04. Principi di OOP in Python

Corso di Python per il Calcolo Scientifico

Angelo Cardellicchio
angelo.cardellicchio@stiima.cnr.it

Outline

- Principi di OOP
  - Classi
  - Ereditarietà
  - Incapsulamento
  - Polimorfismo
- Classi in Python
  - Modificatori di accesso
  - Metodi
  - Metodi di classe
  - Metodi statici
  - Metodi astratti
  - Proprietà

Classi (1)

- La *programmazione orientata agli oggetti* è un paradigma che passa dal focus sulle **funzioni** (centrale nel C e nel paradigma procedurale/imperativo) a quello sui **dati**.
- Nella OOP, **tutto è un oggetto**.
- Possiamo creare degli oggetti di tipo **`Persona`** utilizzando adeguatamente il concetto di **classe**.

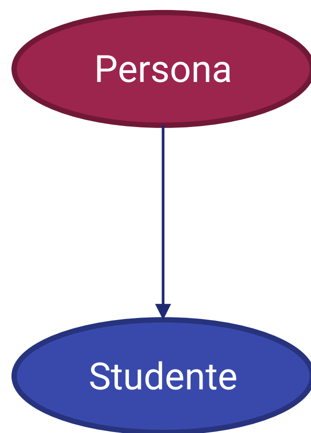
Classi (2)

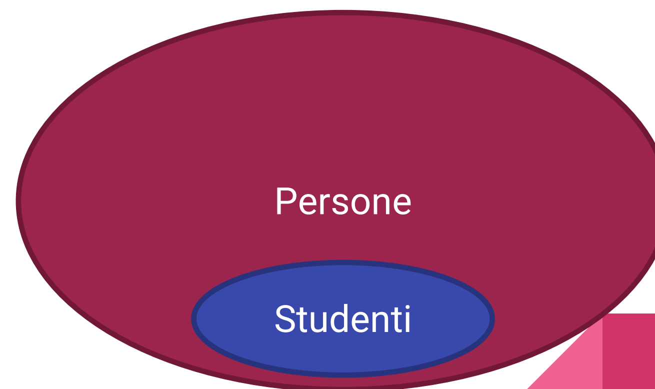
Attributo	Tipo
`nome`	Stringa
`cognome`	Stringa
`genere`	Stringa
`età`	Intero

- Possiamo anche definire dei **metodi**
- È importante sottolineare la differenza tra **classe** e **istanza**

Ereditarietà

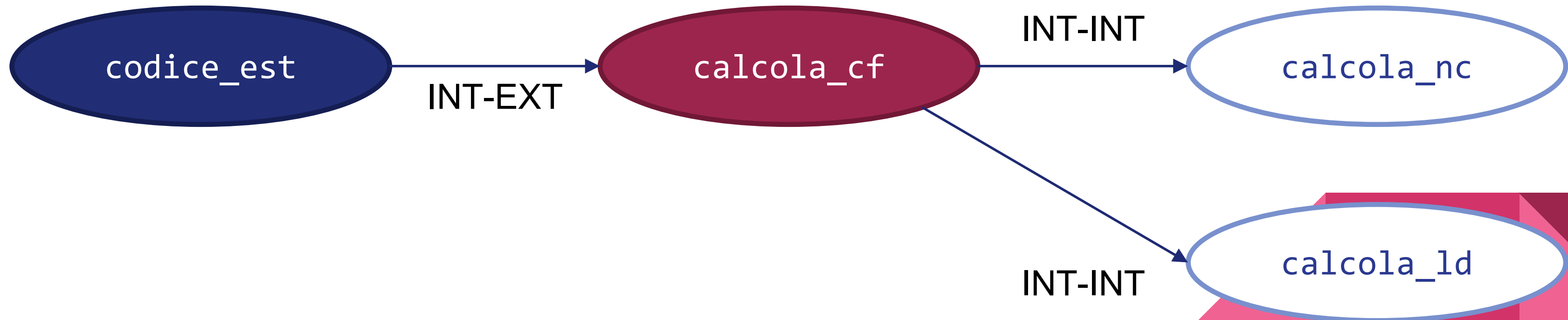
- **Ereditarietà**: possiamo definire una classe che ‘deriva’ da un’altra
- La classe **`Studente`** deriva da **`Persona`**





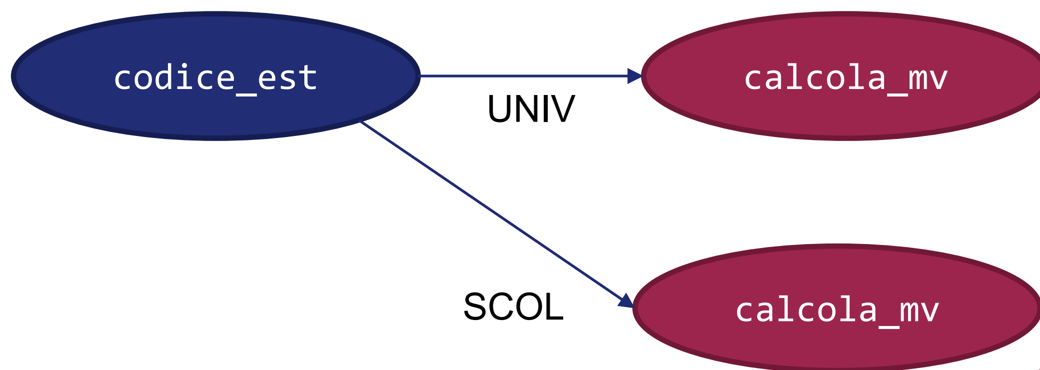
Incapsulamento

- **Incapsulamento**: il codice chiamante interagisce soltanto con un’interfaccia esterna
- **Risultato**: interfaccia stabile anche a seguito di cambio implementazione



Polimorfismo

- **Polimorfismo**: classi diverse possono avere implementazioni differenti dello stesso metodo (override)
- **Risultato**: interfaccia stabile



Classi in Python

- Le classi in Python si dichiarano usando la parola chiave **`class`**:

```
class NomeClasse(ClasseBase):
    # Attributi e metodi di classe...
```

- Python non prevede un costruttore, ma un metodo **`__init__`** per inizializzare i valori degli attributi:

```
class NomeClasse(ClasseBase):
    def __init__(self, *args, **kwargs):
        # ...
        self.arg_1 = arg_1
        # ...
```

Classi in Python

- I parametri **`args`** e **`kwargs`** rappresentano il concetto di **unpacking** (traducibile maccaronicamente con ‘spacchettamento’ di una lista e di un dizionario, rispettivamente).
- Ad esempio, possiamo creare la classe **`Persona`**:

```
class Persona(object):
    def __init__(self, nome, cognome, eta=18):
        self.nome = nome
        self._cognome = cognome
        self.__eta = eta
```

Modificatori di accesso

- Gli underscore prima del nome di un attributo vanno a definire diversi comportamenti.
- Un attributo anteceduto da un *singolo* underscore è inteso (per convenzione) ad uso interno per la sua classe classe.

```
self._cognome = cognome     # Membro "private" (hint)
```

- Un attributo anteceduto da un *doppio* underscore è sottoposto al **name mangling**.
    - In pratica, l'interprete cambia il nome della variabile, in modo che sia vi venga anteposto il nome della classe, allo scopo di evitare collisioni in caso di ereditarietà.

```
self.__eta = eta            # Name mangling
```

Name mangling

- Il name mangling è utile nei meccanismi di ereditarietà.
- Supponiamo di usare il name mangling per l'attributo identificativo nella classe Persona.

```
class Persona():
    def __init__(self, nome, cognome, identificativo):
        self.nome = nome
        self._cognome = cognome
        self.__identificativo = identificativo
```

- Proviamo a vedere le variabili ed i metodi di un oggetto di questa classe mediante la funzione **dir()**:

```
>>> p = Persona("Angelo", "Cardellicchio", 123)
>>> dir(p)
['_Persona__identificativo', ..., '_cognome', 'nome']
```

Name mangling

- Proviamo ora ad estendere la classe **`Persona`**.
- Usiamo in tal senso la classe **`Studente`**.

```
class Studente(Persona):
    def __init__(self, nome, cognome, identificativo):
        super().__init__(nome, cognome, identificativo)
        self.__identificativo = 123456
```

- Creiamo un altro oggetto ed utilizziamo la funzione **`dir()`**:

```
>>> s = Studente("Angelo", "Cardellicchio", 123)
>>> dir(s)
['_Persona__identificativo', '_Studente__identificativo', ...,
      '_cognome’, 'nome']
```

Metodi

- Sono sintatticamente quasi identici alle classiche funzioni Python.
- Accettano come primo parametro la parola chiave **`self`**, che indica che si riferiscono all'istanza attuale della classe.

```
def metodo(self, *args, **kwargs):
    pass
```

- Il riferimento a **self** non va indicato quando si chiama il metodo dall'esterno della classe, ma ci si limita a richiamarlo dall'istanza stessa.

```
p = Persona()          # p è un'istanza di Persona
p.metodo(parametro)    # richiamo il metodo dall'istanza
```

Metodi di classe

- I **metodi di classe** sono contraddistinti dal *decorator* **`@classmethod`**.
- Sono metodi richiamabili sull’intera classe.
- Non hanno il riferimento all’istanza (**`self`**), ma all’intera classe (**`cls`**).
- Sono utilizzati per definire particolari tipi di costruttori/builder.

```
@classmethod
def builder_stringa(cls, stringa: str):
    nome, cognome, eta = stringa.split(' ‘)
    return Persona(nome, cognome, eta)
```

Metodi statici

- I **metodi statici** sono contraddistinti dal *decorator* **`@staticmethod`**.
- Sono metodi che non si applicano solo su un'istanza o sulla classe, ma risultano essere di applicabilità più generica.
- Di solito, sono all'interno di una classe per coerenza logica e funzionale.

```
@staticmethod
def nome_valido(nome):
    if len(nome) < 2:
        return False
    else:
        return True
```

Metodi astratti

- L'ereditarietà si ottiene **specializzando** una classe madre.
- Ad esempio, uno studente è un caso **speciale** di persona, ma il contrario non è vero.
- Si dice che lo studente **eredita** attributi e metodi della persona, definendone di altri.
- Le classi ed i metodi **astratti** sono dei 'prototipi' di metodi, definiti in una classe madre, che saranno implementati nelle classi figlie.
- Sono contraddistinti dalla parola chiave **`ABC`** e dal decorator **`@abstractmethod`.**
- *Vedremo un esempio tra gli esercizi.*

Proprietà

- Python offre un modo alternativo per implementare gli attributi di una classe.
- In particolare, usando le **proprietà**, contraddistinte dal *decorator* **`@property`**, possiamo fare in modo da definire metodi **impliciti** di accesso e modifica dei dati.
- Risultano essere estremamente efficaci nel caso di implementazioni complesse.

Esercizio

- ***Esercizio 1:*** *scrivere una classe* **`Persona`** *utilizzando i concetti visti in precedenza.*
- ***Esercizio 2****: creiamo due classi. La prima è la classe* **`Quadrato`***, che modella tutti i quadrati, la seconda è la classe* **`Cerchio`***, che modella tutti i cerchi.*

Domande?