
PYTHON: Programmazione ad Oggetti

Release 0.1

Enrico Morelli

19 novembre 2004

Email: gentoo-dev@gentoo.it

Copyright © 2003 Gentoo.It(gentoo-dev@gentoo.it) All rights reserved.

See the end of this document for complete license and permissions information.

Sommario

Ben pochi linguaggi hanno le caratteristiche del Python. Questo è dovuto primariamente al fatto che Python è stato progettato come un linguaggio orientato agli oggetti. Per questo motivo fornisce molti costrutti che semplificano la stesura del codice e la sua riusabilità. Python riconosce automaticamente gli oggetti come stringhe, numeri e liste e non necessita di dichiarare tipi e dimensione delle variabili come in C o Java. Completano il quadro la gestione automatica della memoria ed una vasta gamma di librerie e un alto livello di astrazione.

Questo breve articolo cercherà di introdurre il lettore alla comprensione della programmazione ad oggetti nel modo più semplice possibile coprendo argomenti come classi, oggetti, attributi e metodi con uno sguardo ad un paio di concetti avanzati quali costruttori, distruttori ed ereditarietà.

INDICE

| | | |
|----------|---------------------------------------|-----------|
| 1 | Le basi: nozioni e definizioni | 1 |
| 1.1 | Definizione di una classe | 1 |
| 1.2 | Approfondimento | 2 |
| 1.3 | Definizione della classe | 3 |
| 1.4 | Self | 4 |
| 2 | Costruttori | 5 |
| 2.1 | Esempio pratico | 7 |
| 3 | Ereditarietà | 11 |
| 3.1 | Introduzione | 11 |
| 3.2 | Definizione | 11 |
| 3.3 | Utilizzo | 11 |
| 3.4 | In profondità | 14 |
| 4 | Distruttori | 17 |
| 5 | Conclusioni | 19 |
| 6 | History e Licenza | 21 |
| 6.1 | License | 21 |

Le basi: nozioni e definizioni

Prima di iniziare è necessario avere familiarità con alcuni termini. In Python una *classe* è semplicemente una serie di dichiarazioni che eseguono compiti specifici. Una tipica dichiarazione di classe contiene sia variabili che funzioni e serve come il modello da cui generare istanze specifiche di questa classe.

Ci si riferisce a tali istanze specifiche come agli *oggetti* della classe. Ogni oggetto ha determinate caratteristiche o "proprietà" e certe funzioni o *metodi* predefiniti. Queste proprietà e metodi dell'oggetto corrispondono direttamente a variabili e a funzioni interne alla definizione della classe.

Una volta che è stata definita una classe, Python permette di generare tutte le istanze della classe si desiderino. Ognuna di queste istanze è un oggetto completamente indipendente con proprietà e metodi propri e può così essere manipolato indipendentemente dagli altri oggetti. Questa caratteristica diviene utile quando si necessita di generare più di una istanza di un oggetto, ad esempio dovendo aprire due connessioni simultanee ad un database per eseguire due query simultanee.

Le classi sono d'aiuto anche nel mantenere il codice modulare potendo definire classi in file separati ed includendo questi file solo dove si prevede l'uso della classe. Facilitano anche le modifiche apportate al codice, dato che si dovrà editare solo un singolo file per aggiungere nuove funzionalità a tutti gli oggetti generati.

1.1 Definizione di una classe

La definizione base di una classe è la seguente:

```
class Computer:
    # Alcuni utili metodi
    def nome(self):
        # il codice inizia qui
    def descrizione(self):
        # il codice inizia qui
    def so(self):
        # il codice inizia qui
```

Una volta definita la classe, si può istanziare un nuovo oggetto assegnandolo ad una variabile nel seguente modo:

```
Python 2.3.4 (#1, Oct 19 2004, 14:50:10)
[GCC 3.3.4 20040623 (Gentoo Linux 3.3.4-r1, ssp-3.3.2-2, pie-8.7.6)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> pc = Computer()
>>>
```

Attraverso la variabile (*pc* nel caso in esempio) è possibile accedere ai metodi e alle proprietà dell'oggetto.

```
>>> pc.nome()
>>>
>>> pc.descrizione()
>>>
```

Le dichiarazioni mostrate nell'ultimo esempio, mostrano come ogni istanza di una classe sia indipendente dalle altre, il che significa che si potrebbero creare più di un'istanza di `Computer()` e chiamare i metodi di ogni istanza indipendentemente.

```
>>> pc1 = Computer()
>>> pc2 = Computer()
>>>
>>> pc1.nome()
>>>
>>> pc2.nome()
>>>
>>> pc2.descrizione()
>>>
```

1.2 Approfondimento

Si aggiungeranno alcune nuove proprietà modificando la definizione della classe per supportare alcune caratteristiche aggiuntive.

```
class Computer:
    # Alcuni utili metodi
    def nome(self):
        # il codice inizia qui
    def descrizione(self):
        # il codice inizia qui
    def so(self):
        # il codice inizia qui
    def imposta_nome(self, nome):
        # il codice inizia qui
        self.nome = nome
```

La definizione della classe contiene ora un metodo in più, `imposta_nome()`, che modifica il valore della proprietà *"nome"*. Il prossimo esempio mostra l'uso di questo nuovo metodo.

```

>>> pc1 = Computer()
>>> pc2 = Computer()
>>>
>>> # nome del pc1
>>> pc1.imposta_nome("Gentoo")
>>> pc1.nome
'Gentoo'
>>>
>>> # nome del pc2
>>> pc2.imposta_nome("Linus")
>>> pc2.nome
'Linus'
>>>
>>> # cambiamo nome a pc1
>>> pc1.imposta_nome("Pippo")
>>> pc1.nome
'Pippo'
>>>

```

Come illustrato nell'esempio, una volta che vengono definiti nuovi oggetti, si può accedere ai loro metodi individuali e alle loro proprietà in modo indipendente le une dalle altre.

E' importante notare il modo in cui si accede ai metodi e alle proprietà degli oggetti, prefissando il nome del metodo o della proprietà col nome dell'istanza specifica dell'oggetto.

1.3 Definizione della classe

Prendiamo in esame la definizione della classe.

```

class Computer:
    # definizione dei metodi e delle proprietà

```

Ogni definizione di classe inizia con la keyword "class" seguita dal nome della classe. Si può dare alla classe qualsiasi nome che non collida con una parola riservata e di qualsiasi lunghezza. Tutte le variabili ed i metodi della classe sono indentati all'interno del blocco della definizione.

Per creare una nuova istanza di una classe, occorre semplicemente creare una nuova variabile che referenzi la classe.

```

>>> pc1 = Computer()
>>>

```

Il significato in una frase potrebbe essere "si crei un nuovo oggetto della classe *Computer* e lo si assegna alla variabile *pc1*".

Si può quindi accedere a tutti i metodi e a tutte le proprietà della classe attraverso questa variabile.

```

>>> # accedere ad un metodo
>>> pc1.imposta_nome("Gentoo")
>>>
>>> # accedere ad una proprietà
>>> pc1.nome
>>>

```

Di nuovo, se si volesse descrivere l'esempio precedente in una frase, si potrebbe dire "si esegua il metodo `imposta_nome()` col parametro "Gentoo" di questa istanza specifica della classe `Computer`".

1.4 Self

Si sarà notato come nei paragrafi precedenti si sia fatto uso di una parola non ancora conosciuta chiamata `self`.

La variabile `self` ha molto a che fare col funzionamento dei metodi di una classe in Python. Quando un metodo di una classe viene "chiamato" si richiede il passaggio di una referenza all'istanza che è stata chiamata. Questa referenza viene passata come primo argomento del metodo ed è rappresentata dalla variabile `self`.

```

class Computer:
    # alcuni utili metodi
    def imposta_nome(self, nome):
        self.nome = nome

```

L'esempio precedente, mostra l'uso della variabile `self`. Quando un'istanza della classe chiama questo metodo, una referenza all'istanza è passata automaticamente al metodo insieme al parametro aggiuntivo `nome`. Questa referenza è usata quindi per aggiornare la variabile `nome` che appartiene a questa specifica istanza della classe.

```

>>> pc1 = Computer()
>>> pc1.imposta_nome("Gentoo")
>>> pc1.nome
'Gentoo'
>>>

```

Da notare che quando si chiama un metodo di una classe da una istanza, non c'è bisogno di passare esplicitamente questa referenza al metodo, Python fa questo automaticamente.

Con questo in mente, consideriamo il seguente esempio.

```

>>> pc1 = Computer()
>>> Computer.imposta_nome(pc1, "Gentoo")
>>> pc1.nome
'Gentoo'
>>>

```

In questo esempio si è chiamato il metodo della classe direttamente (non attraverso una istanza) passandogli una referenza all'istanza della chiamata al metodo, che lo rende equivalente alla prima istruzione dell'esempio.

Costruttori

E' possibile eseguire automaticamente una funzione quando la classe viene chiamata per creare un nuovo oggetto. In gergo si fa riferimento a questa caratteristica come ad un "costruttore" e per usarlo, occorre che la definizione di classe contenga un metodo chiamato `__init__()`.

Tipicamente, il costruttore `__init__` viene usato per inizializzare variabili o eseguire metodi della classe al momento della prima creazione dell'oggetto. In armonia con quanto detto, si facciano le seguenti modifiche alla classe `Computer`.

```
class Computer:
    # costruttore
    def __init__(self):
        # inizializzazione proprietà
        self.nome = "Gentoo"
        self.tipo = "Pentium 4"
        print "Il nuovo computer Linux"

    # funzione per assegnare un nome
    def imposta_nome(self, nome):
        self.nome = nome

    # funzione per definire il processore
    def imposta_cpu(self, tipo):
        self.tipo = tipo

    # funzione per visualizzare nome e tipo
    def stampa(self):
        print "Il nome del computer è " + self.nome + " e la sua CPU è un " + self.tipo
```

Nell'esempio il costruttore imposta due variabili con valori predefiniti e stampa un breve messaggio indicando che è stato creato un nuovo oggetto.

Le funzioni `imposta_nome()` e `imposta_cpu()` alterano le proprietà dell'oggetto, mentre la funzione `stampa()` provvede un modo semplice per vedere i valori correnti delle proprietà dell'oggetto.

```

>>> pc1 = Computer()
Il nuovo computer Linux
>>> pc1.stampa()
Il nome del computer è Gentoo e la sua CPU è un Pentium 4
>>> pc1.imposta_nome("Pippo")
>>> pc1.imposta_tipo("Amd 64")
>>> pc1.stampa()
Il nome del computer è Pippo e la sua CPU è un Amd 64
>>>

```

Dato che i metodi delle classi funzionano esattamente come funzioni regolari , si può configurare il costruttore per accettare argomenti ed anche specifici valori predefiniti per tali argomenti. Questo rende possibile semplificare la definizione della classe.

```

class Computer:
    # costruttore
    def __init__(self, nome="Gentoo", tipo="Pentium 4"):
        # inizializzazione proprietà
        self.nome = nome
        self.tipo = tipo
        print "Il nuovo computer Linux"

    # funzione per assegnare un nome
    def imposta_nome(self, nome):
        self.nome = nome

    # funzione per definire il processore
    def imposta_cpu(self, tipo):
        self.tipo = tipo

    # funzione per visualizzare nome e tipo
    def stampa(self):
        print "Il nome del computer è " + self.nome + " e la sua CPU è un " + self.tipo

```

L'uso della nuova definizione differisce leggermente dall'esempio precedente.

```

>>> pc1 = Computer("Pippo", "AMD 64")
Il nuovo computer Linux
>>> pc2 = Computer()
Il nuovo computer Linux
>>>
>>> # stampare le informazioni
>>> pc1.stampa()
Il nome del computer è Pippo e la sua CPU è un AMD 64
>>> # si noti che pc2 è stato creato con i valori predefiniti delle proprietà
>>> pc2.stampa()
Il nome del computer è Gentoo e la sua CPU è un Pentium 4
>>> pc1.nome
'Pippo'
>>> pc1.tipo
'AMD 64'
>>> pc2.nome
'Gentoo'
>>> pc2.tipo
'Pentium 4'
>>>

```

Come si dovrebbe aver notato, è anche possibile assegnare valori direttamente alle proprietà delle istanze bypassando i metodi preposti della classe. Per esempio la linea

```

>>> pc2.imposta_nome("Paperino")
>>>

```

è tecnicamente equivalente a

```

>>> pc2.nome="Paperino"
>>>

```

Si noti che è "tecnicamente" possibile, ma in pratica non è raccomandabile usare questo metodo in quanto potrebbe violare l'integrità dell'oggetto. La tecnica preferita è sempre usare i metodi esposti dall'oggetto per modificare le proprietà dell'oggetto stesso.

Per comprendere appieno questo punto, considerate cosa succederebbe se l'autore della classe `Computer()` decidesse di cambiare la variabile "nome" in "nome_pc". Si dovrebbe rimanipolare lo script di test dato che si era referenziata direttamente la variabile "nome". In altre parole, se si fosse scelto di usare il metodo esposto `imposta_nome()`, la modifica alla variabile "nome" avrebbe coinvolto il solo metodo `imposta_nome()` che l'autore stesso della classe avrebbe modificato per riflettere la nuova variabile, mentre il nostro codice non avrebbe dovuto subire alcuna modifica.

Se ci si limita ai metodi esposti le modifiche alle classi non avranno ripercussioni sul nostro codice.

2.1 Esempio pratico

Una volta compresi i principi fondamentali spostiamo l'attenzione verso qualcosa di più pratico. Il prossimo esempio permette di creare differenti oggetti `Clock` inizializzandoli ad una specifica time zone. Si dovrà creare un clock per visualizzare l'ora locale, un altro per visualizzare l'ora di New York ed un terzo per l'ora di Londra, tutto attraverso la potenza degli oggetti Python.

```

# Una semplice classe clock
# ogni oggetto Clock è inizializzato con l'offset (ora e minuti)
# indicanti la differenza tra l'ora GMT e l'ora locale

class Clock:
    # costruttore
    def __init__(self, offsetSign, offsetH, offsetM, city):
        # configurazione delle variabili per l'offset timezone
        # da GMT, in ore e minuti, e il nome della città
        self.offsetSign = offsetSign
        self.offsetH = offsetH
        self.offsetM = offsetM
        self.city = city

        # stampa un messaggio
        print "Clock creato"

    # metodo per visualizzare l'ora corrente dato l'offset
    def display(self):
        # si usa la funzione gmtime() per convertire l'ora locale in GMT
        # vengono importati i metodi richiesti dal modulo time
        from time import time, gmtime

        self.GMTTime = gmtime(time())

        self.seconds = self.GMTTime[5]
        self.minutes = self.GMTTime[4]
        self.hours = self.GMTTime[3]

        # calcolo dell'ora
        if (self.offsetSign == '+'):
            # l'ora della città è avanti rispetto alla GMT
            self.minutes = self.minutes + self.offsetM
            if (self.minutes > 60):
                self.minutes = self.minutes - 60
                self.hours = self.hours + 1
            self.hours = self.hours + self.offsetH
            if (self.hours >= 24):
                self.hours = self.hours - 24
        else:
            # l'ora della città è indietro rispetto alla GMT
            self.seconds = 60 - self.seconds
            self.minutes = self.minutes - self.offsetM

            if (self.minutes < 0):
                self.minutes = self.minutes + 60
                self.hours = self.hours - 1

            self.hours = self.hours - self.offsetH

            if (self.hours < 0):
                self.hours = 24 + self.hours

        # visualizzare il risultato
        self.localTime = str(self.hours) + ":" + str(self.minutes) + ":" + str(self.seconds)
        print "A " + self.city + " sono attualmente le " + self.localTime

```

Come si nota dall'esempio, il costruttore della classe inizializza un oggetto con quattro variabili: il nome della città, un indicatore per controllare se l'ora della città è avanti o indietro rispetto alla GMT, e la differenza tra l'ora locale in questa città e la standard GMT in ore e minuti.

Una volta che questi attributi sono stati assegnati, il metodo `display()` esegue alcuni semplici calcoli per ottenere l'ora locale della città.

Ecco il risultato.

```
>>> londra = Clock("+", 0, 00, "Londra")
Clock creato
>>> londra.display()
A Londra sono attualmente le 8:52:21
>>> bombay = Clock("+", 5, 30, "Bombay")
Clock creato
>>> bombay.display()
A Bombay sono attualmente le 14:23:5
>>> us_ct = Clock("-", 6, 00, "USA/Centro")
Clock creato
>>> us_ct.display()
A USA/Centro sono attualmente le 2:53:11
>>>
```

Si conoscono ora le basi del paradigma della programmazione ad oggetti sotto Python e si dovrebbe essere in grado di costruire ed usare oggetti propri.

Ereditarietà

3.1 Introduzione

Una delle caratteristiche maggiormente utili nella programmazione orientata agli oggetti è la possibilità di riusare oggetti esistenti ed aggiungere nuove caratteristiche a questi oggetti. Questa caratteristica è chiamata "ereditarietà". Creando un nuovo oggetto che eredita le proprietà e i metodi di un oggetto esistente, anche da classi Python, riduce il tempo di stesura del codice e di test.

3.2 Definizione

Python permette di derivare una nuova classe da una classe esistente specificando il nome della classe base tra parentesi al momento della definizione della nuova classe. Così se si vuole derivare una nuova classe chiamata `Portatili()` derivata dalla classe base `Computer()`, la definizione della classe dovrebbe assomigliare all'esempio che segue.

```
class Portatili(Computer):  
    # seguono i metodi e le definizioni della proprietà
```

E' possibile ereditare anche da più di una classe base.

```
class Portatili(Computer, Monitor, Stampanti):  
    # seguono i metodi e le definizioni della proprietà
```

3.3 Utilizzo

L'uso di una classe derivata è pressoché uguale a quello di una normale classe, eccetto per un piccolo particolare: nel caso in cui un oggetto acceda ad un metodo o ad una proprietà che non viene trovato nella classe derivata, Python cercherà automaticamente nella classe base (e nei "progenitori" della classe base) il metodo o la proprietà richiesta.

Nel prossimo esempio si creerà una nuova classe `Portatili()` che viene ereditata dalla classe base `Computer()`.

```

class Computer:
    # costruttore
    def __init__(self, nome="Gentoo", tipo="Pentium 4"):
        # inizializzazione proprietà
        self.nome = nome
        self.tipo = tipo
        print "Il nuovo computer Linux"

    # funzione per assegnare un nome
    def imposta_nome(self, nome):
        self.nome = nome

    # funzione per definire il processore
    def imposta_cpu(self, tipo):
        self.tipo = tipo

    # funzione per visualizzare nome e tipo
    def stampa(self):
        print "Il nome del computer è " + self.nome + " e la sua CPU è un " + self.tipo

class Portatili(Computer):
    pass

```

A questo punto si dovrebbe essere in grado di poter eseguire il prossimo esempio.

```

>>> pc1 = Portatili()
Il nuovo computer Linux
>>> pc1.stampa()
Il nome del computer è Gentoo e la sua CPU è un Pentium 4
>>> pc1.imposta_nome('Paperino')
>>> pc1.imposta_tipo('AMD 64')
>>> pc1.stampa()
Il nome del computer è Paperino e la sua CPU è un AMD 64
>>>

```

Come si è potuto notare il codice funziona esattamente come gli esempi precedenti eccetto per il fatto che si è usata la nuova classe `Portatili()`. Questo indica che la classe `Portatili()` ha ereditato con successo le proprietà ed i metodi della classe base `Computer()`.

Ci si riferisce alla dichiarazione della classe derivata `Portatili()` come ad una *"empty sub-class test"* ("test di una sottoclasse vuota"). Essenzialmente è una nuova classe che funziona esattamente come la classe parente e può essere usata al suo posto.

Si noti inoltre che una classe derivata eredita automaticamente il costruttore della classe base se non ne possiede uno proprio. Comunque, se si definisce un costruttore per la classe derivata, questo sovrascrive il costruttore della classe base.

```

class Portatili(Computer):
    # costruttore
    # accetta nome, tipo, memoria
    def __init__(self, nome="Pippo", tipo="Pentium 4 M", memoria="128MB"):
        self.nome = nome
        self.tipo = tipo
        self.memoria = memoria
        print "Ecco il nuovo portatile"

```

Si noti dal seguente esempio, cosa succede creando una nuova istanza della classe.

```

>>> p1 = Portatili()
Ecco il nuovo portatile
>>> p1.nome
'Pippo'
>>> p1.memoria
'128MB'
>>> p1.stampa()
Il nome del computer è Pippo e la sua CPU è un Pentium 4 M

```

Questo è vero anche per altri metodi. Il prossimo esempio mostra cosa succede se si crea un nuovo metodo *stampa()* nella classe *Portatili()*.

```

class Portatili(Computer):
    # costruttore
    # accetta nome, tipo, memoria
    def __init__(self, nome="Pippo", tipo="Pentium 4 M", memoria="128MB"):
        self.nome = nome
        self.tipo = tipo
        self.memoria = memoria
        print "Ecco il nuovo portatile"
    def stampa(self):
        print "Il nome del nuovo portatile è " + self.nome + " e la sua CPU è un " + self.tipo

```

Ecco il suo funzionamento:

```

>>> p1 = Portatili()
Ecco il nuovo portatile
>>> p1.stampa()
Il nome del nuovo portatile è Pippo e la sua CPU è un Pentium 4 M
>>>

```

Si utilizzerà ora la classe *Clock()* vista in precedenza, per creare una nuova classe *AlarmClock()* da lei derivata.

```

# una classe derivata da clock
# ogni oggetto AlarmClock viene inizializzato con gli offset (hours, minutes)
# che indicano la differenza tra l'ora locale e quella GMT.

class AlarmClock(Clock):
    pass

```

Verifichiamo che questa nuova classe erediti tutti i metodi e le proprietà della classe base.

```

>>> londra = AlarmClock("+", 0, 00, "Londra")
Clock creato
>>> londra.display()
A Londra sono attualmente le 8:52:21
>>>

```

Si aggiungerà ora un nuovo metodo alla classe derivata.

```

class AlarmClock(Clock):

    # azzerare clock per visualizzare GMT
    def reset_to_gmt(self):
        self.offsetSign = "+"
        self.offsetH = 0
        self.offsetM = 0
        self.city = "Londra"
        print "Clock azzerato all GMT!"

```

Ecco il risultato.

```

>>> bombay = AlarmClock("+", 5, 30, "Bombay")
Clock creato
>>> bombay.display()
A Bombay sono attualmente le 16:45:32
>>> bombay.reset_to_gmt()
Clock azzerato alla GMT!
>>> bombay.display()
A Londra sono attualmente le 11:15:39
>>>

```

Si è visto, quindi, come AlarmClock() erediti i metodi dalla classe base Clock() mentre al contempo abbia metodi propri.

3.4 In profondità

Esistono diverse funzioni interne al Python per navigare tra classi e oggetti. Una di queste, la funzione *type()*, permette di distinguere tra classi e istanze.

```

>>> type(Computer)
<type 'classobj'>
>>> pc1 = Computer()
Il nuovo computer Linux
>>> type(pc1)
<type 'instance'>
>>>

```

Potreste avere già familiarità con la funzione *dir*, che restituisce una lista delle proprietà e dei metodi di un oggetto. Vediamo *dir()* alle prese con una classe.

```

>>> dir(Computer)
['__del__', '__doc__', '__init__', '__module__', 'imposta_nome',
'imposta_cpu', 'stampa']
>>>

```

Ed anche con un oggetto della classe.

```

>>> pc1 = Computer("Pippo", "Athlon 64")
Il nuovo computer Linux
>>> dir(pc1)
['__doc__',
'__init__',
'__module__',
'imposta_cpu',
'imposta_nome',
'nome',
'stampa',
'tipo']

>>>

```

Ogni classe dispone della proprietà *__bases__* che mantiene il nome(i) della classe(i) da cui questa particolare classe è derivata. Per la maggior parte del tempo questa proprietà non contiene valori, la sua utilità diviene tale se si lavora con classi che ereditano metodi e proprietà da altre classi.

```

>>> # classe base - nessun antenato
>>> Computer.__bases__
()
>>> # classe derivata - con classe base
>>> Portatili.__bases__
(<class __main__.Computer at 0x4040929c>,)
>>>

```

Se si vogliono vedere i valori delle proprietà di una specifica istanza, si può usare la proprietà *__dict__* dell'istanza, che restituisce un dizionario con la coppia nome-valore.

```

>>> pc1=Computer("Pippo", "Athlon 62")
Il nuovo computer Linux
>>> pc1.__dict__
{'tipo': 'Athlon 64', 'nome': 'Pippo'}
>>>

```

Mentre la corrispondente proprietà `__class__` identifica la classe dalla quale è stata creata l'istanza.

```
>>> pc1.__class__
<__main__.Computer instance at 0x4040f12c>
>>>
```

Distruttori

In Python, un oggetto viene automaticamente distrutto ogni volta che tutti i suoi riferimenti non vengono più usati o quando lo script completa la propria esecuzione. Un distruttore è una funzione speciale che permette l'esecuzione di comandi immediatamente prima della distruzione dell'oggetto.

Non c'è necessità, di solito, di definire un distruttore, ma se si vuole vedere come si potrebbe fare, eccone l'esempio.

```
class Computer:
    # costruttore
    def __init__(self, nome="Gentoo", tipo="Pentium 4"):
        # inizializzazione proprietà
        self.nome = nome
        self.tipo = tipo
        print "Il nuovo computer Linux"

    # funzione per assegnare un nome
    def imposta_nome(self, nome):
        self.nome = nome

    # funzione per definire il processore
    def imposta_cpu(self, tipo):
        self.tipo = tipo

    # funzione per visualizzare nome e tipo
    def stampa(self):
        print "Il nome del computer è " + self.nome + " e la sua CPU è un " + self.tipo
    # distruttore
    def __del__(self):
        print "Distruzione dell'oggetto chiamato " + self.nome + "!"
```

Da notare che il distruttore deve sempre essere chiamato `__del__()`.

```
>>> pc1 = Computer("Pippo", "Athlon 64")
Il nuovo computer Linux
>>> pc2 = Computer("Paperino", "Centrino")
Il nuovo computer Linux
>>>del pc2
Distruzione dell'oggetto chiamato Paperino!
>>> del pc1
Distruzione dell'oggetto chiamato Pippo!
>>>
```

Conclusioni

Questo breve articolo ha trattato le basi della programmazione ad oggetti in Python. Non è sicuramente esauriente, e si rimanda il lettore ad approfondire l'argomento sul sito ufficiale di Python (www.python.org/doc/current/tut/node11.html), su Python Cookbook ([aspn.activestate.com/ASPN/Cookbook\(Python\)](http://aspn.activestate.com/ASPN/Cookbook(Python))) e sui vari HowTo e FAQ sull'argomento.

History e Licenza

6.1 License

Gentoo.It articles are released under the GPL license.