

Corso di programmazione in Python

Lezione 2

Neapolis Hacklab

`hacklab@officina99.org`

Le classi in Python

Le classi in Python si definiscono tramite il comando `class` seguito dal nome della classe e tra parentesi tonde la classe da cui deve discendere. Ad esempio:

```
1 class NuovaClasse(object):  
2     pass
```

In questo caso viene creata una classe chiamata `NuovaClasse` che discende dalla classe `object`. `object` è la classe di più basso livello da cui ogni oggetto discende.

Il corpo della classe in questo esempio non è stato dichiarato sostituendolo con il comando `pass`. La classe erediterà tutte le attributi e i metodi di `object` senza definirne ulteriori.

Le classi in Python

```
1 class NuovaClasse(object):  
2     pass
```

È stata definita in questo modo una nuova classe o meglio un nuovo tipo di dati: il tipo `NuovaClasse`.

Ciò viene confermato dal fatto che il tipo di `NuovaClasse` restituito dal comando `type` è `<type 'type'>`.

Definito il nuovo tipo possiamo ora creare variabili di tipo `NuovaClasse`: `nc = NuovaClasse()`.

Assegnazione attributi

Le classi in Python sono dinamiche, ovvero possono cambiare la loro struttura durante l'esecuzione. Possiamo ad esempio assegnare un nuovo attributo alla classe o addirittura eliminarlo. Ad esempio:

```
1 nc = NuovaClasse()  
2 nc.nome = "Gennaro"  
3 nc.eta = 20  
4 nc.indirizzo = "via via via ecc."
```

Come abbiamo visto nella dichiarazione della classe NuovaClasse non è stato dichiarato alcun attributo, eppure gli attributi nome, eta, indirizzo vengono assegnati senza alcun errore.

Assegnazione attributi

```
1 nc = NuovaClasse()  
2 nc.nome = "Gennaro"  
3 nc.eta = 20  
4 nc.indirizzo = "via via via ecc."
```

L'assegnazione dinamica degli attributi permette di creare classi molto flessibili.

Richiede però maggiore attenzione durante la scrittura del codice per evitare eventuali bug.

E' possibile sbagliare creando un nuovo attributo anzichè assegnando il valore ad uno esistente.

Sebbene questo tipo di errori siano abbastanza rari esistono comunque strumenti come *PyChecker* e *PyLint* che analizzano il codice alla ricerca di probabili errori come ad esempio l'assegnazione di valori ad attributi non esistenti.

Inizializzazione

L'inizializzazione delle classi in Python avviene tramite il metodo `__init__` ovvero il costruttore.

Come tutti i metodi delle classi Python ha bisogno di almeno un parametro.

Per convenzione questo parametro viene chiamato `self`. Ad esso viene passata l'istanza della classe al momento della chiamata del metodo.

```
1 class Punto(object):
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6 p = Punto(1, 2)
7 p.x      # 1
8 p.y      # 2
```

Inizializzazione

```
1 class Punto(object):
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6 p = Punto(1, 2)
7 p.x      # 1
8 p.y      # 2
```

Il risultato dell'inizializzazione è la creazione di due attributi `x` e `y` con i rispettivi valori iniziali 1 e 2.

Anche se Python permette di definire dinamicamente gli attributi di una classe è comunque una buona norma crearli nel costruttore in modo da semplificare il codice che utilizzerà tale classe.

I metodi

I metodi si definiscono con la stessa sintassi delle funzioni, con l'unica differenza che c'è un parametro obbligatorio.

```
1 class Punto(object):
2     def __init__(self, x, y):
3         self.x = x
4         self.y = y
5
6     def origine(self):
7         self.x = 0
8         self.y = 0
```

Il parametro `self` (come visto in precedenza) contiene l'istanza della classe che ha chiamato il metodo.

La riga 6 definisce il metodo `origine` che imposta a 0 entrambi gli attributi `x` e `y` dell'istanza.

I metodi

```
1 class Punto(object):
2     # ...
3     def sposta(self, x, y):
4         self.x += x
5         self.y += y
```

Un altro esempio è il metodo `sposta` che richiede due parametri `x` e `y`. Esso aggiunge i valori di `x` e di `y` rispettivamente agli attributi `x` e `y` dell'istanza.

```
1 p = Punto(10, 20)
2 p.x      # 10
3 p.y      # 20
4 p.sposta(5, -10)
5 p.x      # 15
6 p.y      # 10
```

I metodi speciali

Definendo la classe *Punto* è stato creato un nuovo tipo che come tale deve poter effettuare tutte le più comuni operazioni.

Implementiamo ad esempio l'addizione e la sottrazione tra due classi Punto:

```
1 class Punto(object):
2     # ...
3     def __add__(self, punto):
4         return Punto(self.x + punto.x, self.y + punto.y)
5     def __sub__(self, punto):
6         return Punto(self.x - punto.x, self.y - punto.y)
```

Il metodo `__add__` viene chiamato quando viene sommato un valore alla classe Punto con l'operatore `+`.

La stessa cosa vale per il metodo `__sub__` con l'operatore `-`.

I metodi speciali

```
1 class Punto(object):
2     # ...
3     def __add__(self, punto):
4         return Punto(self.x + punto.x, self.y + punto.y)
5     def __sub__(self, punto):
6         return Punto(self.x - punto.x, self.y - punto.y)
```

I parametri per i metodi speciali come `__add__` e `__sub__` sono due:

- `self` obbligatorio per tutti i metodi di una classe e a cui sarà assegnata l'istanza della classe a sinistra dell'operatore.
- `punto` che conterrà l'istanza della classe a destra dell'operatore. È stato chiamato *punto* per chiarezza ma il nome può essere qualunque.

Ad esempio:

```
1 p1 = Punto(5, 3)      # Crea il primo punto
2 p2 = Punto(2, 5)      # Crea il secondo punto
3 p3 = p1 + p2          # Viene chiamato il metodo __add__
4 p4 = p1 - p2          # Viene chiamato il metodo __sub__
5
6 p3.x                  # 7
7 p3.y                  # 8
8 p4.x                  # 3
9 p4.y                  # -2
```

Lo stesso vale per tutti gli altri operatori come moltiplicazione, divisione, elevazione a potenza, ecc. Per ulteriori dettagli si veda la guida di riferimento ufficiale su docs.python.org .

Introspezione

L'introspezione è la capacità di ricavare informazioni sui tipi durante l'esecuzione del codice. Queste informazioni possono essere: attributi o metodi di una classe, i parametri di una funzione, i valori predefiniti, il tipo di una variabile, ecc.

Una delle principali funzioni di Python, inerente l'introspezione è `type`. Essa restituisce il *tipo* della variabile che gli viene passata come parametro.

```
1 type(1)                # <type 'int'>
2 type(4.5)              # <type 'float'>
3 type("ciao")          # <type 'str'>
4 type(Punto)           # <type 'type'>
5 type(myFunc)          # <type 'function'>
6 type((1,2,3))         # <type 'tuple'>
7 type(["a", 2, "bc"])  # <type 'list'>
```

Introspezione

Grazie a `type` si può cambiare, ad esempio, il comportamento di una funzione in base al tipo di variabili che viene passato.

Possiamo utilizzare `type` ad esempio per modificare i metodi `__add__` e `__sub__` della classe *Punto* che non effettuano alcun controllo sul tipo di valore passato.

```
1 class Punto(object):
2     # ...
3     def __add__(self, punto):
4         return Punto(self.x + punto.x, self.y + punto.y)
5     def __sub__(self, punto):
6         return Punto(self.x - punto.x, self.y - punto.y)
```

Allo stato attuale l'operazione funziona se il tipo di variabile che si somma al *Punto* ha due attributi numerici `x` e `y`.

Modificando i metodi della classe Punto in questo modo:

```
1 def __add__(self, punto):
2     if type(punto) is Punto:
3         return Punto(self.x + punto.x, self.y + punto.y)
4     if type(punto) is int:
5         return Punto(self.x + punto, self.y + punto)
6     return NotImplemented
7 def __sub__(self, punto):
8     if type(punto) is Punto:
9         return Punto(self.x - punto.x, self.y - punto.y)
10    if type(punto) is int:
11        return Punto(self.x - punto, self.y - punto)
12    return NotImplemented
```

si può aggiungere un punto ad un altro punto oppure un punto ad un numero intero.

Introspezione

```
1  def __add__(self, punto):
2      if type(punto) is Punto:
3          return Punto(self.x + punto.x, self.y + punto.y)
4      if type(punto) is int:
5          return Punto(self.x + punto, self.y + punto)
6      return NotImplemented
7  def __sub__(self, punto):
8      if type(punto) is Punto:
9          return Punto(self.x - punto.x, self.y - punto.y)
10     if type(punto) is int:
11         return Punto(self.x - punto, self.y - punto)
12     return NotImplemented
```

Le righe 6 e 12 ritornano il tipo `NotImplemented` nel caso in cui il tipo da sommare non sia nessuno di quelli supportati.

```
1  def __add__(self, punto):
2      if type(punto) is Punto:
3          return Punto(self.x + punto.x, self.y + punto.y)
4      if type(punto) is int:
5          return Punto(self.x + punto, self.y + punto)
6      return NotImplemented
7  def __sub__(self, punto):
8      if type(punto) is Punto:
9          return Punto(self.x - punto.x, self.y - punto.y)
10     if type(punto) is int:
11         return Punto(self.x - punto, self.y - punto)
12     return NotImplemented
```

È importante rispettare questo standard di ritornare `NotImplemented`. Serve ad avvisare Python che l'operazione non è supportata.

Introspezione

```
1  def __add__(self, punto):
2      if type(punto) is Punto:
3          return Punto(self.x + punto.x, self.y + punto.y)
4      if type(punto) is int:
5          return Punto(self.x + punto, self.y + punto)
6      return NotImplemented
7  def __sub__(self, punto):
8      if type(punto) is Punto:
9          return Punto(self.x - punto.x, self.y - punto.y)
10     if type(punto) is int:
11         return Punto(self.x - punto, self.y - punto)
12     return NotImplemented
```

Nelle prossime slide verrà illustrato il motivo pratico per cui conviene rispettare questo standard.

Ad esempio:

```
1 p1 = Punto(3, 4)
2 p2 = Punto(5, 9)
3
4 p1 + p2          # Punto(8, 13)
5 p1 + 10          # Punto(13, 14)
6 p2 + 5           # Punto(10, 14)
```

Ciò è sicuramente più pulito e leggibile di un metodo per fare l'addizione: `p3 = p1.addiziona(p2)` .

Una nota sui metodi speciali

Che succede se anzichè aggiungere un Punto ad un intero effettuo l'operazione invertendo gli operandi?

Una nota sui metodi speciali

Che succede se anzichè aggiungere un Punto ad un intero effettuo l'operazione invertendo gli operandi?

La riga `p2 = 10 + p1` restituisce un errore. Perché?

Una nota sui metodi speciali

Che succede se anzichè aggiungere un Punto ad un intero effettuo l'operazione invertendo gli operandi?

La riga `p2 = 10 + p1` restituisce un errore. Perché?

La spiegazione è semplice: `10` è un numero intero. Python chiama il relativo metodo `__add__` (del tipo intero) che restituisce il tipo *NotImplemented* inquanto *non sa* come aggiungere un tipo Punto.

Una nota sui metodi speciali

Che succede se anzichè aggiungere un Punto ad un intero effettuo l'operazione invertendo gli operandi?

La riga `p2 = 10 + p1` restituisce un errore. Perché?

La spiegazione è semplice: `10` è un numero intero. Python chiama il relativo metodo `__add__` (del tipo intero) che restituisce il tipo *NotImplemented* inquanto *non sa* come aggiungere un tipo Punto.

Prima di restituire un errore Python controlla se il metodo `__radd__` è definito per il tipo Punto.

Nel caso lo sia esso viene chiamato passando gli stessi parametri come per il metodo `__add__`.

Nel nostro caso l'operazione è commutativa e quindi `__radd__` deve essere uguale a `__add__`.

Una nota sui metodi speciali

La definizione della classe Punto viene quindi modificata in questo modo:

```
1 def __add__(self, punto):
2     if type(punto) is Punto:
3         return Punto(self.x + punto.x, self.y + punto.y)
4     if type(punto) is int:
5         return Punto(self.x + punto, self.y + punto)
6 __radd__ = __add__
7
8 def __sub__(self, punto):
9     # ...
10 __rsub__ = __sub__
```

definendo i due nuovi metodi `__radd__` e `__rsub__` (riga 6 e riga 10).

Un'altra funzione fondamentale dell'introspezione in Python è `dir`.

Data una variabile di qualunque tipo restituisce una lista contenente i nomi di tutti gli attributi e di tutti i metodi.

Ad esempio: `dir(Punto)` restituisce tra i tanti metodi ereditati dal tipo *object* anche

```
1     ["__init__", "origine", "sposta", "__add__",  
2     "__sub__", "__radd__", "__rsub__"]
```

ovvero i metodi definiti precedentemente.

La combinazione `dir` più `type` permette, ad esempio, di analizzare la struttura di un tipo ed estrarre delle informazioni che interessano in modo dinamico.

Di seguito creeremo una funzione che data una classe o un'istanza restituisce una lista dei soli metodi.

Per ottenere questo risultato verranno utilizzate altre due importanti funzioni dell'introspezione in Python, ovvero:

- `getattr` : richiede due parametri, il primo è l'oggetto su cui cercare l'attributo il cui nome è passato nel secondo parametro della funzione.
- `callable` : ritorna True se l'oggetto passato come parametro è una funzione, un metodo o comunque un oggetto utilizzabile come un funzione.

Abbiamo ora tutti gli strumenti per creare la funzione descritta sopra, vediamo come realizzarla...

Introspezione

```
1  def estraiMetodi (oggetto):
2      metodi = []
3      for attr in dir(oggetto):
4          if callable(getattr(oggetto, attr)):
5              metodi.append(attr)
6      return metodi
7
8  print estraiMetodi(Punto)
9  # => ["__init__", "origine", "sposta",
10 # =>      "__add__", "__sub__", "__radd__", "__rsub__"]
```

La funzione è semplice: crea una lista e vi inserisce tutti i nomi dei metodi di `oggetto`.

Introspezione

```
1  def estraiMetodi (oggetto):
2      metodi = []
3      for attr in dir(oggetto):
4          if callable(getattr(oggetto, attr)):
5              metodi.append(attr)
6      return metodi
7
8  print estraiMetodi(Punto)
9  # => ["__init__", "origine", "sposta",
10 # =>      "__add__", "__sub__", "__radd__", "__rsub__"]
```

Riga 3: inizia un ciclo sulla lista restituita da `dir(oggetto)` assegnando alla variabile `attr` i valori di `dir(oggetto)` uno per volta.

Introspezione

```
1  def estraiMetodi (oggetto):
2      metodi = []
3      for attr in dir(oggetto):
4          if callable(getattr(oggetto, attr)):
5              metodi.append(attr)
6      return metodi
7
8  print estraiMetodi(Punto)
9  # => ["__init__", "origine", "sposta",
10 # =>      "__add__", "__sub__", "__radd__", "__rsub__"]
```

Riga 4: `getattr(oggetto, attr)` restituisce l'attributo `attr` di `oggetto`, sia esso un metodo o altro.

Ad esempio: per `p = Punto(30, 20)`,
`getattr(p, 'x')` restituisce `30`.

Introspezione

```
1  def estraiMetodi (oggetto):
2      metodi = []
3      for attr in dir(oggetto):
4          if callable(getattr(oggetto, attr)):
5              metodi.append(attr)
6      return metodi
7
8  print estraiMetodi(Punto)
9  # => ["__init__", "origine", "sposta",
10 # =>      "__add__", "__sub__", "__radd__", "__rsub__"]
```

Riga 4: l'attributo restituito da *getattr* viene passato a **callable** che ritorna True se è un metodo.

Introspezione

```
1  def estraiMetodi (oggetto):
2      metodi = []
3      for attr in dir(oggetto):
4          if callable(getattr(oggetto, attr)):
5              metodi.append(attr)
6      return metodi
7
8  print estraiMetodi(Punto)
9  # => ["__init__", "origine", "sposta",
10 # =>     "__add__", "__sub__", "__radd__", "__rsub__"]
```

In caso affermativo il nome dell'attributo viene aggiunto alla lista *metodi* (riga 5).

Ereditarietà

In Python definire una classe che ne estenda un'altra è molto semplice. Ne abbiamo già visto un caso nella definizione del tipo *Punto*.

```
1 class Punto(object):  
2     # ...
```

La definizione indica la creazione del nuovo tipo *Punto* che discende dal tipo *object*.

In Python definire una classe che ne estenda un'altra è molto semplice. Ne abbiamo già visto un caso nella definizione del tipo *Punto*.

```
1 class Punto(object):  
2     # ...
```

La definizione indica la creazione del nuovo tipo *Punto* che discende dal tipo *object*.

Attenzione: in Python ogni classe base deve discendere da *object*. Se viene omissa Python crea una classe detta *old-style* che risale alla vecchia gestione dei tipi di Python, ormai in disuso e mantenuta solo per compatibilità con le versioni precedenti.

Per maggiori dettagli consultare la guida ufficiale

<http://docs.python.org/ref/node33.html>

Vediamo un esempio concreto di questo concetto estendendo la classe *Punto*.

Creiamo ora un nuovo tipo di dati *Punto3D* che differisce da *Punto* per l'aggiunta di una coordinata, ovvero *z*.

Vediamo un esempio concreto di questo concetto estendendo la classe *Punto*.

Creiamo ora un nuovo tipo di dati *Punto3D* che differisce da *Punto* per l'aggiunta di una coordinata, ovvero *z*.

```
1 class Punto3D(Punto):
2     def __init__(self, x, y, z):
3         Punto.__init__(self, x, y)
4         self.z = z
```

Questa è la definizione del tipo *Punto3D*. Per ora è stato definito solo il costruttore, gli altri metodi saranno definiti in seguito.

Vediamo nel dettaglio la definizione...

Ereditarietà

```
1 class Punto3D(Punto):  
2     def __init__(self, x, y, z):  
3         Punto.__init__(self, x, y)  
4         self.z = z
```

Riga 1: viene definito il tipo *Punto3D* che deriva dal tipo *Punto* da cui eredita tutti gli attributi e i metodi.

Ereditarietà

```
1 class Punto3D(Punto):  
2     def __init__(self, x, y, z):  
3         Punto.__init__(self, x, y)  
4         self.z = z
```

Riga 1: viene definito il tipo *Punto3D* che deriva dal tipo *Punto* da cui eredita tutti gli attributi e i metodi.

Riga 2: viene quindi definito il nuovo costruttore.

La creazione di un *Punto3D* richiede una coordinata in più rispetto a *Punto*.

Quindi i parametri diventano 3: *x*, *y* e *z*.

```
1 class Punto3D(Punto):  
2     def __init__(self, x, y, z):  
3         Punto.__init__(self, x, y)  
4         self.z = z
```

Riga 1: viene definito il tipo *Punto3D* che deriva dal tipo *Punto* da cui eredita tutti gli attributi e i metodi.

Riga 2: viene quindi definito il nuovo costruttore.

La creazione di un *Punto3D* richiede una coordinata in più rispetto a *Punto*.

Quindi i parametri diventano 3: *x*, *y* e *z*.

Riga 3: a questo punto, come per ogni linguaggio di programmazione a oggetti, viene chiamato il costruttore della classe genitore passandogli tutti i parametri necessari, compreso *self*.

Ereditarietà

```
1 class Punto3D(Punto):  
2     def __init__(self, x, y, z):  
3         Punto.__init__(self, x, y)  
4         self.z = z
```

Riga 3: il costruttore di Punto provvederà ad inizializzare gli attributi x e y, come precedentemente definito.

```
1 class Punto3D(Punto):  
2     def __init__(self, x, y, z):  
3         Punto.__init__(self, x, y)  
4         self.z = z
```

Riga 3: il costruttore di Punto provvederà ad inizializzare gli attributi x e y, come precedentemente definito.

Riga 4: infine viene definito il nuovo attributo della classe Punto3D ovvero z.

Ereditarietà

```
1 class Punto3D(Punto):  
2     def __init__(self, x, y, z):  
3         Punto.__init__(self, x, y)  
4         self.z = z
```

Riga 3: il costruttore di Punto provvederà ad inizializzare gli attributi x e y, come precedentemente definito.

Riga 4: infine viene definito il nuovo attributo della classe Punto3D ovvero z.

Punto3D è ora pronto per essere istanziato: `p = Punto3D(3, 6, 10)`.

Viene illustrata di seguito la definizione del metodo `__add__` per il tipo *Punto3D*. Le stesse regole valgono anche per gli altri metodi speciali.

Viene illustrata di seguito la definizione del metodo `__add__` per il tipo *Punto3D*. Le stesse regole valgono anche per gli altri metodi speciali.

```
1 def __add__(self, valore):  
2     if type(valore) is Punto3D:  
3         return Punto3D(self.x + valore.x,  
4             self.y + valore.y,  
5             self.z + valore.z)
```

Viene controllato se il tipo da aggiungere è un *Punto3D* (riga 2), nel caso viene restituito un altro *Punto3D* sommando ogni coordinata (riga 3).

```
6     if type(valore) is Punto:  
7         return Punto3D(self.x + valore.x,  
8             self.y + valore.y,  
9             self.z)
```

Se invece l'addizione viene fatta con il tipo *Punto* la somma degli attributi viene fatta solo per x e y, lasciando z invariato e restituendo comunque un *Punto3D*.

```
10     if type(valore) is int:
11         return Punto3D(self.x + valore,
12                         self.y + valore,
13                         self.z + valore)
14     return NotImplemented
15 __radd__ = __add__
```

Se invece l'addizione viene fatta con un numero intero allora esso viene addizionato a tutti gli attributi del punto (x,y,z).

```
10     if type(valore) is int:
11         return Punto3D(self.x + valore,
12                         self.y + valore,
13                         self.z + valore)
14     return NotImplemented
15 __radd__ = __add__
```

Se invece l'addizione viene fatta con un numero intero allora esso viene addizionato a tutti gli attributi del punto (x,y,z).

Da notare che alla fine del metodo `__add__` (riga 14) c'è `return NotImplemented` che indica a Python l'impossibilità di effettuare l'operazione se non con un *Punto*, un *Punto3D* o un numero intero.

```
10     if type(valore) is int:
11         return Punto3D(self.x + valore,
12                         self.y + valore,
13                         self.z + valore)
14     return NotImplemented
15 __radd__ = __add__
```

Se invece l'addizione viene fatta con un numero intero allora esso viene addizionato a tutti gli attributi del punto (x,y,z).

Da notare che alla fine del metodo `__add__` (riga 14) c'è `return NotImplemented` che indica a Python l'impossibilità di effettuare l'operazione se non con un *Punto*, un *Punto3D* o un numero intero.

Infine viene definito il metodo `__radd__` come per *Punto*.

Conclusioni

Sono stati curati, in questa lezione, i seguenti argomenti:

- Le classi in Python: costruzione di nuovi tipi.
- I metodi speciali.
- L'introspezione: una panoramica sulle funzionalità.
- Ereditarietà in Python.

Prova ad esercitarti creando nuovi tipi di dati. Ad esempio, prova ad implementare le frazioni in Python. Crea un tipo che abbia gli attributi di numeratore e denominatore ed implementa tutte le operazioni aritmetiche relative.

La seconda lezione finisce quì!
Alla prossima!