# Object Oriented Programming

Prof. Ing. Loris Penserini, PhD

https://orcid.org/0009-0008-6157-0396

# Classi Astratte e Interfacce

# Classi Astratte

La classe astratta, è una classe in cui uno o più metodi non sono implementati. Per cui, da una classe astratta **non** si può creare un oggetto.

Similmente alle interfacce, servono a definire delle **linee guida per creare le sottoclassi** che ne ereditano le proprietà. Cioè una sottoclasse che eredita da una classe astratta, ne deve implementare i metodi astratti, se vuole istanziare oggetti.

Nel caso una sottoclasse, che eredita da una classe astratta, non implementa tutti i metodi di quest'ultima, deve essere anch'essa dichiarata astratta.

- Possono contenere istruzioni, proprietà e metodi come le normali classi
- Possono utilizzare i modificatori di visibilità come le normali classi
- Possono avere il costruttore
- Vale l'ereditarietà come per le classi

# Classi Astratte in Python

Python usa il modulo **abc** (Abstract Base Classes) per creare classi astratte, cioè classi che definiscono metodi ma non li implementano.

Una **classe astratta** è una classe che:

- **non può essere istanziata direttamente**;
- può contenere sia **metodi astratti** (da implementare nelle sottoclassi) sia **metodi concreti** (già implementati e condivisi);
- serve come *modello* per altre classi.

Le classi derivate sono obbligate a implementare quei metodi, proprio come un'interfaccia!

# Esempio di Classi Astratte in Python

OOP > abstract > forma.py > ...

```python
from abc import ABC, abstractmethod

class Forma(ABC):
    @abstractmethod
    def area(self):
        pass

    def descrivi(self):
        print("Sono una forma geometrica.")
```

OOP > abstract > cerchio.py > ...

```python
from forma import Forma

class Cerchio(Forma):
    def __init__(self, raggio):
        self.raggio = raggio

    def area(self):
        return 3.14 * self.raggio ** 2
```

## main.py

OOP > abstract > main.py

```python
from cerchio import Cerchio

print(Cerchio(10).area())

```

## output

# Interfaccia

In Python il concetto di interfaccia esiste, ma non nello stesso modo "formale" che trovi in Java o PHP.

In Python, si può ottenere lo stesso comportamento di un'interfaccia in due modi principali:

- Utilizzando le classi astratte: cioè classi che definiscono metodi ma non li implementano, per cui le classi derivate sono obbligate a implementare quei metodi, proprio come un'interfaccia!
- Utilizzando il «duck-typing»: nessuna interfaccia formale, conta solo che l'oggetto *abbia* il metodo richiesto.

# Esempio di suo di «interfaccia»

OOP > interfacce > veicolo.py > ...

```python
from abc import ABC, abstractmethod

# Classe astratta = interfaccia
class Veicolo(ABC):

    @abstractmethod
    def avvia(self):
        pass

    @abstractmethod
    def ferma(self):
        pass

```

OOP > interfacce > auto_bici.py > ...

```python
from veicolo import Veicolo

class Auto(Veicolo):
    def avvia(self):
        print("Accendo il motore dell'auto")

    def ferma(self):
        print("Spengo il motore dell'auto")


class Bicicletta(Veicolo):
    def avvia(self):
        print("Inizio a pedalare")

    def ferma(self):
        print("Smetto di pedalare")

```

# Usiamo un'Interfaccia: «main.py»

OOP > interfacce > main.py > ...

```python
from auto_bici import Auto, Bicicletta

a = Auto()
b = Bicicletta()

a.avvia()
b.ferma()
```

**output**

```
Nome, Cognome e Età dello STUDENTE_1:
Nome: Mario Rossi - Età: 18 anni
```

Se nella classe **auto_bici.py** si prova a non implementare un metodo, Python genererà un "TypeError":

Cannot instantiate abstract class "Auto"

"Veicolo.ferma" is not implemented

**Quindi funziona esattamente come un'interfaccia.**

# Interfaccia informale (duck typing)

Python è dinamico: non serve per forza un'interfaccia formale. Se due classi hanno gli stessi metodi, puoi usarle in modo intercambiabile.

Questo è il famoso principio del "duck typing": "se cammina come un'anatra e starnazza come un'anatra, allora è un'anatra".

OOP > duck_typing > cane.py > ...

```
1  class Cane:
2      def parla(self):
3          print("Bau!")
4
```

OOP > duck_typing > gatto.py > ...

```
1  class Gatto:
2      def parla(self):
3          print("Miao!")
4
```

# Esempio «main.py» con duck typing

OOP > duck_typing > main.py > ...

```python
from cane import Cane
from gatto import Gatto

# Funzione che accetta "qualunque cosa" con il metodo parla()
def fai_parlare(animale):
    animale.parla()

fai_parlare(Gatto())
fai_parlare(Cane())
```

output

```
Miao!
Bau!
```

# INCAPSULAMENTO

# INCAPSULAMENTO

E' una proprietà della OOP che facilita a seguire la filosofia dell'ingegneria del software dell'usabilità e della modularità delle applicazioni.

In pratica, nella OOP con l'utilizzo dell'incapsulamento, si 'aggregano' all'interno di un oggetto i dati e le azioni che lo interessano, che diventano quindi elementi **visibili e gestiti** solo dall'oggetto stesso (o da una sua classe), mentre si rendono **pubblici** solo metodi (o attributi) che servono all'oggetto per poter essere riutilizzato in altri contesti applicativi.

**Quindi, si parla spesso di limitare l'accesso diretto agli elementi di un oggetto, ovvero di occultamento delle informazioni (*information hiding*).**

# INCAPSULAMENTO

La sua funzione principale è quella di proteggere lo stato interno dell'oggetto, permettendo l'accesso solo attraverso metodi controllati, come **getter** e **setter**.

Questo approccio realizza il principio dell'**information hiding** ("nascondere le informazioni"), cioè l'idea che i dettagli di implementazione non debbano essere visibili o accessibili dall'esterno.

**In questo modo, l'oggetto espone solo ciò che serve per interagirvi, mentre mantiene private le parti interne che potrebbero cambiare o compromettere la consistenza dei dati.**

# Caratteristiche

L'incapsulamento consente quindi di:

- migliorare la sicurezza e l'affidabilità del codice, impedendo modifiche indesiderate agli attributi;
- semplificare la manutenzione, poiché le modifiche interne non influenzano il codice esterno;
- e promuovere la modularità, favorendo la progettazione di componenti indipendenti e riutilizzabili.

**In sintesi, l'incapsulamento permette di trattare l'oggetto come una "scatola nera": l'utente ne conosce il comportamento esterno, ma non ha bisogno di sapere come funziona al suo interno.**

# Utilizzo

La sua funzione principale è quella di **proteggere lo stato interno dell'oggetto**, permettendo l'accesso solo attraverso metodi controllati, come *getter* e *setter*.

Questo approccio realizza il principio dell'**information hiding** ("nascondere le informazioni"), cioè l'idea che i dettagli di implementazione **non debbano essere visibili o accessibili dall'esterno**.

In questo modo, l'oggetto espone solo ciò che serve per interagirvi, mentre mantiene private le parti interne che potrebbero cambiare o compromettere la consistenza dei dati.

# Modificatori di visibilità in Python

Python non ha modificatori espliciti come **public**, **private** o **protected** (tipici di Java o PHP), ma **usa una convenzione** sui nomi per indicare il livello di accesso.

| Livello | Sintassi | Accessibilità | Descrizione |
|---|---|---|---|
| **Pubblico** | nome | Accessibile ovunque | attributi/metodi visibili all'esterno |
| **Protetto** | _nome | Convenzione: "non toccare dall'esterno" | ancora accessibile, ma "interno" |
| **Privato** | __nome | Nascosto tramite name mangling | non accessibile direttamente da fuori |

# Il «name mangling»

Python non ha modificatori espliciti, per cui il **name mangling** (letteralmente "modifica del nome") è una tecnica che Python usa per rinominare internamente gli attributi che iniziano con due underscore (__).

In pratica, Python trasforma automaticamente il nome dell'attributo per evitare conflitti e rendere difficile l'accesso diretto dall'esterno.

```
class Persona:
    def __init__(self, nome):
        self.__nome = nome  # attributo "privato"
```

```
p = Persona("Luca")
print(p.__nome)
```

```
AttributeError: 'Persona' object has no attribute '__nome'
```

# Il name mangling

Python rinomina l'attributo così:

**_Persona__nome**

cioè aggiunge il nome della classe davanti all'attributo. Quindi si può ancora accedervi tecnicamente (anche se non si dovrebbe fare):

```
print(p._Persona__nome)    # funziona
```

**Per cui in Python l'incapsulamento non fornisce una protezione dei dati veramente sicura.** Puoi pensare al name mangling («storpiatura del nome») come a una "cifratura leggera del nome": non è segreto davvero, ma serve a dire: "questo è solo per uso interno, non toccarlo".

# Project work 3

Riutilizzate i metodi del project work 2, per la classe «Persona» (getNome, getCognome), questi devono essere «protected». Per cui, accessibili dalle classi figlie. Mentre, le variabili di «Persona» (nome e cognome), devono diventare private, cioè accessibili solo dalla classe stessa.

Poi dalla classe «Studente» implementate il metodo «getStato()» che legge il nome e il cognome dalla classe «Persona» e restituisce i valori al chiamante.

# PW 3: Soluzione

Le variabili sono «private», cioè accessibili solo dalla classe che le ospita

I metodi sono «protected», cioè accessibili solo dalle classi figlie

OOP > oop_pw3 > persona.py > Persona > get_pag_benvenuto

```python
class Persona(object):
    # Costruttore: viene chiamato quando si crea un nuovo oggetto
    def __init__(self, nome, cognome, eta, interessi):
        # Attributi dell'oggetto (proprietà)
        self.__nome = nome
        self.__cognome = cognome
        self.__eta = eta
        self.__interessi = interessi
        self.saluto = f"Buongiorno, sono {nome} {cognome}"

    # Metodo che restituisce la frase di saluto
    def get_pag_benvenuto(self):
        self.saluto = f"Buongiorno, sono {self.__nome} {self.__cognome}"
        return self.saluto

    def _getNome(self):
        return self.__nome

    def _getCognome(self):
        return self.__cognome

    def _getEta(self):
        return self.__eta

    def _getInteressi(self):
        return self.__interessi
```

# PW 3: Soluzione

OOP > oop_pw3 > studente.py > Studente > getPagBenvenutoStud

```python
from persona import Persona

class Studente(Persona):
    def setIndStudio(self,ind_studio):
        self.ind_studio = ind_studio

    def getPagBenvenutoStud(self):
        self.saluto_persona = self.get_pag_benvenuto()
        return self.saluto_persona + ", e frequento " + self.ind_studio

    def getNomeStud(self):
        return self._getNome()

    def getCognomeStud(self):
        return self._getCognome()

    def getEtaStud(self):
        return self._getEta()

    def getInteressiStud(self):
        return self._getInteressi()
```

# PW 3: Soluzione

OOP > oop_pw3 > main.py > ...

```python
from studente import Studente

nome = "Mario"
cognome = "Rossi"
eta = "18"
interessi = "Ciclismo"

Studente1 = Studente(nome,cognome,eta,interessi)
Studente1.setIndStudio("Sitemi Informativi Aziendali")
print("Nome, Cognome e Età dello STUDENTE_1:")
print("Nome: " + Studente1.getNomeStud() + " " + Studente1.getCognomeStud() +
      " - Età: " + Studente1.getEtaStud() + " anni")
print()

# Utilizzo di metodi "protected" per accedere a variabili "private", della classe "Persona"
print(Studente1._getNome() + " " + Studente1._getCognome() + " - " + Studente1._getEta() + " anni")
```

output

```
Nome, Cognome e Età dello STUDENTE_1:
Nome: Mario Rossi - Età: 18 anni

Mario Rossi - 18 anni
```

# GRAZIE!