UNIVERSITÀ di VERONA

Linguaggio di Programmazione Python

26/04/2022

Giulio Mazzi

giulio.mazzi@univr.it

Programmazione ad oggetti

La *programmazione a oggetti (OOP)* è un paradigma che si basa sull'idea di **classi**: tipi personalizzati definiti da un programmatore per raccogliere e organizzare codice e dati.

L'OOP è un topic enorme. Nel corso ne vedremo solo una piccola parte, principalmente legata all'uso di classi e oggetti già pronti.

Definiamo:

- **Classe**: un tipo. In python tutto è una classe, compresi tipi builtin
- **Oggetto (o istanza)**: un elemento di una classe. Per esempio, 5 è un oggetto della classe `int`, `'ciao'` è un oggetto/istanza della classe `str`...

Classi

Possiamo definire una **classe** nel seguente modo:

```
class Punto:
    """ Rappresenta un punto in un piano bidimensionale """
```

Possiamo definire un **oggetto** della nostra classe. I dati al suo interno si chiamano **attributi della classe**.

```
p = Punto()
p.x = 5.0
p.y = 6.0
print(f"{p.x = }, {p.y = }")
```

```
p.x = 5.0, p.y = 6.0
```

N.B.: a differenza del Java o altri linguaggi con tipaggio statico, non dobbiamo necessariamente definire gli elementi che stanno all'interno di una classe (ma è buona cosa farlo!).

Classi

Le nostre classi creano oggetti **mutabili**, è possibile modificare i valori contenuti all'interno.

Passare un oggetto della classe a una funzione passa l'elemento stesso, non una sua copia.

Per fare una copia, posso importare il modulo `copy` e usare `copy()` (per copie superficiali) oppure `deepcopy()` (per copie profonde).

```
q = Punto()
q.x = 1
q.y = 2
print(f"{q.x = }, {q.y = }")

def sposta(punto):
    punto.x += 1
    punto.y += 1

sposta(q)
print(f"{q.x = }, {q.y = }")
```

```
q.x = 1, q.y = 2
q.x = 2, q.y = 3
```

Metodi

Le classi non definiscono solo il contenuto (in termini di dati) di un tipo, ma anche il loro **comportamento**. Una funzione interna alla classe si chiama **metodo**, il primo elemento è sempre un riferimento alla classe stessa. Posso usa un metodo con la sintassi `oggetto.metodo(…)`

```
class Time:
    """ Rappresenta l'ora del giorno

        Attributi: hour, minute, second.
    """
    def print_time(self):
        print(f"{self.hour}:{self.minute}:{self.second}")

time = Time()
time.hour = 11
time.minute = 59
time.second = 30
time.print_time()
```

```
11:59:30
```

Magic methods

Esistono dei metodi speciali, chiamati magic methods o "dunders" (*double under*score) che permettono alla nostra classe di comportarsi in modo speciale.

I più comuni sono __init__ e __str__, usati per inizializzare e convertire in stringhe rispettivamente.

Posso anche implementare molto altro, per esempio il supporto all'operatore +.

```
class Time:
    """ Rappresenta l'ora del giorno

        Attributi: hour, minute, second.
    """
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second

    def __str__(self):
        return f"{self.hour:02}:{self.minute:02}:{self.second:02}"

    # ...

start = Time(11, 59, 30)
elapse = Time(1, 22, 19)

start.add_time(elapse)
start.add_seconds(90)
print(start)
```

```
13:23:19
```

Inheritance

Python supporta il meccanismo dell'*ereditarietà, che* permette di derivare nuove classi a partire da quelle già definite realizzando una gerarchia di classi.

La classe eredita tutti I metodi dalla classe padre, e può aggiungerne di nuovi (o specializzare quelli esistenti.

```
class Padre():
    """ classe padre """

class Figlio(Padre):
    """ classe figlio, eredita da Padre """

p = Padre()
f = Figlio()
print(isinstance(p, Figlio), isinstance(f, Padre))
```

```
False True
```

Iteratori

Python usa un meccanismo basato su **iteratori** per scorrere oggetti che rappresentano collezioni finite di altri oggetti.

Per esempio, possiamo scorrere una list usando un iterator con `iter()`

Il meccanismo degli iteratori fa da base a molte delle features che abbiamo visto, come ad esempio le slice, zip, enumerate, items(), ecc...

```python
l = [i for i in range(10)]

# uso esplicito dell'iteratore
for i in iter(l):
    print(i)

# uso implicito, ma identico!
for i in l:
    print(i)
```

Per implementarlo, vanno implementati I metodi `__iter__()` e `__next__()`

Generatori

Il concetto di iteratore si può estendere con l'uso dei generatori: delle funzioni che producono nuovi elementi in maniera sequenziale.

I generatori sono più potenti degli iteratori, possono generare un numero potenzialmente infinito di elementi!

L'istruzione yield generalizza il return. Mi ritorna il valore e salva la posizione, richiamare il generatore mi fa ripartire dallo stesso punto

```
def my_range(n):
    """ versione semplificata di range """
    i = 0
    while i < n:
        yield i
        i += 1

for i in my_range(10):
    print(i, end=',')
```

```
0,1,2,3,4,5,6,7,8,9,
```

Le funzioni sono oggetti

In python, anche le funzioni sono oggetti.

Posso salvare una funzione in una variabile, passarla come argomento ad una altra funzione ecc.

Questa feature permette a python di supportare anche il paradigma della programmazione funzionale.

```
def raddoppia(x):
    return x * 2

l = [i for i in range(1, 6)]
l = list(map(raddoppia, l))
print(l)
```

```
[2, 4, 6, 8, 10]
```

Decoratori

I decoratori sono funzioni che prendono come input altre funzioni, servono ad "arricchire" una funzione.

Sono funzioni che prendono e ritornano una funzione, arricchendola in qualche modo.

Per usarli, uso la sintassi:

```
@decoratore
Def funzione....
```

```python
from time import time

def timer(func):
    # stampa il tempo di esecuzione di una funzione
    def wrap_func(*args, **kwargs):
        t1 = time()
        result = func(*args, **kwargs)
        t2 = time()
        print(f'Funzione {func.__name__} eseguita in {(t2-t1):.4f}s')
        return result
    return wrap_func

@timer
def somma(n):
    res = 0
    for i in range(n):
        res += i
    return res

somma(10)
somma(10000000)
```

```
Funzione somma eseguita in 0.0000s
Funzione somma eseguita in 0.4989s
```