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” quindi non significa necessariamente un'istanza di classe.
- Diversamente da quanto accade in C++ o Modula-3, in Python i tipi built-in non possono essere usati come classi base per estensioni utente.

Classi

- La forma più semplice di definizione di una classe

```
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.
- Una classe Python quindi è uno spazio di nomi

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). 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.

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

Classi

- L'instanziazione 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)
```

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
```

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 oggetti hanno i metodi: ad esempio gli oggetti lista hanno 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).

Classi

```
class MiaClasse:
        i = 12345
        def f(self):
                return 'ciao mondo'

x.f()
```

- 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.

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.

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)
```

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)
```

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

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.

Classi

```
>>> 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)

Classi

- In Python si può cambiare la definizione degli operatori predefiniti quando applicati a tipi definiti dall'utente (overloading degli operatori)
- 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.

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.

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'
```

In generale, le funzioni per gli operatori sono:

__add__ (self, other)
__sub__ (self, other)
__mul__ (self, other)
__div__ (self, other)
__mod__ (self, other)
__divmod__ (self, other)
__pow__ (self, other[, modulo])
__lshift__ (self, other)
__rshift__ (self, other)
__and__ (self, other)
__xor__ (self, other)

__or__ (self, other)
__radd__ (self, other)
__rsub__ (self, other)
__rmul__ (self, other)
__rdiv__ (self, other)
__rmod__ (self, other)
__rdivmod__ (self, other)
__rpow__ (self, other)
__rlshift__ (self, other)
__rrshift__ (self, other)
__rand__ (self, other)
__rxor__ (self, other)
__ror__ (self, other)

Le funzioni con in prefisso ‘r’ (reverse operands) vengono invocate solo se l’operando sinistro non supporta l’operazione

Classi

- La maggior parte dei metodi che abbiamo scritto fin qui operano 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

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.

Ereditarità

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

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

- Il nome NomeClasseBase deve 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.

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.Denominator e)
```

- Per testare il lavoro, lo si salva 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.

classe di esempio: Frazione

- 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
```

classe di esempio: Frazione

- 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 *:  and 'instance: 'int'
```

classe di esempio: Frazione

- 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.

classe di esempio: Frazione

- 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__
```

classe di esempio: Frazione

- 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__.

classe di esempio: Frazione

- 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 di visore tra due numeri. In caso di 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 conuna 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.

classe di esempio: Frazione

- 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
```

classe di esempio: Frazione

- 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.

classe di esempio: Frazione

- 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
```

Eccezioni

- Se il programma si blocca a causa di un errore in esecuzione viene creata un'eccezione: l'interprete si ferma e mostra un messaggio d'errore.

Esempi di eccezioni

1. Divisione per zero

```
>>> print 55/0
ZeroDivisionError: integer division or modulo
```

2. la richiesta di un elemento di una lista con un indice errato:

```
>>> a = []
>>> print a[5]
IndexError: list index out of range
```

3. la richiesta di una chiave non esistente in un dizionario:

```
>>> b = {}
>>> print b['pippo']
KeyError: pippo
```

Eccezioni

- In ogni caso il messaggio d'errore è composto di due parti: il tipo dell'errore e le sua specifica, separati dai due punti. Di norma Python stampa la traccia del programma al momento dell'errore.
- La soluzione per gestire le eccezioni è usare le istruzioni **try** ed **except.**

```
NomeFile = raw_input('Inserisci il nome del file: ')
try:
  f = open (NomeFile, "r")
except:
  print 'Il file', NomeFile, 'non esiste'
```

- L'istruzione `try` esegue le istruzioni nel suo blocco. Se non si verificano eccezioni l'istruzione `except` ed il blocco corrispondente vengono saltate ed il flusso del programma prosegue dalla prima istruzione presente dopo il blocco `except`. Nel caso si verifichino eccezioni viene interrotto immediatamente il flusso del blocco `try` ed eseguito il blocco `except`

Eccezioni

- La funzione `FileEsiste` prende un nome di un file e ritorna vero se il file esiste, falso se non esiste.

```
def FileEsiste(NomeFile):
  try:
            f = open(NomeFile)
            f.close()
            return 1
  except:
            return 0
```

- È possibile utilizzare blocchi di `except` multipli per gestire diversi tipi di eccezioni.

Eccezioni

- Con `try/except` possiamo fare in modo di continuare ad eseguire un programma in caso di errore.
- È possibile sollevare delle eccezioni nel corso del programma con l'istruzione `raise` in modo da generare un errore in esecuzione quando qualche condizione non è verificata:

```
def InputNumero():
  x = raw_input (’Dammi un numero: ')
  if x > 16 :
    raise 'ErroreNumero', 'mi aspetto numeri minori di 17!'
  return x
```

- In questo caso viene generato un errore in esecuzione quando è introdotto un numero maggiore di 16.

Eccezioni

- L'istruzione `raise` prende due argomenti: il tipo di eccezione e l'indicazione specifica del tipo di errore. ErroreNumero è un nuovo tipo di eccezione che è stata introdotta appositamente per questa applicazione.
- Se la funzione che chiama InputNumero gestisce gli errori il programma continua, altrimenti Python stampa il messaggio d'errore e termina l'esecuzione:

```
>>> InputNumero()
Dimmi un numero: 17
ErroreNumero: mi aspetto numeri minori di
  17!
```

- Il messaggio di errore include l'indicazione del tipo di eccezione e l'informazione aggiuntiva che è stata fornita