di __str__ cambia anche print (che invoca __str__):
- >>> print P
- (3, 4)
- Quando scriviamo una nuova classe iniziamo quasi sempre scrivendo __init__ (rende più facile istanziare) e __str__ (utile per il debug)

Python – Classi

- Per verificare **l'uguaglianza fra oggetti**, analogamente a Java, si può solo verificare l'uguaglianza dei riferimenti (debole) o controllare l'uguaglianza degli attributi corrispondenti, definendo una funzione apposita (come equals in Java):

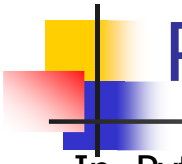
```
>>> P1 = Punto()
>>> P1.x = 3
>>> P1.y = 4
>>> P2 = Punto()
>>> P2.x = 3
>>> P2.y = 4
```

- uguaglianza debole

```
>>> P1 == P2
0
```

- uguaglianza forte

```
def StessoPunto(P1, P2) :
  return (P1.x == P2.x) and (P1.y == P2.y)
```



Python – Classi

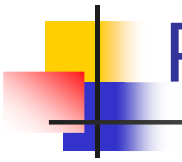
- In Python si può **cambiare la definizione degli operatori predefiniti** quando applicati a tipi definiti dall'utente ("overloading dell'operatore").
- Se vogliamo ridefinire l'operatore somma + scriveremo un metodo chiamato __add__:

```
class Punto:
  ...
  def __add__(self, AltroPunto):
    return Punto(self.x + AltroPunto.x, self.y + AltroPunto.y)
```

- Quando applicheremo l'operatore + ad oggetti Punto Python invocherà il metodo __add__:

```
>>>  P1 = Punto(3, 4)
>>>  P2 = Punto(5, 7)
>>>  P3 = P1 + P2
>>>  print P3
(8, 11)
```

- L'espressione P1 + P2 è equivalente a P1.__add__(P2) ma ovviamente più elegante.



Python – Classi

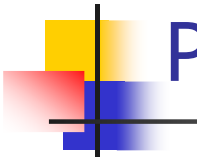
- E' anche possibile ridefinire l'operatore moltiplicazione, aggiungendo il metodo __mul__ o __rmul__ o entrambi.
- Se l'operatore a sinistra di * è di tipo Punto, Python invoca __mul__ assumendo che anche l'altro operando sia un oggetto di tipo Punto. In questo caso si calcola il prodotto dei due punti secondo le regole dell'algebra lineare:

```
def __mul__(self, AltroPunto):
return self.x * AltroPunto.x + self.y * AltroPunto.y
```

- Se l'operando a sinistra di * è un tipo primitivo e l'operando a destra è di tipo Punto, Python invocherà __rmul__ per calcolare una moltiplicazione scalare:

```
def __rmul__(self, AltroPunto):
return Punto(AltroPunto * self.x,  AltroPunto * self.y)
```

- Il risultato della moltiplicazione scalare è un nuovo punto le cui coordinate sono un multiplo di quelle originali. Se AltroPunto è un tipo che non può essere moltiplicato per un numero in virgola mobile __rmul__ produrrà un errore in esecuzione.



Python – Classi

- Questo esempio mostra entrambi i tipi di moltiplicazione:

```
>>> P1 = Punto(3, 4)
>>> P2 = Punto(5, 7)
>>> print P1 * P2
43
>>> print 2 * P2
(10, 14)
```

- Cosa accade se proviamo a valutare P2 * 2? Dato che il primo parametro è un Punto Python invoca __mul__ con 2 come secondo argomento. All'interno di __mul__ il programma prova ad accedere la coordinata x di AltroPunto e questo tentativo genera un errore dato che un numero intero non ha attributi:

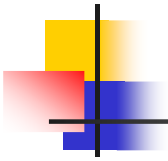
```
>>> print P2 * 2
AttributeError: 'int' object has no attribute 'x'
```


Python – Classi

- In generale, le funzioni per gli operatori sono:

__add__ (self, other)	__radd__ (self, other)
__sub__ (self, other)	__rsub__ (self, other)
__mul__ (self, other)	__rmul__ (self, other)
__div__ (self, other)	__rdiv__ (self, other)
__mod__ (self, other)	__rmod__ (self, other)
__divmod__ (self, other)	__rdivmod__ (self, other)
__pow__ (self, other[, modulo])	__rpow__ (self, other)
__lshift__ (self, other)	__rlshift__ (self, other)
__rshift__ (self, other)	__rrshift__ (self, other)
__and__ (self, other)	__rand__ (self, other)
__xor__ (self, other)	__rxor__ (self, other)
__or__ (self, other)	__ror__ (self, other)

- Le funzioni con in prefisso 'r' (reverse operands) vengono invocate solo se l'operando sinistro non supporta l'operazione, come visto nell'esempio



Python – Classi

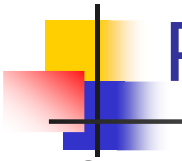
- La maggior parte dei metodi che abbiamo scritto finora lavorano solo per un tipo specifico di dati.
- Ci sono comunque operazioni che si vorrebbe poter applicare a molti tipi. Se più tipi di dato supportano lo stesso insieme di operazioni si possono scrivere funzioni (**polimorfiche**) che lavorano indifferentemente con ciascuno di questi tipi.
- Per esempio l'operazione MoltSomma (comune in algebra lineare) prende tre parametri: il risultato è la moltiplicazione dei primi due e la successiva somma del terzo al prodotto. Possiamo scriverla così:

```
def MoltSomma(x, y, z):
  return x * y + z
```

- Questo metodo lavorerà per tutti i valori di x e y che possono essere moltiplicati e per ogni valore di z che può essere sommato al prodotto.

Python – Classi

- Possiamo invocarla con valori numerici:
- >>> MoltSomma(3, 2, 1)
- 7
- o con oggetti di tipo Punto:
- >>> P1 = Punto(3, 4)
- >>> P2 = Punto(5, 7)
- >>> print MoltSomma(2, P1, P2)
- (11, 15)
- >>> print MoltSomma(P1, P2, 1)
- 44
- Nel primo caso il punto P1 è moltiplicato per uno scalare e il prodotto è poi sommato a un altro punto (P2). Nel secondo caso il prodotto punto produce un valore numerico al quale viene sommato un altro valore numerico.
- Una funzione che accetta parametri di tipo diverso è chiamata **polimorfica**.



Python – Classi

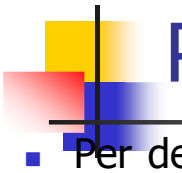
- Come esempio ulteriore di **polimorfismo** consideriamo il metodo DirittoERovescio che stampa due volte una stringa, prima direttamente e poi all'inverso:

```
def DirittoERovescio(Stringa):
    import copy
    Rovescio = copy.copy(Stringa)
    Rovescio.reverse()
    print str(Stringa) + str(Rovescio)
```

- Dato che il metodo reverse è un modificatore si deve fare una copia della stringa prima di rovesciarla: in questo modo il metodo reverse non modificherà la lista originale ma solo una sua copia.
- Ecco un esempio di funzionamento di DirittoERovescio con le liste:

```
>>>  Lista = [1, 2, 3, 4]
>>>  DirittoERovescio(Lista)
[1, 2, 3, 4][4, 3, 2, 1]
```

- Era facilmente intuibile che questa funzione riuscisse a maneggiare le liste. Ma può lavorare con oggetti di tipo Punto?

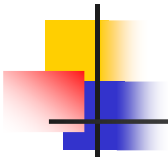


Python – Classi

- Per determinare se una funzione può essere applicata ad un tipo nuovo applichiamo la **regola fondamentale del polimorfismo**:
- Se tutte le operazioni all'interno della funzione possono essere applicate ad un tipo di dato allora la funzione stessa può essere applicata al tipo.
- Le operazioni in *DirittoERovescio* includono copy, reverse e print.
- copy funziona su ogni oggetto e abbiamo già scritto un metodo __str__ per gli oggetti di tipo Punto così l'unica cosa che ancora ci manca è il metodo reverse:

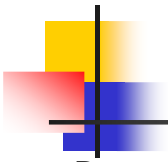
```
def reverse(self):
    self.x , self.y = self.y, self.x
Ora possiamo passare Punto a DirittoERovescio:
>>>  P = Punto(3, 4)
>>>  DirittoERovescio(P)
(3, 4)(4, 3)
```

- Il miglior tipo di polimorfismo è quello involontario, quando si scopre che una funzione già scritta può essere applicata ad un tipo di dati per cui non era stata pensata.



Python – Classi

- Esempio di **composizione di oggetti**:
- Consideriamo un mazzo di carte americano: è composto da 52 carte, ognuna delle quali appartiene a un seme (picche, cuori, quadri, fiori, nell'ordine di importanza nel gioco del bridge) ed è identificata da un numero da 1 a 13 (detto "rango"). I valori rappresentano, in ordine crescente, l'Asso, la serie numerica da 2 a 10, il Jack, la Regina ed il Re. A seconda del gioco il valore dell'Asso può essere considerato inferiore al 2 o superiore al Re.
- Volendo definire un nuovo oggetto per rappresentare una carta da gioco è ovvio che gli attributi devono essere il rango ed il seme. Si stabilisce la seguente codifica:
  - Picche -> 3, Cuori -> 2, Quadri -> 1, Fiori -> 0
  - Asso -> 1, Jack -> 11, Regina -> 12, Re -> 13



Python – Classi

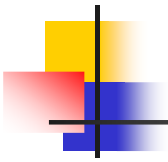
- Passando al codice, si ha:

```
class Carta:
  ListaSemi = ["Fiori", "Quadri", "Cuori", "Picche"]
  ListaRanghi = ["impossibile", "Asso", "2", "3", "4", "5", "6",|
                 "7", "8", "9", "10", "Jack", "Regina", "Re"]
  def __init__(self, Seme=0, Rango=1):
    self.Seme = Seme
    self.Rango = Rango
  def __str__(self):
    return (self.ListaRanghi[self.Rango] + " di " +
            self.ListaSemi[self.Seme])
```

- Il motivo della presenza dell'elemento "impossibile" nel primo elemento di ListaRanghi è di agire come segnaposto per l'elemento 0
- Per creare un oggetto che rappresenta il 3 di fiori si scriverà: TreDiFiori = Carta(0, 3), dove il primo argomento (0) rappresenta il *seme* fiori ed il secondo (3) il *rango* della carta.

Python – Classi

- Definendo altri oggetti:
  ```
  >>> Carta1 = Carta(1, 11)
  >>> print Carta1
  Jack di Quadri
  >>> Carta2 = Carta(1, 3)
  >>> print Carta2
  3 di Quadri
  >>> print Carta2.ListaSemi[1]
  Quadri
  ```
- Lo svantaggio sta nel fatto che se modifichiamo un attributo di classe questo cambiamento si riflette in ogni istanza della classe. Per esempio se decidessimo di cambiare il seme "Quadri" in "Bastoni"...
  ```
  >>> Carta1.ListaSemi[1] = "Bastoni"
  >>> print Carta1
  Jack di Bastoni
  ```
- ...*tutti* i Quadri diventerebbero dei Bastoni:
  ```
  >>> print Carta2
  3 di Bastoni
  ```
- Non è solitamente una buona idea modificare gli attributi di classe.



Python – Classi

- Per i tipi primitivi sono già definiti operatori condizionali (<, >, ==, ecc.) che confrontano i valori. Per i tipi definiti dall'utente si può ridefinire il comportamento di questi operatori aggiungendo il metodo __cmp__
- __cmp__ prende due parametri, self e Altro, e ritorna 1 se self>Altro, -1 viceversa e 0 se sono uguali.
- Per rendere confrontabili le carte dobbiamo innanzitutto decidere quale attributo sia il più importante, se il rango o il seme. Si decide che il seme ha priorità rispetto al rango:
  ```
  def __cmp__(self, Altro):
    # controlla il seme
    if self.Seme > Altro.Seme: return 1
    if self.Seme < Altro.Seme: return -1
    # se i semi sono uguali controlla il rango
    if self.Rango > Altro.Rango: return 1
    if self.Rango < Altro.Rango: return -1
    # se anche i ranghi sono uguali le carte sono uguali!
    return 0
  ```
- In questo tipo di ordinamento gli Assi hanno valore più basso dei 2.

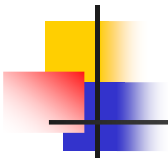
Python – Classi

- Quella che segue è la definizione di classe della classe Mazzo, e due metodi per la stampa:

```
class Mazzo:
  def __init__(self):
    self.Carte = []
    for Seme in range(4):
      for Rango in range(1, 14): self.Carte.append(Carta(Seme, Rango))

class Mazzo:
  ...
  def StampaMazzo(self):
    for Carta in self.Carte:        print Carta

class Mazzo:
  ...
  def __str__(self):
    s = ""
    for i in range(len(self.Carte)): s = s + " "*i + str(self.Carte[i]) + "\n"
    return s
```

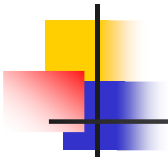



Python – Classi

- Le classi in Python permettono **l'ereditarietà**. La sintassi per la definizione di una classe derivata ha la forma seguente:

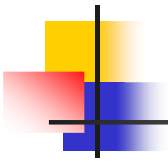
```
class NomeClasseDerivata(NomeClasseBase):
<istruzione-1>
. . .
<istruzione-N>
```

- Il nome NomeClasseBase dev'essere definito in uno scope contenente la definizione della classe derivata.
- L'esecuzione della definizione di una classe derivata procede nello stesso modo di una classe base. Quando viene costruito l'oggetto classe, la classe base viene memorizzata. Viene usata per risolvere i riferimenti ad attributi: se un attributo richiesto non viene rinvenuto nella classe, allora viene cercato nella classe base. Tale regola viene applicata ricorsivamente se la classe base è a sua volta derivata.
- Le classi derivate possono sovrascrivere i metodi delle loro classi base. Con "NomeClasseBase.nomemetodo(self, argomenti)" si può richiedere l'esecuzione del metodo sovrascritto, come in Java



Python – Classi

- Python supporta pure l'**ereditarietà multipla**. Una definizione di classe con classi base multiple ha la forma seguente:
  class NomeClasseDerivata(Base1, Base2, Base3):
  <istruzioni>
- La sola regola necessaria per chiarire la semantica è la regola di risoluzione usata per i riferimenti agli attributi di classe. Essa è prima-in-profondità, da-sinistra-a-destra. Perciò, se un attributo non viene trovato in NomeClasseDerivata, viene cercato in Base1, poi (ricorsivamente) nelle classi base di Base1 e, solo se non vi è stato trovato, viene ricercato in Base2, e così via.
- Ad alcuni una regola "prima in larghezza" (breadth first), che ricerca in Base2 e Base3 prima che nelle classi base di Base1, sembra più naturale. Comunque ciò richiederebbe di sapere se un particolare attributo di Base1 sia in effetti definito in Base1 o in una delle sue classi base prima che si possano valutare le conseguenze di un conflitto di nomi con un attributo di Base2.



Python – Classi

- l'esecuzione della maggior parte degli oggetti può essere replicata ciclicamente mediante un'istruzione for:
  for elemento in [1, 2, 3]: print elemento
  for elemento in (1, 2, 3): print elemento
  for chiave in {'uno':1, 'due':2}: print chiave
  for carattere in "123": print carattere
  for line in open("myfile.txt"): print line
- Questo stile d'accesso utilizza gli **iteratori**; dietro le quinte, l'istruzione for richiama sull'oggetto contenitore la funzione iter(): l'oggetto iteratore da essa restituito definisce il metodo next(), il quale introduce degli elementi nel contenitore uno per volta. Quando non ci sono più elementi, next() solleva un'eccezione StopIteration che termina il ciclo iniziato dal for.

Python – Classi

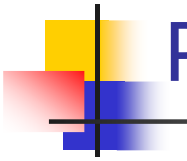
- L'esempio che segue mostra il funzionamento dell'iteratore

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> it.next()
'a'
>>> it.next()
'b'
>>> it.next()
'c'
>>> it.next()
Traceback (most recent call last): File "<pyshell#6>", line 1, in -
   toplevel-
it.next()
StopIteration
```


Python – Classi

- Visti i meccanismi di base del protocollo iteratore, è semplice aggiungere un comportamento iteratore alle proprie classi, basta definire un metodo __iter__() che restituisca un oggetto con un metodo next(). Se la classe definisce next(), __iter__() può restituire solo self:

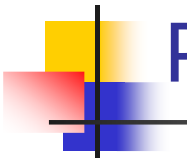
```
>>> class Reverse:
"Iteratore per eseguire un ciclo al contrario su una sequenza"
def __init__(self, data):
    self.data = data
    self.index = len(data)
def __iter__(self):
    return self
def next(self):
    if self.index == 0:
      raise StopIteration
    self.index = self.index - 1
    return self.data[self.index]
>>> for carattere in Reverse('spam'): print carattere
m
a
p
s
```



Python – Classi

- I **generatori** sono semplici ma efficaci strumenti per creare iteratori. Sono scritti come funzioni regolari, pur usando l'istruzione yield ogni qualvolta restituiscano dei dati. Siccome ricorda tutti i valori dei dati e l'ultima espressione eseguita, alla chiamata a next() il generatore riprende da dove s'era fermato:

```
>>> def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]
>>> for char in reverse('golf'):
        print char
f
l
o
g
```

Python – Classi

- Ciò che è fattibile con i generatori può essere fatto con iteratori basati su una classe, come visto nella precedente sezione. La creazione automatica dei metodi __iter__() e next() rende i generatori così compatti.
- Un'altra caratteristica chiave è il salvataggio automatico delle variabili locali e dello stato d'esecuzione tra una chiamata e l'altra, cosa che rende la funzione più facile a scriversi e più chiara di un approccio con variabili di classe come self.index e self.data.
- Oltre alla creazione automatica di metodi e salvataggio dello stato del programma, quando terminano, i generatori sollevano l'eccezione StopIteration. Insieme, queste caratteristiche facilitano la creazione di iteratori con la semplice scrittura di una normalissima funzione.

Python – Classi

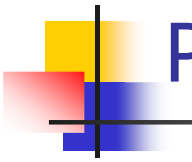
- La programmazione orientata agli oggetti permette al programmatore di creare nuovi tipi di dato. Esploreremo questa capacità costruendo una **classe di esempio, Frazione,** che possa lavorare come i tipi di dato numerico predefiniti:

```
class Frazione:
  def __init__(self, Numeratore, Denominatore=1):
    self.Numeratore = Numeratore
    self.Denominatore = Denominatore
def __str__(self):
    return "%d/%d" % (self.Numeratore, self.Denominatore)
```

- Per testare il lavoro, lo si salva in in un file chiamato frazione.py e lo si importa nell'interprete:

```
>>> from Frazione import Frazione
>>> f = Frazione(5,6)
>>> print "La frazione e'", f
La frazione e' 5/6
```

- Come abbiamo già visto il comando print invoca il metodo __str__ implicitamente.



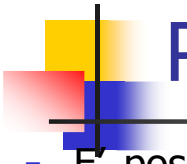
Python – Classi

- Ci interessa poter applicare le consuete operazioni matematiche a operandi di tipo Frazione. Per farlo procediamo con la ridefinizione degli operatori matematici quali l'addizione, la sottrazione, la moltiplicazione e la divisione.
- Iniziamo dalla moltiplicazione perché è la più semplice da implementare. Il risultato della moltiplicazione di due frazioni è una frazione che ha come numeratore il prodotto dei due numeratori, e come denominatore il prodotto dei denominatori. __mul__ è il nome usato da Python per indicare l'operatore *:

```
class Frazione:
  ...
  def __mul__(self, Altro):
    return Frazione(self.Numeratore * Altro.Numeratore,
                    self.Denominatore * Altro.Denominatore)
```

- Possiamo testare subito questo metodo calcolando il prodotto di due frazioni:

```
>>> print Frazione(5,6) * Frazione(3,4)
15/24
```



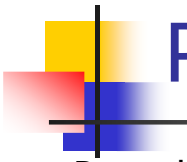
Python – Classi

- E' possibile estendere il metodo per gestire la moltiplicazione di una frazione per un intero, usando la funzione built-in type per controllare se Altro è un intero. In questo caso prima di procedere con la moltiplicazione lo si convertirà in frazione:

```
class Frazione:
  ...
  def __mul__(self, Altro):
    if type(Altro) == type(5):
      Altro = Frazione(Altro)
    return Frazione(self.Numeratore * Altro.Numeratore,
                    self.Denominatore * Altro.Denominatore)
```

- La moltiplicazione tra frazioni e interi ora funziona, ma solo se la frazione compare alla sinistra dell'operatore:

```
>>> print Frazione(5,6) * 4
20/6
>>> print 4 * Frazione(5,6)
TypeError: unsupported operand type(s) for *: 'int' and 'instance
```

Python – Classi

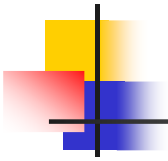
- Per valutare l'operatore di moltiplicazione, Python controlla l'operando di sinistra per vedere se questo fornisce un metodo __mul__ che supporta il tipo del secondo operando. Se il controllo non ha successo Python passa a controllare l'operando di destra per vedere se è stato definito un metodo __rmul__ che supporta il tipo di dato dell'operatore di sinistra. Visto che non abbiamo ancora scritto __rmul__ il controllo fallisce. E' possibile scrivere __rmul__ come segue:

```
class Frazione:
  ...
  __rmul__ = __mul__
```

- Con questa assegnazione diciamo che il metodo __rmul__ è lo stesso di __mul__, così che per valutare 4 * Fraction(5,6) Python invoca __rmul__ sull'oggetto Frazione e passa 4:

```
>>> print 4 * Frazione(5,6)
20/6
```

- Dato che __rmul__ è lo stesso di __mul__ e che quest'ultimo accetta parametri interi è tutto a posto.



Python – Classi

- L'addizione è più complessa della moltiplicazione ma non troppo: la somma di a/b e c/d è infatti la frazione (a*d+c*b)/b*d.
- Usando il codice della moltiplicazione come modello possiamo scrivere __add__ e __radd__:

```
class Frazione:
  ...
  def __add__(self, Altro):
    if type(Altro) == type(5):
      Altro = Frazione(Altro)
    return Fraction(self.Numeratore   * Altro.Denominatore +
                    self.Denominatore * Altro.Numeratore,
                    self.Denominatore * Altro.Denominatore)
  __radd__ = __add__
```

- Possiamo testare questi metodi con frazioni e interi:

```
>>> print Frazione(5,6) + Frazione(5,6)        60/36
>>> print Frazione(5,6) + 3                    23/6
>>> print 2 + Frazione(5,6)                    17/6
```

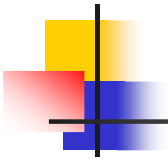
- I primi due esempi invocano __add__; l'ultimo __radd__.

Python – Classi

- Per ridurre la frazione ai suoi termini più semplici dobbiamo dividere il numeratore ed il numeratore per il loro massimo comune divisore (MCD).
- In generale quando creiamo e gestiamo un oggetto Frazione dovremmo sempre dividere numeratore e denominatore per il loro MCD.
- Euclide di Alessandria (circa 325--265 A.C.) inventò un algoritmo per calcolare il massimo comune divisore tra due numeri interi m e n: Se n divide perfettamente m allora il MCD è n. In caso contrario il MCD è il MCD tra n ed il resto della divisione di m diviso per n. Questa definizione ricorsiva può essere espressa in modo conciso con una funzione:

```
def MCD(m, n):
  if m % n == 0:
    return n
  else:
    return MCD(n, m%n)
```

- Nella prima riga del corpo usiamo l'operatore modulo per controllare la divisibilità. Nell'ultima riga lo usiamo per calcolare il resto della divisione.



Python – Classi

- Dato che tutte le operazioni che abbiamo scritto finora creano un nuovo oggetto Frazione come risultato potremmo inserire la riduzione nel metodo di inizializzazione:

```
class Frazione:
  def __init__(self, Numeratore, Denominatore=1):
    mcd = MCD(numeratore, Denominatore)
    self.Numeratore   = Numeratore / mcd
    self.Denominatore = Denominatore / mcd
```

- Quando creiamo una nuova Frazione questa sarà immediatamente ridotta alla sua forma più semplice:

```
>>> Frazione(100,-36)
-25/9
```


Python – Classi

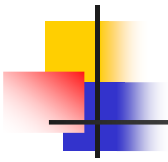
- Supponiamo di dover confrontare due oggetti di tipo Frazione, a e b valutando a == b. L'implementazione standard di == ritorna vero solo se a e b sono lo stesso oggetto, effettuando un confronto debole. Nel nostro caso vogliamo un confronto forte. Dobbiamo quindi insegnare alle frazioni come confrontarsi tra di loro. Si possono ridefinire tutti gli operatori di confronto fornendo un nuovo metodo __cmp__.
- Per convenzione __cmp__ ritorna un numero negativo se self è minore di Altro, zero se sono uguali e un numero positivo se self è più grande.
- Il modo più semplice per confrontare due frazioni è la moltiplicazione incrociata: se a/b > c/d allora ad > bc. Con questo in mente ecco quindi il codice per __cmp__:

```
class Frazione:
  ...
  def __cmp__(self, Altro):
    Differenza = (self.Numeratore  * Altro.Denominatore -
          Altro.Numeratore * self.Denominatore)
    return Differenza
```

- Se self è più grande di Altro allora Differenza è positiva. Se Altro è maggiore allora Differenza è negativa. Se sono uguali Differenza è zero.

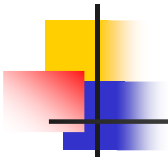
Python – Classi

- Dobbiamo ancora implementare la sottrazione ridefinendo __sub__ e la divisione con il corrispondente metodo __div__.
- Un modo per gestire queste operazioni è quello di implementare la negazione ridefinendo __neg__ e l'inversione con __invert__: possiamo infatti sottrarre sommando al primo operando la negazione del secondo, e dividere moltiplicando il primo operando per l'inverso del secondo. Poi dobbiamo fornire __rsub__ e __rdiv__.
- Purtroppo non possiamo usare la scorciatoia già vista nel caso di addizione e moltiplicazione dato che sottrazione e divisione non sono commutative. Non possiamo semplicemente assegnare __rsub__ e __rdiv__ a lle corrispondenti __sub__ e __div__, dato che in queste operazioni l'ordine degli operandi fa la differenza...
- Per gestire la negazione unaria, che non è altro che l'uso del segno meno con un singolo operando (da qui il termine "unaria" usato nella definizione), sarà necessario ridefinire il metodo __neg__.
- Potremmo anche calcolare le potenze ridefinendo __pow__ ma l'implementazione in questo caso è un po' complessa: se l'esponente non è un intero, infatti, può non essere possibile rappresentare il risultato come Frazione.



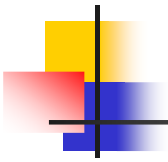
Python – Reference Library

- **Panoramica sulle librerie di riferimento di Python**:
- Il modulo **os** fornisce numerose funzioni per interagire con il Sistema Operativo:
- chdir(*path*)
  - Change the current working directory to *path*. Availability: Macintosh, Unix, Windows.
- getcwd()
  - Return a string representing the current working directory. Availability: Macintosh, Unix, Windows.
- open(*file, flags*[*, mode*])
  - Open the file *file* and set various flags according to *flags* and possibly its mode according to *mode*. The default *mode* is 0777 (octal), and the current umask value is first masked out. Return the file descriptor for the newly opened file. Availability: Macintosh, Unix, Windows.



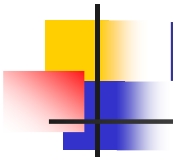
Python – Reference Library

- read(fd, n)
  - Read at most n bytes from file descriptor fd. Return a string containing the bytes read. If the end of the file referred to by fd has been reached, an empty string is returned. Availability: Macintosh, Unix, Windows.
- write(fd, str)
  - Write the string str to file descriptor fd. Return the number of bytes actually written. Availability: Macintosh, Unix, Windows.
  - Note: read and write functions are intended for low-level I/O and must be applied to a file descriptor as returned by open() or pipe(). To write a "file object" returned by the built-in function open() or by popen() or fdopen(), or sys.stdout or sys.stderr, use its write() method.



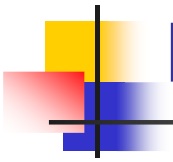
Python – Reference Library

- popen(command[, mode[, bufsize]])
  - Open a pipe to or from command. The return value is an open file object connected to the pipe, which can be read or written depending on whether mode is 'r' (default) or 'w'. The bufsize argument has the same meaning as the corresponding argument to the built-in open() function. The exit status of the command (encoded in the format specified for wait()) is available as the return value of the close() method of the file object, except that when the exit status is zero (termination without errors), None is returned. Availability: Macintosh, Unix, Windows.
- fsync(fd)
  - Force write of file with filedescriptor fd to disk. On Unix, this calls the native fsync() function; on Windows, the MS _commit() function.



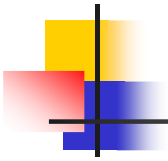
Python – Reference Library

- uname()
  - Return a 5-tuple containing information identifying the current operating system. The tuple contains 5 strings: (sysname, nodename, release, version, machine). Availability: recent flavors of Unix.
- getpid()
  - Return the current process id. Availability: Unix, Windows.
- getenv(varname[, value])
  - Return the value of the environment variable varname if it exists, or value if it doesn't. value defaults to None. Availability: most flavors of Unix, Windows.



Python – Reference Library

- system(command)
  - Execute the command (a string) in a subshell. This is implemented by calling the Standard C function system(), and has the same limitations.
  - Note that POSIX does not specify the meaning of the return value of the C system() function, so the return value of the Python function is system-dependent.
  - On Windows, the return value is that returned by the system shell after running command, given by the Windows environment variable COMSPEC: on command.com systems (Windows 95, 98 and ME) this is always 0; on cmd.exe systems (Windows NT, 2000 and XP) this is the exit status of the command run.
  - Availability: Macintosh, Unix, Windows.

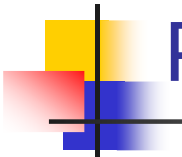


Python – Reference Library

- spawnl(mode, path, ...)
  - Execute the program path in a new process. If mode is P_NOWAIT, this function returns the process ID of the new process; if mode is P_WAIT, returns the process's exit code if it exits normally, or -signal, where signal is the signal that killed the process. On Windows, the process ID will actually be the process handle, so can be used with the waitpid() function.
- fork()
  - Fork a child process. Return 0 in the child, the child's process id in the parent. Availability: Macintosh, Unix.
- kill(pid, sig)
  - Send signal sig to the process pid. Constants for the specific signals available on the host platform are defined in the signal module. Availability: Macintosh, Unix.
- Le funzioni built-in dir() ed help() sono utili come aiuto interattivo quando si lavora con moduli di grosse dimensioni come os:
  >>> dir(os) <restituisce una lista di tutti i moduli delle funzioni>
  >>> help(os) <restituisce una pagina di manuale con le docstring dei moduli>

Python – Reference Library

- Molti script spesso invocano argomenti da riga di comando richiedendo che siano processati. Questi argomenti vengono memorizzati nel modulo **sys** ed argv è un attributo, come fosse una lista. Il seguente output fornisce il risultato dall'esecuzione di "python demo.py one two three"da riga di comando:
  >>> import sys
  >>> print sys.argv
  ['demo.py', 'one', 'two', 'three']
- Il modulo getopt processa sys.argv usando le convenzioni della funzione getopt() di Unix.
- Viene fornita una più potente e flessibile riga di comando dal modulo optparse.
- Il modulo sys ha anche attributi per stdin, stdout e stderr. L'ultimo attributo è utile per mostrare messaggi di avvertimento:
  >>> sys.stderr.write('file di log ... ')
  file di log …
- Per terminare l'esecuzione dello script usare "sys.exit()".



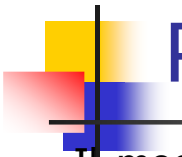
Python – Reference Library

- Il modulo **re** fornisce gli strumenti per processare le stringhe con le espressioni regolari. Per lavorare su complesse corrispondenze e manipolazioni, le espressioni regolari offrono una succinta, ottimizzata soluzione:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
```

- Quando vengono richieste solamente semplici funzionalità, è preferibile usare i metodi delle stringhe perché sono più semplici da leggere e da correggere se dovessero contenere degli errori:

```
>>> 'the per doo'.replace('doo', 'due')
'the per due'
```

Python – Reference Library

- Il modulo **math** fornisce l'accesso alle funzioni della sottostante libreria C per i numeri matematici in virgola mobile:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

- il modulo **random** fornisce gli strumenti per creare selezioni di numeri casuali:

```
>>> import random
>>> random.choice(['mela', 'pera', 'banana'])
'mela'
>>> random.sample(xrange(100), 10)
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random() # numero in virgola mobile casuale
0.17970987693706186
>>> random.randrange(6) # intero casuale scelto da range(6)
4
```

Python – Reference Library

- Vi sono numerosi moduli per accedere ad internet e processarne i protocolli. Due moduli semplici sono **urllib2** per recuperare dati da url e **smtplib** per spedire mail:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
... if 'EST' in line: # vedete Eastern Standard Time ...
print line
<BR>Nov. 25, 09:43:32 PM EST
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@tmp.org', 'jceasar@tmp.org',
"""
To: jceasar@tmp.org
From: soothsayer@tmp.org
Beware the Ides of March.
""")
>>> server.quit()
```


Python – Reference Library

- Il modulo **datetime** fornisce classi per manipolare data e tempo in modo semplice. Il modulo supporta anche oggetti che abbiano consapevolezza della propria time zone.

```
>>> from datetime import date
>>> now = date.today()
>>> now datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y or %d%b %Y is a %A on the %d day of %B")
'12-02-03 or 02Dec 2003 is a Tuesday on the 02 day of December'
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

Python – Reference Library

- Archiviazione e compressione vengono supportati direttamente dai moduli ed includono: **zlib, gzip, bz2, zipfile, tarfile**.
- >>> import zlib
- >>> s = 'witch which has which witches wrist watch'
- >>> len(s) 41
- >>> t = zlib.compress(s)
- >>> len(t) 37
- >>> zlib.decompress(t)
- 'witch which has which witches wrist watch`
- Python fornisce strumenti di misurazione delle prestazioni, ad esempio con il modulo **timeit**:
- >>> from timeit import Timer
- >>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
- 0.57535828626024577
- >>> Timer('a,b = b,a', 'a=1; b=2').timeit()
- 0.54962537085770791
- I moduli **profile** e **pstats** consentono di identificare sezioni critiche per la tempistica in blocchi di codice.