Laurea in Ingegneria Gestionale

Corso di Fondamenti di Informatica
A.A. 2017/2018

DIPARTIMENTO DI INGEGNERIA INFORMATICA
AUTOMATICA E GESTIONALE ANTONIO RUBERTI

SAPIENZA
UNIVERSITÀ DI ROMA

classi e oggetti

Slide tratte da:

Pensare da informatico
Versione Python
di Allen B. Downey, Jeffrey Elkner e Chris Meyers
Traduzione di Alessandro Pocaterra

Capitoli 12, 13 e 14

Disponibili al link
https://www.python.it/doc/Howtothink/Howtothink-html-it/index.htm

Tipi composti definiti dall'utente

Abbiamo usato alcuni dei tipi composti predefiniti e ora vediamo come definire nuovi tipi composti.

Consideriamo ad esempio il concetto matematico di punto nelle due dimensioni.

Un modo naturale di rappresentare un punto in Python è una coppia di numeri in virgola. Potremmo raggruppare i numeri in una lista, ma sarebbe poco elegante (ad esempio, non sarebbe evidente quale elemento della lista rappresenta la coordinata sull'asse delle ascisse, e quale sull'asse delle ordinate).

Tipi composti definiti dall'utente: La classe `Punto`

Un modo alternativo è quello di definire un nuovo tipo composto attraverso la creazione di una nuova **classe**.

Una definizione di classe ha questa sintassi:

```
class Punto :
  pass
```

la definizione dell'esempio crea una nuova classe chiamata `Punto`.

L'istruzione `pass` non ha effetti: è stata usata per il solo fatto che la definizione prevede un corpo che deve ancora essere scritto.

Costruttore per oggetti di classe `Punto`

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 invocat una funzione chiamata `Punto`:

```
P1 = Punto()
```

Alla variabile `P1` è assegnato il riferimento ad un nuovo oggetto di classe `Punto`. Una funzione come `Punto`, che crea nuovi oggetti e riserva quindi della memoria per depositarne i valori, è detta costruttore.

Attributi per `Punto`

Possiamo aggiungere un nuovo dato ad un'istanza usando la notazione punto:

```
>>> P1.x = 3
>>> P1.y = 4
```

In questo caso stiamo selezionando una "voce" da un'istanza e queste voci che fanno parte dell'istanza sono dette **attributi**.

La variabile `P1` si riferisce ad un oggetto di classe `Punto` che contiene due attributi ed ogni attributo (una coordinata) denota un numero in virgola mobile.

Attributi per `Punto`

Possiamo leggere il valore di un attributo con la stessa sintassi:

```
>>> P1.y
4
>>> x = P1.x
>>> x
3
```

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`. La variabile locale `x` e l'attributo `x` di `P1` sono variabili differenti: la notazione punto consente infatti di identificare la variabile cui ci si riferisce evitando le ambiguità.

Attributi per `Punto`

Si può usare la notazione punto all'interno di una espressione. Ad esempio:

```
print( '(' + str(P1.x) + ',' + str(P1.y) + ')' )
DistanzaAlQuadrato = P1.x * P1.x + P1.y * P1.y
```

La prima riga stampa `(3, 4)`; la seconda calcola il valore `25`.

se si stampa direttamente il valore di `P1` si ottiene un risultato analogo al seguente

```
>>> print(P1)
<__main__.Punto instance at 80f8e70>
```

che 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)

Istanze come parametri

Puoi passare un oggetto come parametro ad una funzione nel solito modo:

```
def StampaPunto(P) :
  print( '(' + str(P.x) + ',' + str(P.y) + ')' )
```

`StampaPunto` prende un oggetto `Punto` come argomento e ne stampa gli attributi in forma standard.

Chiamando `StampaPunto(P1)` la stampa è `(3, 4)`.

Uguaglianza fra oggetti

Si considerino le seguenti istruzioni:

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

L'operatore == verifica se due riferimenti (`P1` e `P2`) "puntano" allo stesso oggetto (anche se `P1` e `P2` hanno le stesse coordinate non fanno riferimento allo stesso oggetto ma a due oggetti diversi).

Se assegniamo `P1` a `P2` allora le due variabili sono *alias* dello stesso oggetto:

```
>>> P2 = P1
>>> P1 == P2
True
```

Uguaglianza fra oggetti

Questo tipo di uguaglianza è detta uguaglianza superficiale (o debole) perché si limita a confrontare solo i riferimenti delle variabili e non il contenuto degli oggetti.

Per confrontare il contenuto degli oggetti, e realizzare una uguaglianza profonda (o forte), si può scrivere una funzione chiamata `StessoPunto`:

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

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

Ovviamente se due oggetti sono uguali in modo superficiale, lo sono in modo profondo

La classe Rettangolo

Definiamo la nuova classe:

```
class Rettangolo:
  pass
```

Per istanziare un nuovo oggetto rettangolo:

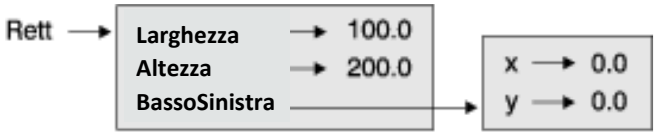
```
Rett = Rettangolo()
Rett.Larghezza = 100
Rett.Altezza = 200
```

Questo codice crea un nuovo oggetto `Rettangolo` con due attributi in virgola mobile. Aggiungiamo il punto di riferimento in basso a sinistra per caratterizzare il posizionamento del rettangolo sul piano cartesiano e per specificarlo inseriamo un oggetto all'interno di un altro oggetto:

```
Rett.BassoSinistra = Punto()
Rett.BassoSinistra.x = 0
Rett.BassoSinistra.y = 0
```

La classe Rettangolo

L'operatore punto è usato per comporre l'espressione: `Rett.BassoSinistra.x` significa "vai all'oggetto cui si riferisce `Rett` e seleziona l'attributo chiamato `BassoSinistra`; poi vai all'oggetto cui si riferisce `BassoSinistra` e seleziona l'attributo chiamato `x`".



Istanze come valori di ritorno

Le funzioni possono ritornare istanze. Possiamo quindi scrivere una funzione `TrovaCentro` che prende un oggetto di tipo `Rettangolo` come argomento e restituisce un oggetto di tipo `Punto` che contiene le coordinate del centro del rettangolo:

```
def TrovaCentro(Rettangolo):
  P = Punto()
  P.x = Rettangolo.BassoSinistra.x \
     + Rettangolo.Larghezza/2
  P.y = Rettangolo.BassoSinistra.y \
     + Rettangolo.Altezza/2
  return P
```

Per chiamare questa funzione passiamo la variabile `Rett` (inizializzata come visto nelle slide precedenti) come argomento e assegniamo il risultato ad una variabile chiamata `Centro`:

```
>>> Centro = TrovaCentro(Rett)
>>> StampaPunto(Centro)
(50.0, 100.0)
```

Gli oggetti sono mutabili

Possiamo cambiare lo stato di un oggetto facendo un'assegnazione ad uno dei suoi attributi. Ad esempio, possiamo cambiare le dimensioni di Rett:

```
Rett.Larghezza = Rett.Larghezza + 50
Rett.Altezza = Rett.Altezza + 100
```

Incapsulando questo codice in un metodo e generalizzandolo diamo la possibilità di aumentare le dimensioni di qualsiasi rettangolo:

```
def AumentaRettangolo(Rett, AumentoLargh, AumentoAlt) :
  Rett.Larghezza = Rett.Larghezza + AumentoLargh
  Rett.Altezza = Rett.Altezza + AumentoAlt
```

Invocare questo metodo ha lo stesso effetto di modificare il rettangolo che è passato come argomento (parametro `Rett`).

Modifiche visibile attraverso il parametro attuale

Creiamo un nuovo rettangolo chiamato `R1` e passiamolo a `AumentaRettangolo`:

```
>>> R1 = Rettangolo()
>>> R1.Larghezza = 100.0
>>> R1.Altezza = 200.0
>>> R1.BassoSinistra = Punto()
>>> R1.BassoSinistra.x = 0.0;
>>> R1.BassoSinistra.y = 0.0;
>>> AumentaRettangolo(R1, 50, 100)
```

Mentre stiamo eseguendo `AumentaRettangolo` il parametro `Rett` è un alias per `R1`. Ogni cambiamento apportato `Rett` modifica direttamente `R1` e viceversa.

Gli oggetti sono mutabili

Questo comportamento (la modifica dell'oggetto passato come parametro al momento dell'invocazione) ovviamente non cambia anche se la funzione restituisce un rettangolo

```
def AumentaRettangolo(Rett, AumentoLargh, AumentoAlt) :
  Rett.Larghezza = Rett.Larghezza + AumentoLargh
  Rett.Altezza = Rett.Altezza + AumentoAlt
  Rett.BassoSinistro = Rett.BassoSinistro
  return Rett
```

Invocando questa funzione con il comando

```
>>> R2 = AumentaRettangolo(R1, 50, 100)
```

Abbiamo che `R1` , `Rett` e quindi anche `R2` si riferiscono ("puntano") allo stesso oggetto (sono alias), per cui le modifiche al contenuto dell'oggetto fatte tramite uno di questi alias si ritrovano anche accedendo all'oggetto con altri alias, in particolare con `R1` ed `R2`, che sono gli unici alias disponibili al termine dell'esecuzione della funzione `AumentaRettangolo`

Copia

La copia di un oggetto è spesso una comoda alternativa all'alias, in particolare se **non** vogliamo che le modifiche fatte ad un oggetto riferendoci ad esso attraverso un certo alias siano "visibili" anche accedendo all'oggetto usando un altro alias.

Il modulo `copy` contiene una funzione `copy` che permette di duplicare qualsiasi oggetto:

```
>>> from copy import copy
>>> P1 = Punto()
>>> P1.x = 3
>>> P1.y = 4
>>> P2 = copy(P1)
>>> P1 == P2
False
>>> StessoPunto(P1, P2)
True
```

Copia

In pratica, il comando

```
>>> P2 = copy(P1)
```

Equivale ai comandi

```
P2 = Punto()
P2.x = P1.x
P2.y = P2.y
```

Copia

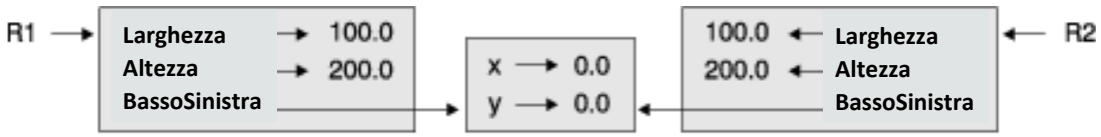
Per copiare un semplice oggetto come `Punto` che non contiene altri oggetti al proprio interno `copy` è sufficiente. Questa è chiamata copia debole:

```
>>> Punto2 = copy(Punto1)
```

Quando abbiamo a che fare con `un Rettangolo` che contiene al proprio interno un riferimento ad un altro oggetto `Punto`, con la funzione `copy` **viene copiato il riferimento a** `Punto` così che sia il vecchio che il nuovo `Rettangolo` si riferiscono allo stesso oggetto invece di averne uno proprio per ciascuno.

Copia

Se creiamo il rettangolo `R1` nel solito modo e ne facciamo una copia `R2` usando `copy`, otteniamo la seguente situazione in memoria:



Quasi certamente non è questo ciò che vogliamo. In questo caso, invocando `AumentaRettangolo` su uno dei rettangoli non si cambieranno le dimensioni dell'altro, ma `MuoviRettangolo` sposterà entrambi! Questo comportamento genera parecchia confusione e porta facilmente a commettere errori.

Copia "profonda"

Fortunatamente il modulo `copy` contiene un altro metodo chiamato `deepcopy` che copia correttamente non solo l'oggetto ma anche gli eventuali oggetti presenti al suo interno:

```
>>> from copy import deepcopy
>>> Oggetto2 = deepcopy(Oggetto1)
```

Ora `Oggetto1` e `Oggetto2` sono oggetti completamente separati e occupano diverse zone di memoria.