Python: Classi

Rights & Credits

Questo notebook è stato creato da Agostino Migliore. La parte iniziale è stata ispirata da un notebook di:

- Simone Campagna (CINECA)
- Mirko Cestari (CINECA)
- Nicola Spallanzani (CNR-NANO)

Costruzione di una classe

Abbiamo visto il concetto di classe nella prima lezione: un *template* per definire oggetti. Essa rappresenta un modo per definire nuovi oggetti in aggiunta a quello predefiniti. Definire una nuova classe significa definire una nuova categoria, un nuovo tipo di oggetti che sono caratterizzati da **attributi** e **metodi**.

Abbiamo visto pure che una **instance** di una classe è una sua realizzazione specifica, quindi un oggetto specifico della classe, che viene creato a partire dal *template* che rappresenta tale classe.

Per esempio, immaginiamo che non esista il tipo *numero complesso* e creiamolo ex novo come segue:

In [1]:
```
class Complex(object):
    def __init__(self, r=0.0, i=0.0):
        self.re = r   # add attribute "re" to self
        self.im = i       # add attribute "im" to self
    def __mul__(self, other):
        return Complex(self.re*other.re-self.im*other.im,
                       self.re*other.im+self.im*other.re)
    def __imul__(self, other):
        self.re = self.re*other.re-self.im*other.im
        self.im = self.re*other.im+self.im*other.re
        return self
    def __repr__(self):
        return "({0}+{1}j)".format(self.re, self.im)
```

__init__ è un metodo speciale che viene usato per costruire le classi. Possiamo definirlo come il costruttore della classe, attraverso il quale si definiscono e inizializzano gli attributi del nuovo tipo di oggetto.
Il primo argomento passato al costruttore e ad ogni metodo definito per la data classe è self ("se stesso" in inglese), che rappresenta la stessa instance della classe. Gli altri argomenti

sono attributi della classe, cioè attributi di tutti gli oggetti della classe. In questo caso, abbiamo conferito agli oggetti della classe gli attributi `re` ed `im`, che stanno per le sue parti reale e immaginaria. Nella definizione della classe intrinseca `complex` di Python (quella vera), tali attributi sono chiamati `real` ed `imag`.
Definiamo adesso, per esempio, un oggetto della classe appena creata e assegniamogli il nome `A` come segue:

In [2]:
```
A = Complex(r=2, i=3)
```

Potevamo procedere anche come segue:

In [3]:
```
A = Complex(2,3)
```

In [4]:
```
print(A, type(A), sep='\n\n')
```

```
(2+3j)

<class '__main__.Complex'>
```

In [5]:
```
print(A.re, A.im)
```

```
2 3
```

Confrontando la cella di sopra con la definizione della classe, si può notare come `self` sia l'oggetto stesso assegnato ad `A`.

Nel definire la classe, abbiamo anche introdotto i metodi per effettuare prodotti tra oggetti della classe.
L'operazione di moltiplicazione è stata introdotta definendo il metodo __mul__ che agisce tra l'oggetto dato (`self`) e un altra (`other`) instance della stessa classe.

Definiamo, adesso, un altro oggetto della stessa classe:

In [6]:
```
B = Complex(r=1, i=4)
B
```

Out[6]:
```
(1+4j)
```

La nuova classe definita ci fornisce una ricetta per effettuare la moltiplicazione tra i suoi oggetti, che è proprio quella per i numeri complessi:

In [7]:
```
A*B
```

Out[7]:
```
(-10+11j)
```

Verifica: (2+3j)*(1+4j) = 2 + 8j + 3j +12j^2 = 2 + 11j - 12 = - 10 + 11j. L'operazione di sopra si può effettuare anche come segue:

In [8]:
```
A.__mul__(B)
```

Out[8]:
```
(-10+11j)
```

Questo modo di scrivere l'operazione di moltiplicazione esprime appieno la natura di *linguaggio orientato agli oggetti* di Python:

- `A` è un oggetto;
- `__mul__` è un metodo, di cui è fornito l'oggetto, che ci dice come far interagire tale oggetto con altri dello stesso tipo (cioè della stessa classe);
- forniamo (come argomento del metodo applicato con riferimento ad `A`) l'altro oggetto `B` con cui far interagire `A` tramite la data operazione.

In realtà, quando usiamo il simbolo di moltiplicazione `*` che ci è familiare, Python richiama l'uso della sua operazione intrinseca `__mul__`. Questo vale anche per la moltiplicazione in generale. Per esempio, quando effettuiamo l'operazione `2.0*3.0`, Python sta in realtà eseguendo quanto segue:

In [9]:
```
2.0.__mul__(3.0)
```

Out[9]:
```
6.0
```

Se cerchiamo di fare la somma di `A` e `B` otteniamo il seguente messaggio di errore:

In [10]:
```
A + B
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[10], line 1
----> 1 A + B

TypeError: unsupported operand type(s) for +: 'Complex' and 'Complex'
```

Infatti, non abbiamo definito l'operazione di addizione di elementi della classe `Complex`. Lo facciamo nella nuova classe seguente:

In [11]:
```
class Complex_bis(object):
    def __init__(self, r=0.0, i=0.0):
        self.re = r   # add attribute "re" to self
        self.im = i       # add attribute "im" to self
    def __mul__(self, other):
        return Complex_bis(self.re*other.re-self.im*other.im,
                           self.re*other.im+self.im*other.re)
    def __imul__(self, other):
        self.re = self.re*other.re-self.im*other.im
        self.im = self.re*other.im+self.im*other.re
        return self
    def __add__(self, other):
        return Complex(self.re+other.re,
                           self.im+other.im)
    def __repr__(self):
        return "({0}+{1}j)".format(self.re, self.im)
```

Ridefinaimo, quindi, `A` e `B` come oggetti della nuova classe appena creata:

In [12]:
```
A = Complex_bis(r=2, i=3)
B = Complex_bis(r=1, i=4)
print(A, B)
```

```
(2+3j) (1+4j)
```

Ora possiamo anche sommare tali oggetti:

In [13]:
```
A*B, A+B
```

Out[13]:
```
((-10+11j), (3+7j))
```

Infine, consideriamo il seguente numero complesso (un oggetto del tipo `complex` come realmente definito in Python):

In [14]:
```
a = 2 + 3j
```

e cerchiamo di sommarlo ad `A`:

In [15]:
```
a + A
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[15], line 1
----> 1 a + A

TypeError: unsupported operand type(s) for +: 'complex' and 'Complex_bis'
```

Otteniamo un errore perché i due oggetti appartengono a classi diverse e quindi sono visti come eterogenei: `A` è un oggetto di tipo `Complex_bis` (anche se ne abbiamo definito le proprietà come alcune delle proprietà dei numeri complessi) mentre `a` è un oggetto di tipo `complex`. Otterremmo lo stesso tipo di errore se cercassimo di sommare una stringa ad un numero intero:

In [16]:
```
a + "efg"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[16], line 1
----> 1 a + "efg"

TypeError: unsupported operand type(s) for +: 'complex' and 'str'
```

Eredità

Una classe può essere definita a partire da un'altra classe. Le proprietà (attributi e metodi) della classe madre passano alla classe figlia, che può avere nuove caratteristiche e anche sovrascrivere quelle vecchie.

Nel costruire la nuova classe a partire dalla precedente, si può usare il metodo super(), che chiama il costruttore della classe madre e ne passa gli argomenti alla classe figlia.

In [17]:
```
class Complex_tris(Complex_bis):
    def __init__(self, r=0.0, i=0.0):
        super().__init__(r,i)
    def __sub__(self, other):
        return Complex_tris(self.re-other.re,
                            self.im-other.im)
    def __repr__(self):
        if self.im >= 0:
            return "({0}+{1}j)".format(self.re, self.im)
        else:
            return "({0}{1}j)".format(self.re, self.im)
```

In [18]:
```
A = Complex_tris(r=2, i=3)
B = Complex_tris(r=1, i=4)
print(A, B)
```

```
(2+3j) (1+4j)
```

In [19]:
```
print(A*B, A+B, A-B)
```

```
(-10+11j) (3+7j) (1-1j)
```

In realtà, sopra abbiamo già usato l'ereditarietà, definendo le classi `Complex` e `Complex_bis` come figlie della classe `object`. Quest'ultima è una classe intrinseca di Python, definita come la base, o genitore, di tutte le classi. Essa definisce lo stato di base, di partenza che tutti gli oggetti devono avere. La sua indicazione è, comunque, non essenziale.

Esercizio

Creare una classe `persona` con gli attributi `nome` ed `età` e dotarla di metodi che ne stampino gli attributi.