Lezione 7

Object Oriented Programming (OOP)

Python è un linguaggio di programmazione che supporta il **paradigma della programmazione a oggetti** (*Object Oriented Programming*).

A differenza del *procedural programming* (visto finora) in cui il *software* è una lista di istruzioni organizzata in funzioni (procedure) che manipolano i dati, nell'*Object Oriented Programming* il *software* è costruito tramite **oggetti** che interagiscono tra loro attraverso lo scambio di messaggi

Ogni **oggetto** contiene sia le strutture dati (attributi o campi) che le istruzioni (procedure o metodi) necessarie a manipolarli. Una particolarità degli **oggetti** è che i metodi possono accedere e modificare i **propri *data field***, questa funzionalità è evidenziata in Python dalla keyword `self` (in altri linguaggi è *this*)

Gli oggetti sono definiti e descritti tramite una **classe** che è l'astrazione di un oggetto reale (es. `Turtle` /Tartaruga) con i suoi attributi che ne rispecchiano le caratteristiche, e i suoi metodi che descrivono il modo in cui in interagisce con l'ambiente. Un oggetto è l'istanza di una classe.

Classi

Finora abbiamo utilizzato **oggetti** descitti da classi già disponibili in Python: `int`, `str`, `turtle.Turtle`, `turtle.Window` ...

Ma nello scrivere un programma spesso è necessario definire nuovi *tipi di oggetti*.
La sintassi per descrivere la classe di un nuovo oggetto è: keyword `class`, nome (CapitalLetters) del nuovo oggetto e *token* `:`

```
class MyObject :
    """Docstring per descrivere l'oggetto"""

    attributo = 5  # attributo (dati) dell'oggetto

    def metodo_aggiungi(self) :  # istruzioni (metodo) per interagire (self)
        self.attributo += 1      # operazione su attributi dell'oggetto (self)
```

Ogni *instance method* **dell'oggetto** ha sempre come primo parametro `self` (self, convenzione) che indica la **referenza all'oggetto**

Per accedere agli attributi dell'oggetto dai metodi, si usa `self`

Inizializzazione

Quando si costruisce un oggetto (istanza) della classe, questo deve essere inizializzato. Per farlo esiste un *metodo speciale* `__init__`, detto **metodo di inizializzazione** che è chiamato automaticamente come prima azione del **Costruttore**.
Nell' `__init__`, ad esempio, è possibile assegnare il valore iniziale agli attributi.

Definiamo una classe `Punto`

```
class Punto:
    """Una classe per rappresentare e operere con una coppia di coordinate x,y come punto su un piano"""

    def __init__(self):
        self.x = 0
        self.y = 0
```

In questo caso il **metodo di inizializzazione** si occupa di definire il valore di partenza degli attributi dell'oggetto istanziato dal **Costruttore**:

```
>>> p = Punto()   # Costruttore della classe Punto. Ha lo stesso nome della classe
```

e sarà *poi* possibile vedere il valore dei suoi attributi con il token `.`

```
>>> print(p.x, p.y)
0 0
```

Costruttore

```
>>> p = Punto()
```

La funzione `Punto()` si chiama **costruttore**: crea un nuovo oggetto della classe.
Ogni classe mette automaticamente a disposizione una funzione costruttore che ha sempre il nome della classe.

La classe si può pensare come una fabbrica: contiene tutto il necessario per creare un oggetto. Ogni volta che si chiama il costruttore, si chiede alla fabbrica di produrre un nuovo oggetto.
Appena l'oggetto è stato creato, il metodo `__init__` viene eseguito per settare il *factory default*.

La combinazione di:

- Crea nuovo oggetto
- Imposta il *factory default*

si chiama **inizializzazione**

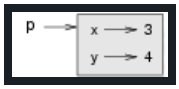
> Ogni classe ha a disposizione dei metodi *speciali* predefiniti (detti *dunder*, da *double under_score*)
>
> Non sono chiamati esplicitamente dal programmatore, ma direttamante dall'interprete in situazioni particolari.
>
> Quando è necessario modificarli bisogna farne l'*overload*

Attributi

Si può *accedere* e modificare gli attributi di un oggetto con la *dot notation* `.`

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

Gli attributi *vivono* nei *namespace* dei moduli e degli oggetti: l'importante è capire l'ordine delle assegnazioni che è **risolto** dalla **dot notation**.



La variabile `p` punta ad un oggetto di tipo `Punto` che ha due *attributi*: ogni attributo punta ad un valore.

Per accedere al valore di un attributo uso la sintassi:

```
>>> x = p.x
>>> print(x)
3
```

`p.x` significa "vai all'oggetto a cui punta p e recupera il valore del suo attributo x".

ATTENZIONE: Assegnando il valore `p.x` ad una variabile `x` non c'è nessun conflitto tra la variabile `x` (*global namespace*) e l'attributo `x` (che appartiene la *namespace* dell'istanza `p`). La notazione tramite `.` (*dot notation*) ha lo scopo di risolvere l'ambiguità

Inizializzazione con parametri

Con la clesse Punto appena definita, se volessimo creare il punto (7,6) dovremmo fare:

```
>>> p = Punto()
>>> p.x = 7
>>> p.y = 6
```

E' possibile generalizzare `__init__` per rendere la classe `Punto` più versatile:

```
class Punto:
    """Una classe per rappresentare e operere con una coppia di coordinate x,y come punto su un piano"""

    def __init__(self, x, y):
        """Inizializza le coordinate del punto"""
        self.x = x
        self.y = y
```

Ora è possibile creare un punto a partire da una coppia di numeri:

```
>>> p = Punto(7, 6)
```

Inizializzazione con default

E' possibile definire dei valori di *default* nell' `__init__` come in qualunque funzione in cui si definiscono dei *parametri opzionali*:

```
class Punto:
    """Una classe per rappresentare e operere con una coppia di coordinate x,y come punto su un piano"""

    def __init__(self, x=0,  y=0):
        """Inizializza le coordinate del punto"""
        self.x = x
        self.y = y
```

Con questa sintassi `x` ed `y` sono opzionali: se nessun valore è passato al costruttore, sarà mantenuto il default (`x = 0` e `y = 0`):

```
>>> p = Punto(7, 6)
>>> print(p.x, p.y)
7 6

>>> q = Punto()
>>> print(q.x, q.y)
0 0
```

Metodi dell'istanza

Un metodo si comporta come una funzione ma è appicato alla specifica istanza (oggetto) della classe:

```
raffaello.forward(50)
```

Si accede ai metodi dell'istanza con la *dot notation*, esattamente come nel caso degli attributi.

Perchè creare una classe specifica `Punto` invece di usare una semplice coppia `(x,y)` ?
Definiamo un nuovo **metodo** della classe che permetta di calcolare la distanza del punto (coppia x,y) dall'origine:

```
class Punto:
    """Una classe per rappresentare e operere con una coppia di coordinate x,y"""

    def __init__(self , x = 0, y = 0):
        """Inizializza le coordinate del punto"""
        self.x = x
        self.y = y

    def distanza_da_origine(self):
        """Calcola la distanza dall'origine"""
        return (self.x**2 + self.y**2)**0.5
```

Ora per ogni **oggetto** di tipo `Punto` sarà possibile chiamare

```
>>> p = Point(4, 5)
>>> dist = p.distanza_da_origine()
>>> print(dist)
6.4031242374328485
```

Non bisogna specificare nessun valore per l'argomento `self` : lo fa Python per noi dietro le quinte!

Infatti scrivendo:

```
dist = p.distanza_da_origine(p)
```

si ottiene un errore:

```
TypeError: distanza_da_origine() takes 1 positional argument but 2 were given
```

in cui la funzione ha ricevuto 2 argomenti (`p` e `self` in automatico), quando ne aspettava solo 1 (`self`)

Metodi della classe

I metodi dell'istanza di una classe (la maggior parte dei metodi che scriveremo) possono accedere liberamente ai metodi e attributi dell'oggetto per modificarne lo stato. Questi si riconoscono perchè utilizzano il parametero `self` . Per poter essere utilizzati **deve esistere un istanza della classe** (cioè deve essere stato chiamato il costruttore)

```
p = Punto()
```

Esistono anche altri metodi che sono comuni a tutti gli oggetti e appartengono alla classe: **class methods**.

Questi non possono modificare lo stato di una specifica istanza, ma solo lo stato della classe. Come i *metodi dell' istanza* conoscevano le informazioni dell'oggetto tramite la variabile `self` , i metodi della classe conoscono le informazioni della classe tramite la variabile `cls` .

Si costruiscono come i metodi dell'istanza, ma si aggiunge il **decoratore** `@classmethod` e si usa il parametro `cls`

Spesso sono utilizzati per simulare l'*overload* del metodo speciale `__init__`

Ad esempio il "default" dell' `__init__` :

```
class Punto:
    """Una classe per rappresentare e operere con una coppia di coordinate x,y"""

    def __init__(self , x=0, y=0):
        """Inizializza le coordinate del punto"""
        self.x = x
        self.y = y
```

puo' essere scritto meglio con un metodo della classe `Punto` che vale per qualunque istanza

```
class Punto:
    """Una classe per rappresentare e operere con una coppia di coordinate x,y"""

    @classmethod
    def origin(cls) :
        """Metodo della classe che imposta i valori x=0, y=0"""
        return cls(0, 0)   # cls contiene tutte le informazioni sulla classe

    def __init__(self, x, y):
        """Inizializza le coordinate del punto"""
        self.x = x
        self.y = y
```

In questo caso sarà possibile creare il punto (0,0) con

```
p = Punto.origin()
```

> Questi metodi non necessitano di una istanza della classe per essere 'chiamati'

Metodi statici

Il terzo tipo di metodi che si possono avere sono i metodi **statici**. A differenza dei precedenti, questi non sono legati nè all'istanza, nè alla classe: non possono modificare lo stato dell'oggetto e neppure lo stato della classe.
Possono accedere solo ad altri attibuti statici dell'oggetto.

> possono essere pensati come "funzioni" nel namespace della classe, e sono utili per definire operazioni comuni a tutti gli oggetti, ma che non necessitano di sapere lo stato degli oggetti.

Si costruiscono come delle funzioni a cui si aggiunge il **decoratore** `@staticmethod`

Ad esempio, se volessimo controllare se una coordinata è valida (entro un limite da noi definito):

```
class Punto:
    """Una classe per rappresentare e operere con una coppia di coordinate x,y"""

    @classmethod
    def origin(cls) :
        """Metodo della classe che imposta i valori x=0, y=0"""
        return cls(0, 0)   # cls contiene tutte le informazioni sulla classe


    def __init__(self, x, y):
        """Inizializza le coordinate del punto"""
        self.x = x
        self.y = y

    @staticmethod
    def coordinata_valida(x) : # NON c'è self: non so nulla dell'istanza
        """controllo se la coordinata è entro il limite 100"""
        return x < 100
```

Per la classe punto, la validità della coordinata non dipende dall'esistenza dell'oggetto `p = Punto()`, quindi

```
>>> Punto.coordinata_valida(273)
False
```

Metodi speciali

I metodi **speciali** (*dunder*) sono chiamati direttamante dall'interprete in situazioni particolari. Alcuni di questi che useremo sono:

- `__init__()` : chiamato nell'inizializzazione `mc = MyClass()`
- `__str__()` : chiamato nella conversione a string o nel print `str(mc)`
- `__call__()` : chiamato quando si usa un'istanza come una funzione `MyClass()`
- `__len__()` : chiamato quando si usa il built-in *len* `len(mc)`
- `__contains__(x)` : chiamato quando si usa il built-in *in* `x in mc`
- `__getitem__(key)` : chiamato quando si usa l'operatore indicizzazione `mc[key]`
- `__setitem__(key,val)` : chiamato quando si usa l'operatore indicizzazione `mc[key] = val`

e gli operatori `__eq__(x)` per `==`, `__lt__(x)` per `<`, `__le__(x)` per `<=`, `__gt__(x)` per `>`, ...e altri (https://docs.python.org/3/reference/datamodel.html#special-method-names)

Utilizzare gli oggetti

Un oggetto instanziato da una classe definita da noi è un **tipo** che si comporta come tutti gli altri tipi di Python finora visti e puo' essere ulizzato allo stesso modo con le funzioni built-in. Ad esempio `type`:

```
>>> p = Punto()
>>> print(type(p))
<class '__main__.Punto'>
```

E si può utilizzare come qualunque altro tipo dati visto finora. Ad esempio si può scrivere una funzione che usa il tipo `Punto` come parametro:

```
def punto_medio(p1, p2):
    """Calcola il punto medio tra due oggetti di tipo Punto
       * p1: tipo Punto
       * p2: tipo Punto
       Return: un oggetto di tipo Punto
    """

    mx = (p1.x + p2.x)/2
    my = (p1.y + p2.y)/2

    return Punto(mx, my)
```

la funzione `punto_medio` così definita ha come argomenti due oggetti di tipo `Punto` e ritorna un oggetto di tipo `Punto`:

```
>>> p = Punto(3, 4)
>>> q = Punto(5, 12)
>>> r = punto_medio(p, q)
>>> print(r.x, r.y)
4.0  8.0
```

Si possono ritornare istanze di `Punto` anche dei **metodi** dell'oggetto:

```
class Punto:
    #...

    def punto_medio(self, target):
        """Restitusce il punto medio tra me e target (Punto)"""

        mx = (self.x + target.x)/2
        my = (self.y + target.y)/2

        return Punto(mx, my)
```

Facciamo Pratica

1. Modificare la classe `Punto` aggiungendo il metodo `distanza_da_punto` che calcola la distanza da un secondo punto
2. Fare l'*overload* del metodo `__str__` per rappresentare in modo leggibile l'oggetto `Punto`

Ereditare da altre classi

Una caratterisica fondamentale della programmazione ad oggetti è la possibilità di **ereditare** metodi da altre classi (*ereditarietà*)

In pratica, l'**ereditarietà** ci permette di definire una classe che sia una specializzazione di un'altra classe più generica.

La nuova classe specifica (*child class*) **eredita** dalla classe generica (*parent class*) tutti i metodi.

L'**ereditarietà** mi consente di aggiungere metodi e funzionalità, senza modificare una classe già esistente che rappresenta in modo completo ed esaustivo un determinato oggetto.

`Punto` e `PuntoMassa`

Prendiamo ad esempio un oggetto *massa puntiforme*. Questo oggetto ha delle caratteristiche e funzionalità **specifiche** legate alla sua proprietà *massa*, ma dal **generico** punto di vista geometrico si comporta esattamente come un oggetto di tipo *Punto*.

Concettualmente un oggetto *massa puntiforme* è una **specializzazione** dell'oggetto `Punto` *nel caso in cui si sia anche* la proprietà "massa". La classe `PuntoMassa` può quindi essere costuita *a partire* dalla classe `Punto` ereditandone tutti i suoi metodi.

In Python una classe figlia si costruisce aggiungendo come *argomento* delle definizione, la classe *parent* che si vuole ereditare:

```
class PuntoMassa(Punto):
    """Una classe per rappresentare una massa puntiforme"""

    pass
```

> `pass` è la keyword Python che "non fa nulla" e spesso si usa come placeholder

Costruendo un oggetto di tipo `PuntoMassa`:

```
>>> pm = PuntoMassa(4,5)
```

Si possono usare tutti i metodi ereditati da `Punto` (**compreso `__init__`**).

L'oggetto *massa puntiforme* è caratterizzato anche da una massa, oltre che dalle sue coordinate: va quindi inizializzato *sia* con le coordinata *che* con il valore della massa.

Per farlo serve un metodo `__init__` *diverso* da quello della classe `Punto` che abbia 3 parametri:

```
class PuntoMassa(Punto):
    """Una classe per rappresentare una massa puntiforme"""

    def __init__(self, x=0, y=0, m=0):
        """Sovrascrive l'__init__ di Punto per tenere conto della massa"""
        Punto.__init__(x, y)  # Per le coordinate, uso l'__init__ di 'Punto'
        self.m = m            # Aggiungo il nuovo attributo: massa
```

> **Attenzione**: il metodo `__init__` della funzione *child* ha la precedenza (**sovrascrive**) quello della classe *parent*

Aggiungere metodi

Ogni oggetto di tipo `PuntoMassa` conosce (eredita) tutti i metodi di `Punto` per quanto riguarda l'aspetto geometrico.

La nuova classe **derivata** può concentrarsi sulla gestione dell'aspetto *fisico* dell'oggetto: le operazioni sulla massa:

```
class PuntoMassa(Punto):
    #...
    g = 9.81

    def peso(self):
        """Restitusce il peso dell'oggetto"""

        return self.m*g
```

Abbiamo creato una classe specializzata `PuntoMassa` senza modificare in alcun modo la classe più generica `Punto` : abbiamo un oggetto per definire il concetto di punto e un oggetto per rappresentare un particolare tipo di punto: la massa puntiforme.

Abbiamo tradotto nel codice la stessa astrazione che facciamo nel mondo reale: abbiamo ragionato **a oggetti**!

Riassunto

```
class Punto:
    """Classe per rappresentare e operere con una coppia di coordinate x,y"""

    def __init__(self, x=0, y=0):
        """Inizializza le coordinate del punto"""
        self.x = x
        self.y = y

class PuntoMassa(Punto):
    """Classe per rappresentare un punto con massa"""
    g = 9.81

    def __init__(self, x=0, y=0, m=0):
        """Sovrascrive l'__init__ di Punto per tenere conto della massa"""
        super().__init__(x, y)  # Per le coordinate, uso l'__init__ di 'Punto'
                                # NOTA:uso SUPER
        self.m = m              # Aggiungo il nuovo attributo: massa

    def peso(self):
        """Restitusce il peso dell'oggetto"""
        return self.m*self.g
```

Si può usare così:

```
>>> pm = PuntoMassa(1, 2, 10)
>>> print(pm.x)  # x è definito in Punto
1
>>> print(pm.peso()) # peso() è definito in PuntoMassa
98.1
```

La funzione `super()` e MRO

La funzione built-in `super()` ritorna un oggetto temporaneo della *superclass* che permette di accedere ai metodi della classe base in modo esplicito. E' molto utile perchè:

- permette di non usare il nome della classe base in modo esplicito
- semplifica la gestione di ereditarietà multipla

Usiamo sempre `super()`

Il **Method Resolution Order** (MRO) è l'ordine con cui i metodi sono chiamati nel caso di ereditarietà multiple. Si può sempe controllare con l'attributo `__mro__` dell'oggetto.

Le regole base sono:

- Il metodo della classe derivata è sempre chiamato prima di quello della classe base
- Se ci sono più classi *parent*, il metodo è chiamato nell'ordine con cui sono elencate nella definizione della classe derivata

Convenzioni

- Per i nomi delle **classi** si usa il *CapitalLetters*: `MyClass`
- Per i nomi dei **metodi** si usa il *snake_case*: `mc.my_method`
- Mettere sempre una *docstring* che spieghi l'oggetto rappresentato da quella classe
- `self` per gli *instance method*
- `cls` per i *class method*
- Python non differenzia tra variabili pubbliche e private:
  - metodi e attributi "privati" si indicano con *under_score*: `mc._metodo_privato`

    ATTENZIONE: non `__` per non fare confusione con i metodi *dunder*