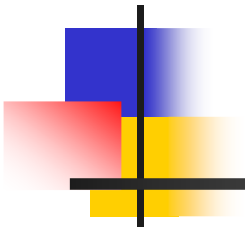
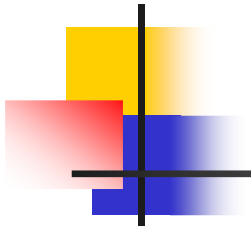
Linguaggi

Corso M-Z - Laurea in Ingegneria Informatica
A.A. 2009-2010

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

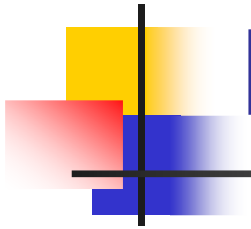


Il linguaggio Python



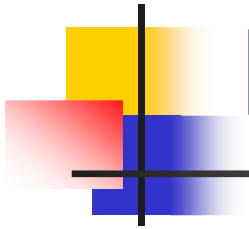
Python

- 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
  - Programmare in Python è facile ed immediato



Python - interprete

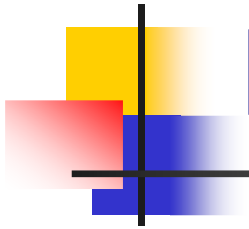
- Python è un linguaggio interpretato; Ci sono **due modi** di usare l'interprete: **a linea di comando o in modo script** (estensione .py)
- Le funzioni di editing di riga dell'interprete di solito non sono molto sofisticate. L'interprete opera all'incirca come una shell Unix: quando viene lanciato 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.



Python – tipi di dato numerici

- **Tipi di dato numerici in Python:**
- L'interprete si può comportare come una semplice calcolatrice: si può digitare un'espressione ed esso fornirà il valore risultante. Gli operatori +, -, * e / funzionano come negli altri linguaggi; 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:

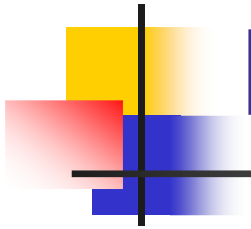
```
>>> larghezza = 20
>>> altezza = 5*9
>>> larghezza * altezza
900
```

- **Notare che non esiste una dichiarazione di tipo, una variabile è riutilizzabile quindi per valori e tipi diversi**
- Un valore può essere assegnato simultaneamente a variabili diverse:

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



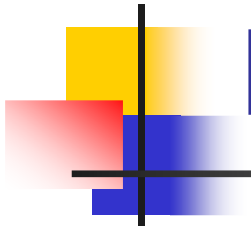
Python – tipi di dato numerici

- **i numeri complessi vengono supportati**; per contrassegnare i numeri immaginari si usa il suffisso "j" o "J". I numeri complessi sono creati con "(real+imagj)", o con la funzione "complex(real, imag)".

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

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

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

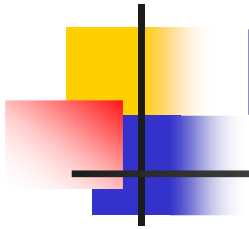


Python – tipi di dato numerici

L'ultima espressione stampata è assegnata alla variabile _ :

```
>>> 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.



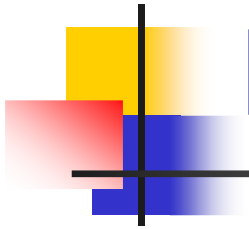
Python – Stringhe

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

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

- produrrà il seguente output:

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



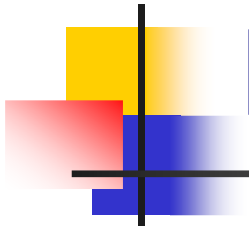
Python – Stringhe

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

```
>>> 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

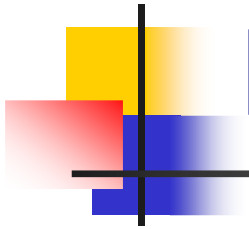
- Immutabilità stringhe:

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

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

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

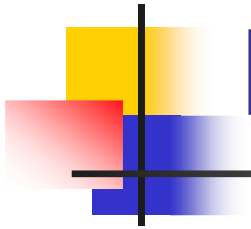
- Abbiamo concatenato la nuova prima lettera ad una porzione di Saluto, e questa operazione non ha avuto alcun effetto sulla stringa originale.
- Numerose sono le funzioni disponibili per le stringhe



Python – Liste

- **La lista è disponibile come tipo predefinito:**

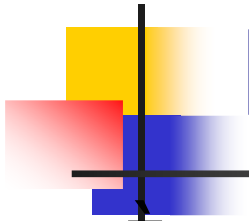
```
>>> a = ['spam', 'eggs', 100, 1234]
>>> a
['spam', 'eggs', 100, 1234]
 >>> a[0]
 'spam'
 >>> a[-2]
 100
 >>> a[1:-1]
 ['eggs', 100]
 >>> a[:2] + ['bacon', 2*2]
 ['spam', 'eggs', 'bacon', 4]
 >>> 3*a[:3] + ['Boe!']
 ['spam', 'eggs', 100, 'spam', 'eggs', 100, 'spam', 'eggs',
    100, 'Boe!']
```



Python – Liste

- Esempi di costruzione di variabili lista:

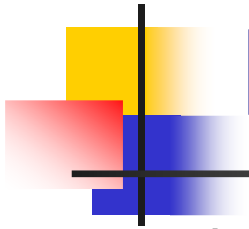
```
>>> # Rimpiazza alcuni elementi:
... a[0:2] = [1, 12]
>>> a
[1, 12, 123, 1234]
>>> # Rimuove alcuni elementi:
 ... a[0:2] = []
>>> a [123, 1234]
>>> # Inserisce alcuni elementi:
... a[1:1] = ['bletch', 'xyzzy']
>>> a
[123, 'bletch', 'xyzzy', 1234]
>>> a[:0] = a # Inserisce (una copia di) se stesso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

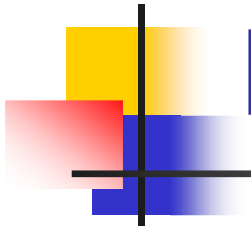
- Anche per le liste sono disponibili metodi (funzioni):

```
>>> 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**:

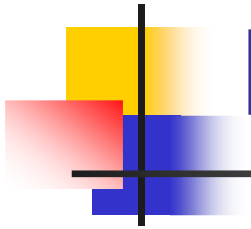
```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```



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']
```



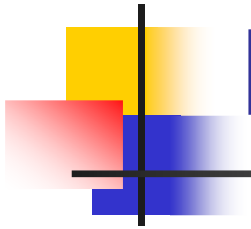
Python – Liste

- 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:

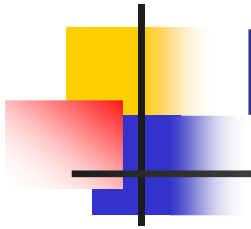
```
>>> def cube(x): return x*x*x
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```



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:

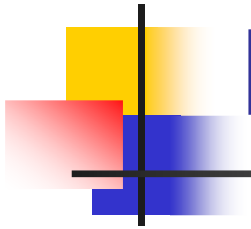
```
>>> def add(x,y): return x+y
>>> reduce(add, range(1, 11))
55
```



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:

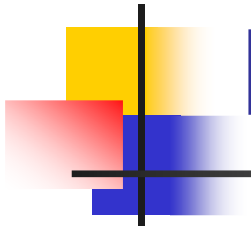
```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Le tuple possono essere annidate:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
```



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:

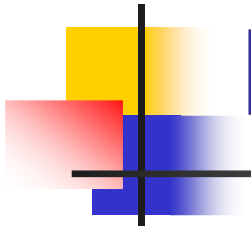
```
>>> empty = ()
>>> singleton = 'hello',
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```



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:

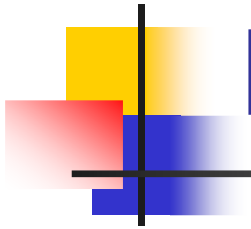
  >>> x, y, z = t
- È chiamata, **unpacking di sequenza**. Lo spacchettamento di sequenza richiede che la lista di variabili a sinistra abbia un numero di elementi pari alla lunghezza della sequenza.
- l'impacchettamento di valori multipli crea sempre una tupla, mentre lo spacchettamento funziona per qualsiasi sequenza.



Python – Insiemi

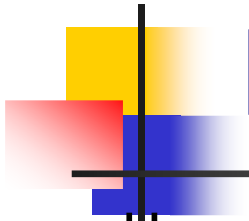
- Python include anche tipi di dati per insiemi (**sets**). Un insieme è una collezione non ordinata che non contiene elementi duplicati al suo interno. Solitamente viene usato per verificare l'appartenenza dei membri ed eliminare gli elementi duplicati.
- Gli oggetti insieme supportano anche le operazioni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



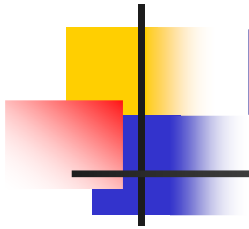
Python – Insiemi

```
>>> 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:

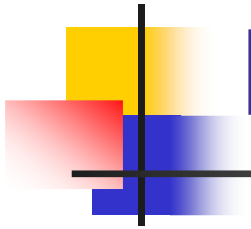
```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True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

Python – Controllo di flusso

- Costrutto **IF**:

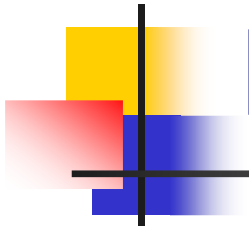
```
>>> x = int(raw_input("Introdurre un numero: "))
>>> if x < 0:
...         x = 0
...         print 'Numero negativo cambiato in zero'
... elif x == 0:
...         print 'Zero'
... elif x == 1:
...         print 'Uno'
... else:
...         print 'Più di uno' ...
```

- 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.

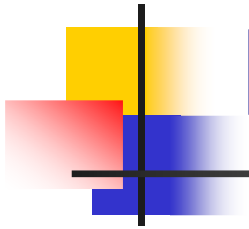


Python – Controllo di flusso

- 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'
```

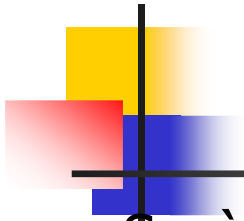
- Si noti che in Python, a differenza che in C, all'interno delle espressioni non può comparire un assegnamento. I programmatori C potrebbero lamentarsi di questa scelta, però essa evita un tipo di problema che è facile incontrare nei programmi C: l'introduzione di = in un'espressione volendo invece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

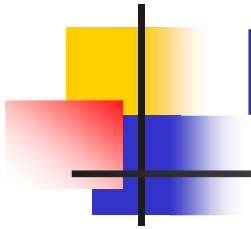


Python – Controllo di flusso

- 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:

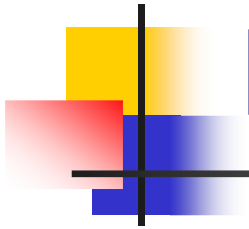
  >>> range(5, 10)
  [5, 6, 7, 8, 9]
  >>> range(0, 10, 3)
  [0, 3, 6, 9]
  >>> range(-10, -100, -30)
  [-10, -40, -70]



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
```



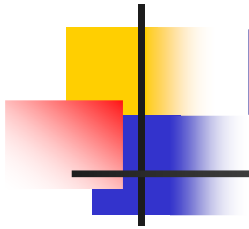
Python – Controllo di flusso

- 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
```



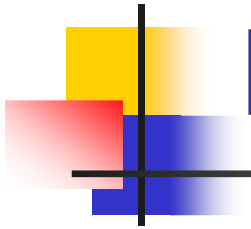
Python – Controllo di flusso

- 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
```

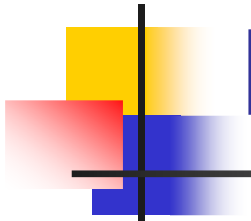


Python – Funzioni

Esempio di definizione di una funzione:

```
def ask_ok(prompt, retries=4, complaint='Sì 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).



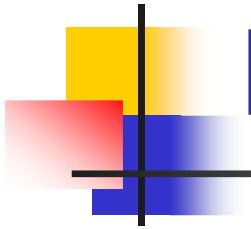
Python – Funzioni

- **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]
```



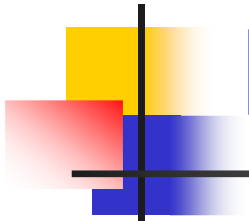
Python – Funzioni

- 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')
parrot() # ERRORE manca un argomento necessario
parrot(voltage=5.0, 'dead')
      # ERR argomento non a parola chiave seguito da una parola chiave
```

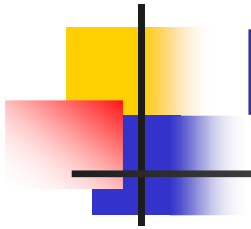


Python – Funzioni

- In Python sono presenti alcune funzionalità dei linguaggi di programmazione funzionale e in Lisp. Con la **parola chiave lambda** possono essere create piccole funzioni senza nome.
- 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:

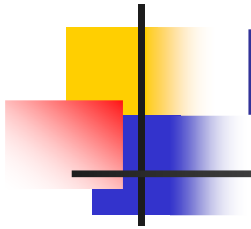
```
>>> def make_incrementor(n):
...   return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(4)
46
>>>f1=make_incrementor(38)
>>>f1(4)
>>> print make_incrementor(13)(7)
20
```

- Le forme lambda permettono la creazione di funzioni personalizzate



Python – Funzioni

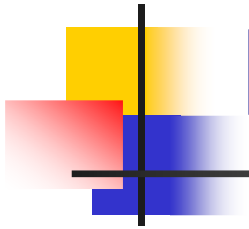
- Il **lambda calcolo** è un sistema di riscrittura definito formalmente dal matematico Alonzo Church. È stato sviluppato per analizzare formalmente le definizioni di funzioni
- La definizione insiemistica di una funzione non fornisce informazioni a proposito di come *effettivamente* si possa passare dagli argomenti ricevuti al risultato ad essi associato. In altre parole, non si descrive come **calcolare** il risultato della funzione stessa. Il lambda calcolo, è invece un formalismo che consente di definire il lato *meccanico* delle funzioni, ovvero proprio quelle procedure che consentono di produrre dei valori in uscita a partire da certi valori in ingresso (per questo è chiamato "calcolo")



Python – Funzioni

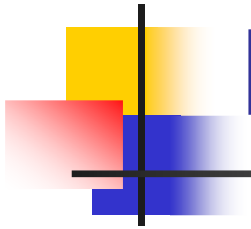
Altri esempi di utilizzo delle funzioni lambda:

- ```
  >>> 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 dogs'.split())*
  [2, 2, 7, 4, 3, 4]



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'.
- 'b' aggiunto al modo apre il file in modo binario, per cui esistono 'rb', 'wb' e 'r+b'.

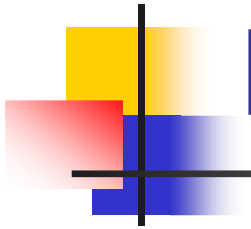
Python – I/O

- **Per leggere i contenuti di un file**, **f.read(lunghezza)** 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.

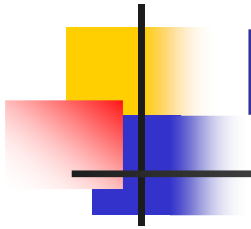
```
>>> f.readlines()
['Questa è la prima riga del file.\n', 'Seconda riga del file\n']
```

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

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

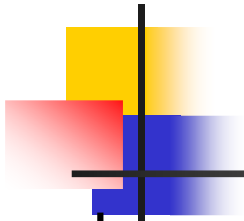
- **f.tell()** restituisce un intero che fornisce la posizione nel file, misurata in byte dall'inizio del file. Per variare la posizione si usa "f.seek(offset, da_cosa)". La posizione viene calcolata aggiungendo ad offset l'argomento da_cosa. Un valore pari a 0 effettua la misura dall'inizio del file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



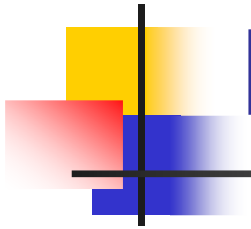
Python – Classi

- 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 (liste, pile, dizionari) sono oggetti**.
- **il termine "oggetto" in Python quindi non significa necessariamente un'istanza di classe.**
- Diversamente da quanto accade in C++ o Modula-3, i tipi built-in non possono essere usati come classi base per estensioni utente.



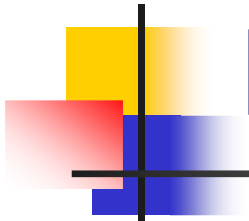
Python – Classi

- 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.



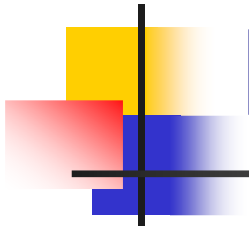
Python – Classi

- 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.



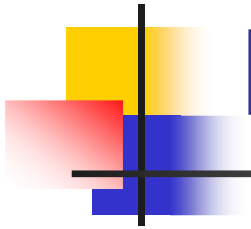
Python – Classi

- 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 definita una 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, che è un involucro (wrapper) per i contenuti dello spazio dei nomi creato definendo la classe**
- **Una classe Python quindi è uno spazio di nomi**



Python – Classi

- **Esempio di creazione di classe**:
  class Punto:
  pass
- Così si crea un nuovo tipo di dato. Per creare un oggetto viene chiamato il **costruttore**: *P1 = Punto()*
- Possiamo aggiungere un dato ad un'istanza usando la notazione punto:
  >>> P1.x = 3.0
- **gli attributi si possono aggiungere o eliminare** (tramite del); le operazioni operano sullo spazio dei nomi della classe, ampliandolo o riducendolo (diversamente da Java). Notare che questi cambiamenti si possono effettuare da una istanza ma hanno effetto su tutte le istanze
- Altro esempio:
  >>> 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". **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.

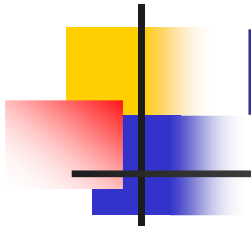


Python – Classi

- Gli **oggetti classe** supportano **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:
    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, quindi **un "attributo" Python comprende attributi e metodi Java**



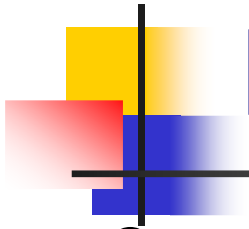
Python – Classi

- **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 istanziazione invoca automaticamente __init__() per l'istanza di classe appena creata.
- Naturalmente il metodo __init__() può avere argomenti, ad esempio:

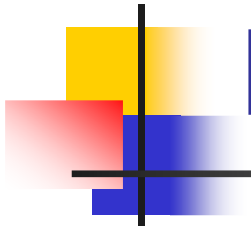
```
>>> class Complesso:
...         def __init__(self, partereale, partimag):
...             self.r = partereale
...             self.i = partimag
...
>>> x = Complesso(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```



Python – Classi

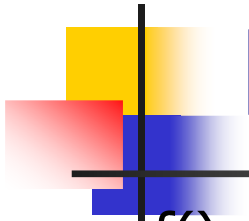
- Ora, cosa possiamo fare con **gli oggetti istanza**? Le sole operazioni che essi conoscono per mezzo dell'istanziazione degli oggetti sono i riferimenti ad attributo. Ci sono **due tipi di nomi di attributo validi: dati e metodi**
- Gli **attributi dato** corrispondono alle "variabili istanza" in Smalltalk, 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
```



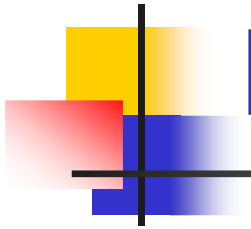
Python – Classi

- 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 hanno 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 (una funzione esiste senza legami con una classe).



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 (self)**. 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 self 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.

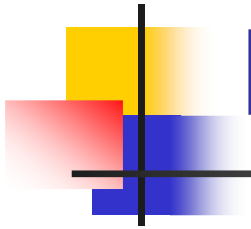


Python – Classi

- 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'
```

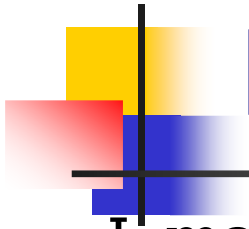
- Ora f e g sono tutti attributi che si riferiscono ad oggetti funzione, di conseguenza sono tutti metodi delle istanze della classe C



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)
```



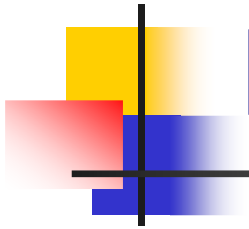
Python – Classi

- 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)
```

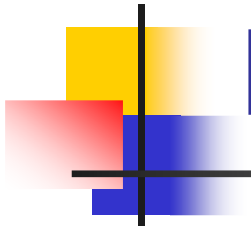


Python – Classi

- 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 più immediati il mantenimento ed il riutilizzo del codice**



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) + ')'
```

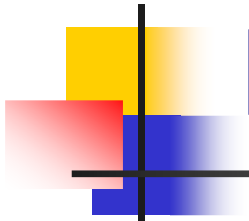
- Il metodo di inizializzazione prende x e y come parametri opzionali. Il loro valore di default è 0.
- Il metodo __str__ ritorna una rappresentazione di un oggetto Punto sotto forma di stringa. Se una classe fornisce un metodo chiamato __str__ questo sovrascrive la funzione str di Python.

```
>>> 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

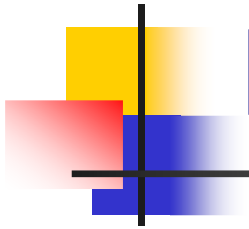
- In Python si può **cambiare la definizione degli operatori predefiniti** quando applicati a tipi definiti dall'utente (**overloading degli operatore**).
- Se vogliamo ridefinire l'operatore somma + scriveremo un metodo chiamato __add__:

```
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.

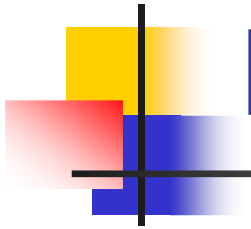


Python – Classi

- 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

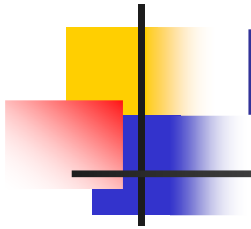


Python – Classi

- 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:

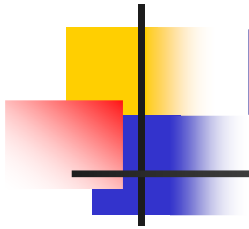
```
def MoltSomma(x, y, z):
    return x * y + z
```

- 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]
```

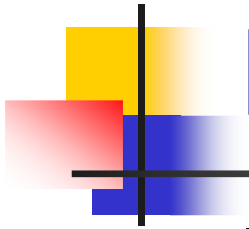
- Era facilmente intuibile che questa funzione riuscisse a maneggiare le liste. Ma può lavorare con oggetti di tipo Punto?

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:

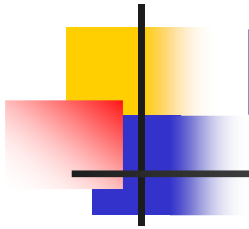
```
def reverse(self):
  self.x , self.y = self.y, self.x
Ora possiamo passare Punto a DirittoERovescio:
>>>   P = Punto(3, 4)
>>>   DirittoERovescio(P)
(3, 4)(4, 3)
```

- **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

- 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.
- 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 **regola semantica necessaria** 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.



Python – Classi

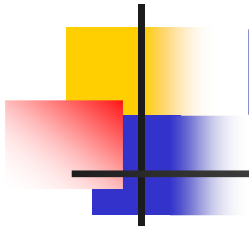
- **classe di esempio: Frazione**

```
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
```

- il comando print invoca il metodo __str__ implicitamente.



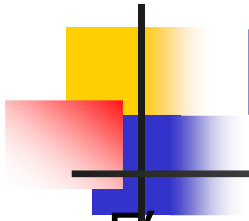
Python – Classi

- 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
```



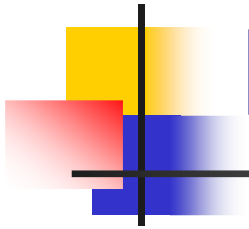
Python – Classi

- 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

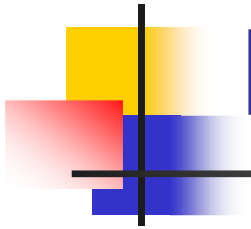
- Per valutare l'operatore di moltiplicazione, Python controlla l'operando di sinistra per vedere se questo fornisce un metodo __mul__ che supporta il tipo del secondo operando. Se il controllo non ha successo Python passa a controllare l'operando di destra per vedere se è stato definito un metodo __rmul__ 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

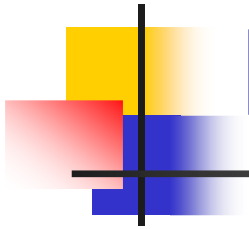
- L'addizione è più complessa della moltiplicazione ma non troppo: la somma di a/b e c/d è infatti la frazione (a*d+c*b)/b*d.
- Usando il codice della moltiplicazione come modello possiamo scrivere __add__ e __radd__:

```
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__.

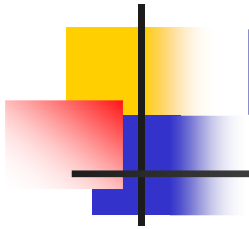


Python – Classi

- 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.
- L'algoritmo di Euclide calcola 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.



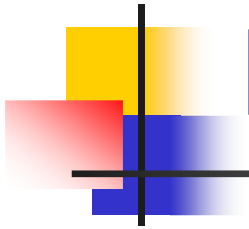
Python – Classi

- 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.