Programmazione a oggetti

I crediti fotografici delle immagini presenti in questo PowerPoint sono riportati all'interno del volume *informatica per progetti – Programmare in Python*, di M. Fiore, A. Zanga, ISBN 9788808899477

Programmazione a oggetti - OOP

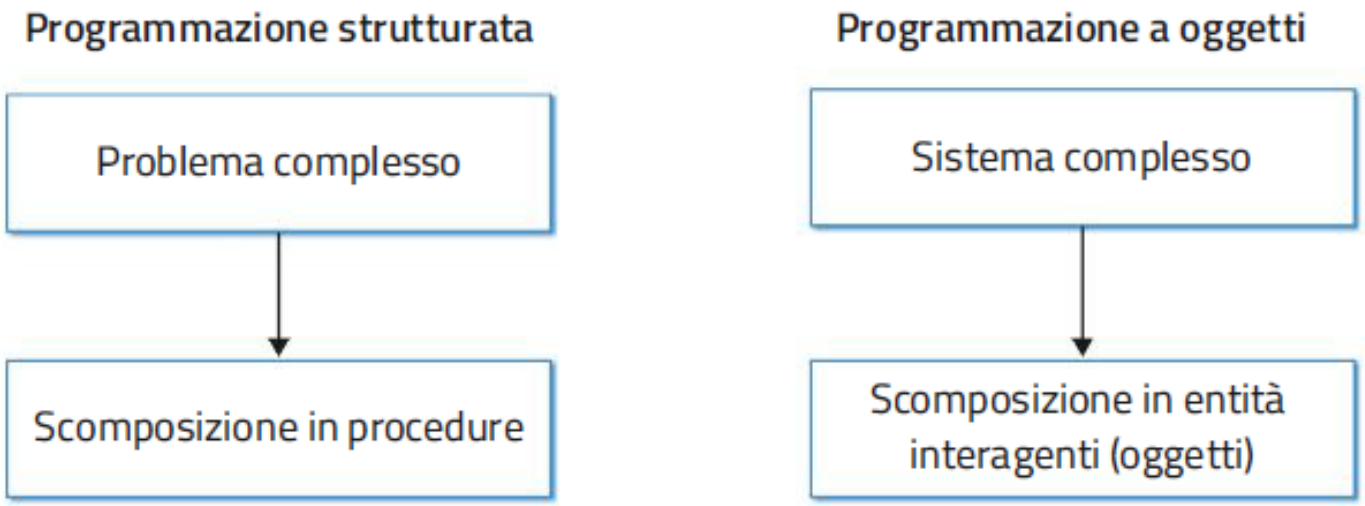
La programmazione orientata agli oggetti OOP è un paradigma di programmazione che si basa sul concetto di oggetto.

> Un **oggetto** rappresenta un'entità della realtà che vogliamo scrivere in codice.

Gli oggetti sono un'astrazione di entità e concetti e contengono tutte le informazioni sullo stato di quell'entità che sono utili per risolvere il problema.
È possibile anche associare a un oggetto le operazioni che è in grado di compiere.

Programmazione a oggetti vs programmazione strutturata

La programmazione a oggetti non è un paradigma di programmazione alternativo alla programmazione strutturata, ma è complementare a essa:



Vantaggi della OOP

La programmazione orientata agli oggetti offre alcuni vantaggi che la programmazione strutturata non possiede:

- Facilità di lettura e comprensione: rappresentare entità del mondo reale come oggetti ci permette di seguire con più facilità la semantica del codice.
- Rapidità della manutenzione: modificare un oggetto è più facile rispetto a dover cercare variabili e funzioni nel codice.
- Robustezza: raggruppare variabili e funzioni comuni nella stessa regione di codice riduce la probabilità di commettere errori di implementazione.
- Riusabilità: definire il comportamento di un oggetto ci permette di scrivere meno codice e riutilizzare quello già esistente.

Oggetti, classi, istanze

Un **oggetto** è definito attraverso i suoi attributi e metodi:

- Un attributo è una variabile che caratterizza lo stato di un oggetto.
- Un metodo è una funzione che definisce il comportamento di un oggetto.

Una **classe** è una descrizione astratta che definisce attributi e metodi di un oggetto.
In pratica, una classe è l'implementazione in codice di un oggetto, che a sua volta è un'astrazione, una rappresentazione semplificata di un'entità del problema che vogliamo risolvere.

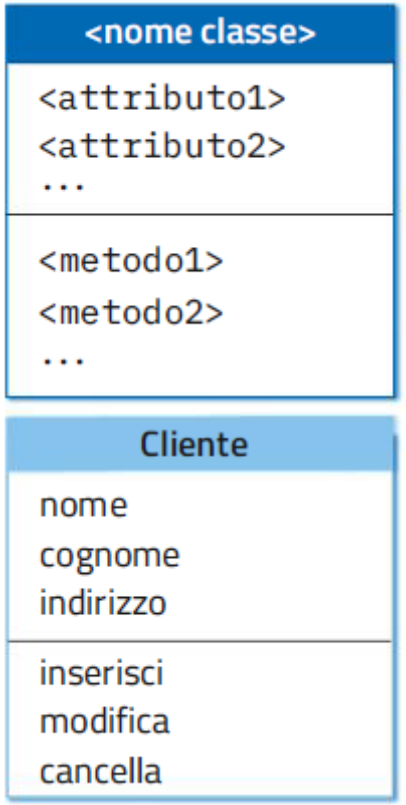
Quando assegniamo dei valori agli attributi di una classe otteniamo un'istanza.
L'**istanza** di un oggetto è un esemplare della classe con specifici valori degli attributi.

Rappresentazione di una classe con UML

Il diagramma delle classi è una rappresentazione grafica del codice che dovremo implementare.
I simboli utilizzati nel diagramma seguono il Linguaggio di Modellazione Unificato, in inglese *Unified Modeling Language* (UML).

Per convenzione, una classe è rappresentata come una tabella composta da tre parti:

- il nome della classe,
- la lista degli attributi
- la lista dei metodi.



Dichiarazione di una classe

La dichiarazione di una classe in Python è composta da:
1. la parola chiave class;
2. il nome della classe;
3. la classe base, inserita tra parentesi tonde seguite dai due punti;
4. gli attributi (opzionali), indentati a destra rispetto alla prima riga;
5. i metodi (opzionali), indentati a destra rispetto alla prima riga.
In Python esiste una classe base chiamata object da cui derivano tutte le altre classi.

```
1  class NomeClasse(object):  #Dichiarazione della classe
2      pass                   #Corpo della classe (vuoto per ora)
```

Per costruire un'istanza della classe è sufficiente scrivere il suo nome seguito da una coppia di parentesi tonde.

```
1  class NomeClasse(object):  #Dichiarazione della classe
2      pass                   #Corpo della classe (vuoto per ora)
3
4  #main
5  istanza = NomeClasse()     #Creazione istanza
6  print(istanza)
```

Metodi fondamentali

Fase del ciclo di vita	Metodo	Descrizione
Costruzione	`__new__`	• Crea una nuova istanza non inizializzata • Viene chiamata in fase di creazione • Se non è definito, viene chiamato quello della classe base `object`
Inizializzazione	`__init__`	• Inizializza l'istanza appena creata • Viene chiamata dopo `__new__` • Il suo scopo è quello di assegnare un valore agli attributi dell'istanza • Se non è definito, viene chiamato quello della classe base `object`
Distruzione	`__del__`	• Distrugge l'istanza • Viene chiamato implicitamente quando l'istanza non è più visibile, oppure esplicitamente con `del` • Se non è definito, viene chiamato quello della classe base `object`

In generale, non definiremo né il costruttore né il distruttore, ma solo l'inizializzatore, in modo da delegare a object la gestione di queste due fasi del ciclo di vita della classe.

Esempio di classe

```
1  class Cliente(object):
2
3      #Attributi
4      nome = None
5      cognome = None
6      indirizzo = None
7
8      #Inizializzatore
9      def __init__(self, nome, cognome, indirizzo):
10         self.nome = nome
11         self.cognome = cognome
12         self.indirizzo = indirizzo
13
14 #main
15 cliente = Cliente("Mario", "Rossi", "Via Monte Napoleone")
16 print(f"Nome: {cliente.nome}, Cognome: {cliente.cognome}")
17 print(f"Indirizzo: {cliente.indirizzo}")

   Nome: Mario, Cognome: Rossi
   Indirizzo: Via Monte Napoleone
```

Definizione di attributi

Riferimento all'istanza

Notazione punto

Creazione dell'istanza

Visualizzazione di una istanza

La visualizzazione è un'operazione così importante che Python prevede un metodo speciale __str__ che consente di restituire una stringa formattata da passare alla funzione print.

```
...
14      #Visualizzazione
15      def __str__(self):
16          s = f"Nome: {self.nome}, Cognome: {self.cognome}\n"
17          s += f"Indirizzo: {self.indirizzo}"
18          return s
19
20  #main
21  cliente = Cliente("Mario", "Rossi", "Via Monte Napoleone")
22  print(cliente)

    Nome: Mario, Cognome: Rossi
    Indirizzo: Via Monte Napoleone
```

Metodi di classe e di istanza

I **metodi di classe** sono funzioni definite all'interno di una classe e sono applicabili alla classe stessa. Utilizzano cls per indicare il riferimento alla classe.
Per definire un metodo di classe dobbiamo utilizzare la funzione classmethod.

```
1 class Classe(object):
2       def metodo_classe(cls):                 #Dichiarazione metodo
3             print("Metodo di classe")         #Corpo metodo
4       #Sovrascrittura con metodo di classe
5       metodo_classe = classmethod(metodo_classe)
```

I metodi di classe sono invocabili sia dalla classe sia su una sua istanza. Per questo motivo sono particolarmente utili quando il metodo non dipende dallo stato dell'istanza, cioè quando non facciamo riferimento agli attributi dell'istanza.

I **metodi di istanza** sono funzioni definite all'interno di una classe e sono invocabili solo sull'istanza della classe. Utilizzano self per indicare il riferimento all'istanza.

Attributi di classe e di istanza

Un **attributo di classe** è un attributo il cui valore è noto durante la dichiarazione della classe.
Un attributo di classe è condiviso da tutte le istanze della classe e viene automaticamente aggiunto come attributo di istanza con il valore che ha al momento della creazione.

```
>>> class Classe(object):                #Dichiarazione classe
>>>     attributo_classe = 0              #Dichiarazione attributo di classe
>>>
>>> print(Classe.attributo_classe)        #Visualizzazione attributo di classe

    0

>>> istanza_1 = Classe()                 #Creazione nuova istanza
>>> print(istanza_1.attributo_classe)     #Visualizzazione attributo di classe

    0

>>> Classe.attributo_classe = 10         #Modifica dell'attributo di classe
>>> istanza_2 = Classe()                 #Creazione nuova istanza
>>> print(istanza_2.attributo_classe)     #Visualizzazione attributo di classe

    10
```

Un **attributo di istanza** è un attributo il cui valore è noto durante l'inizializzazione di un'istanza.
A differenza di un attributo di classe, un attributo di istanza definisce lo stato dell'istanza, ossia l'insieme dei valori delle variabili che ci permettono di differenziare le istanze tra loro.

Visibilità

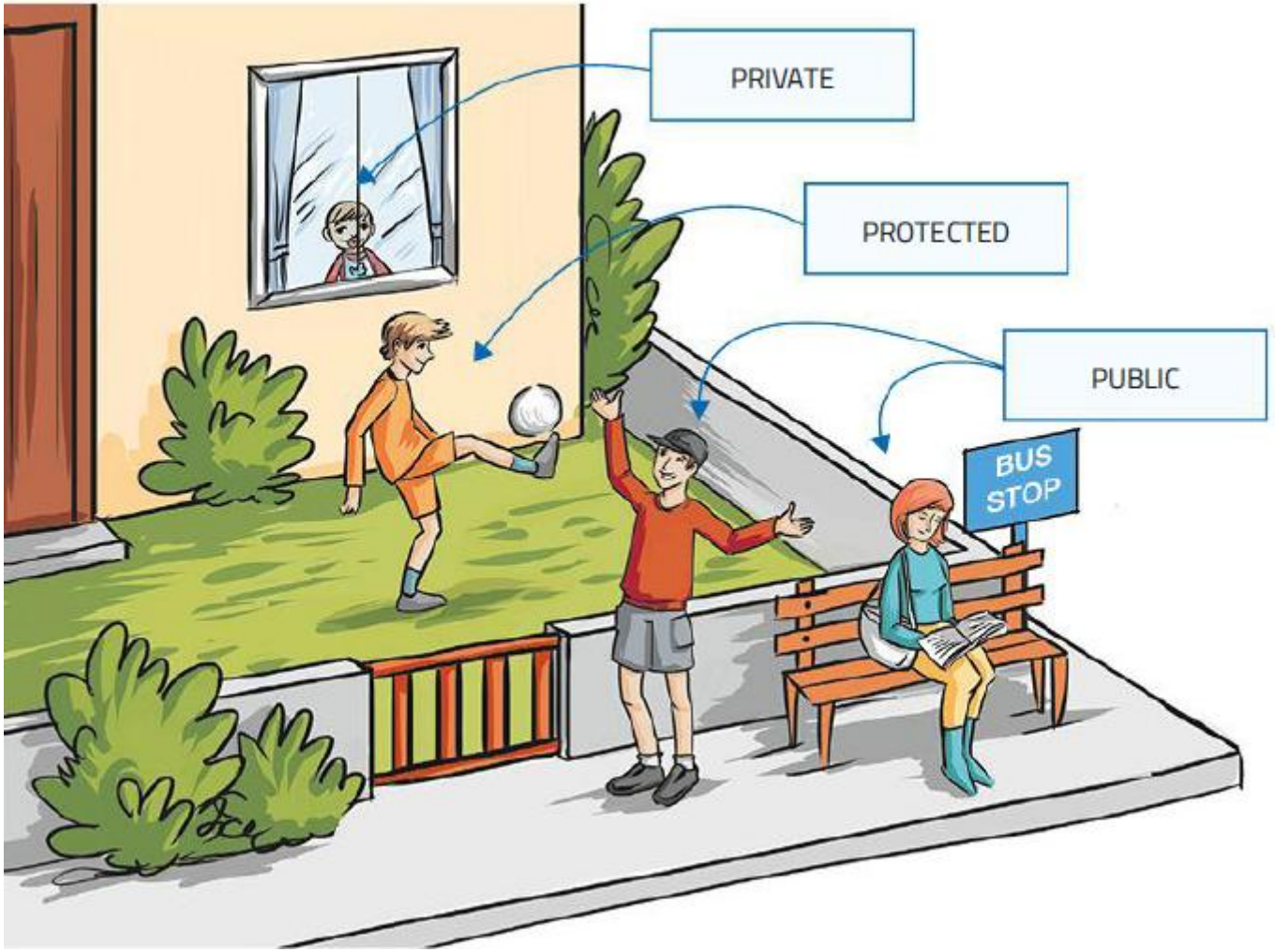
Nella programmazione a oggetti il concetto di visibilità dell'informazione è fondamentale.

Si distinguono tre livelli di accesso:

- public contiene i membri a cui si può accedere dall'esterno della classe;
- private contiene i membri ai quali è possibile accedere solo dall'interno della classe;
- protected contiene membri ai quali si può accedere anche all'interno di classi derivate della stessa.

I livelli di visibilità danno la possibilità allo sviluppatore di controllare l'informazione che vuole esporre all'esterno della classe.

Livelli di visibilità



Visibilità in Python

Mentre in altri linguaggi di programmazione i membri di una classe sono considerati privati, in Python non c'è una distinzione nella sintassi del linguaggio e di base **i membri sono considerati pubblici**.
Esiste una convenzione per cui gli attributi da considerare privati sono preceduti da un doppio *underscore*.

Ma perché distinguere tra membri pubblici e privati?

I motivi sono due.

1. Per imporre vincoli concettuali: in caso di codice scritto da più sviluppatori è utile poter comunicare l'intenzione di esporre o meno alcuni dettagli di implementazione.
2. Per non fornire accesso ad alcune parti della classe: nel caso di sviluppo di applicazioni commerciali potrebbe essere necessario nascondere i dettagli dell'implementazione specifica di alcune parti del codice.

Incapsulamento

Se i membri privati non sono visibili all'esterno della classe, al contrario i membri pubblici lo sono e definiscono quella che viene chiamata interfaccia.

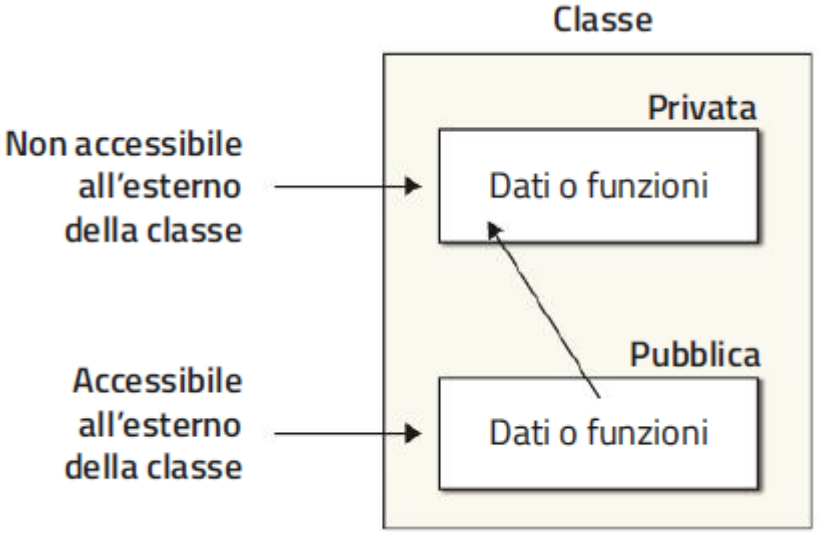
L'**interfaccia di una classe** indica l'insieme dei membri pubblici, cioè l'insieme delle informazioni e funzionalità utilizzabili dalle istanze della classe.
Rendere espliciti, cioè pubblici, i membri di una classe significa esporre in modo visibile le possibili interazioni con le istanze, in modo che tutti gli sviluppatori sappiano quali azioni possono e non possono fare con quelle istanze.

Il termine **incapsulamento** indica la qualità degli oggetti di poter incorporare al loro interno sia gli attributi sia i metodi, cioè le caratteristiche e i comportamenti dell'oggetto

Information hiding

Se combiniamo visibilità e incapsulamento possiamo attuare quello che viene chiamato ***information hiding*** che indica una particolare modalità di implementazione di un oggetto che rende disponibili all'esterno solo alcune funzionalità.
I dettagli sulle caratteristiche e la struttura dell'oggetto sono nascosti all'interno.

Distinguere tra membri pubblici e privati ci permette di utilizzare queste qualità che sono alla base della programmazione a oggetti, garantendo una maggiore capacità di astrazione e flessibilità di implementazione.

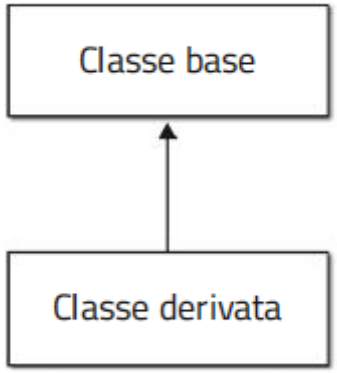


Ereditarietà

L'ereditarietà rappresenta la capacità di creare nuove classi a partire da classi già esistenti: le nuove classi create rappresentano una versione modificata della classe di partenza che viene definita superclasse o classe base.
La nuova classe eredita tutti gli attributi e i metodi della classe base e può essere arricchita con nuovi attributi e nuovi metodi.

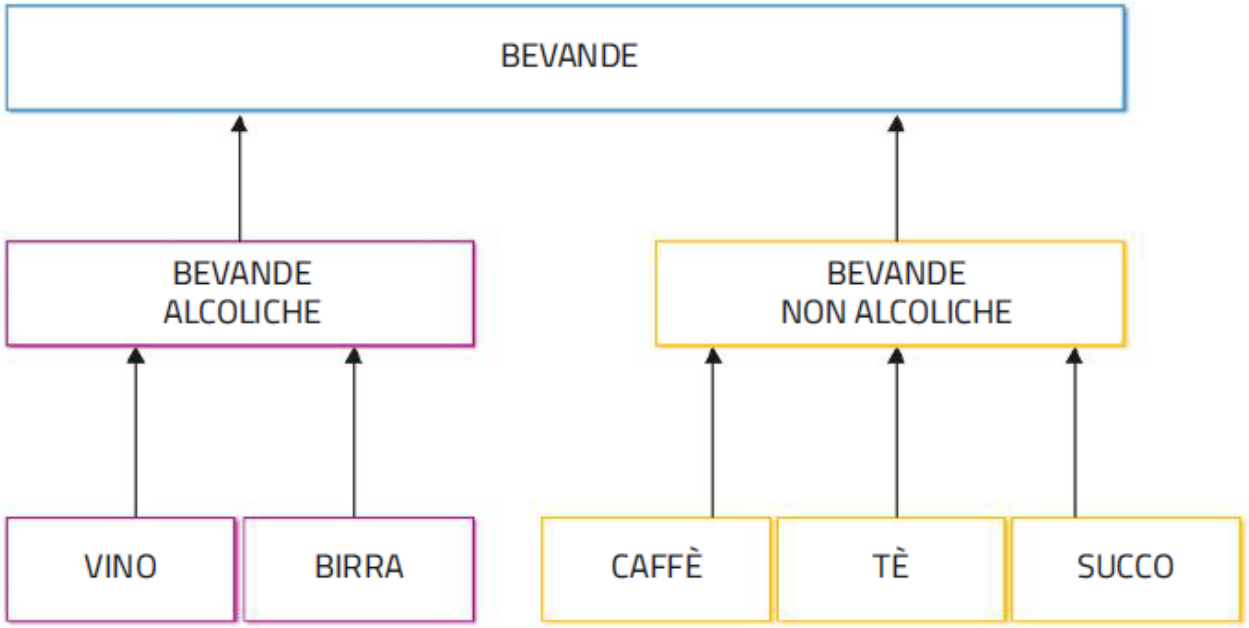
Si definisce **sottoclasse** o classe derivata la classe che viene tratta da un'altra classe con il meccanismo di ereditarietà.
Si definisce **superclasse** o classe base la classe generatrice di una sottoclasse.

Nel diagramma UML delle classi, la classe derivata si indica con una freccia che la collega alla classe base.



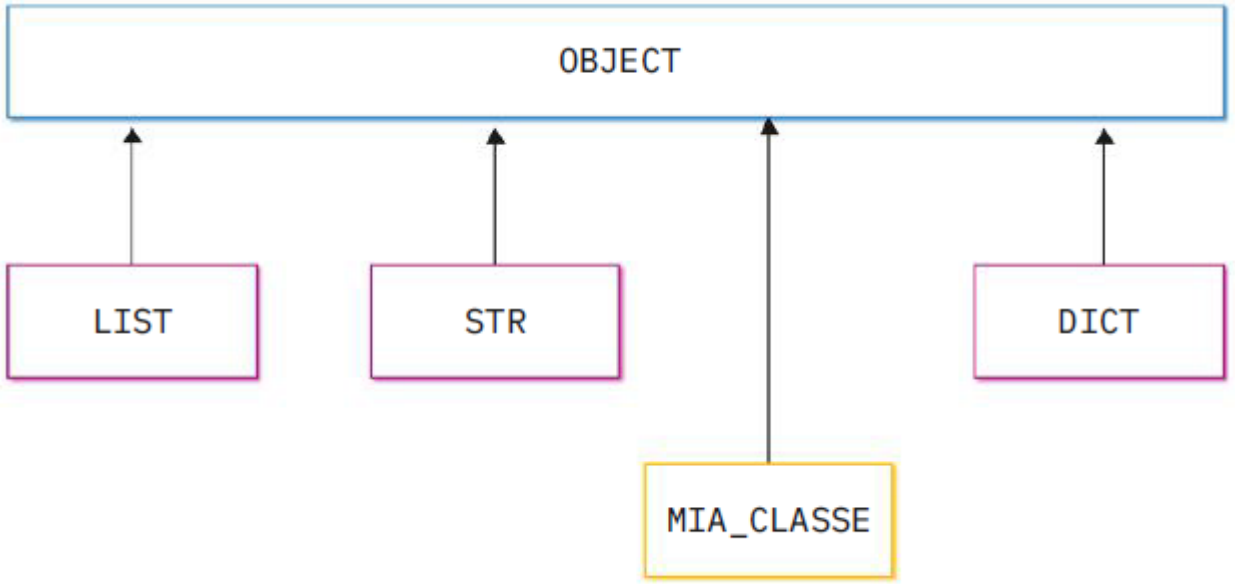
Gerarchia di classi

Il meccanismo di ereditarietà consente di creare *classi figlie* a partire da *classi genitori* permettendo il riutilizzo del codice e rendendo gli script più modulari e organizzati; consente anche la creazione di più classi a livelli differenti, permettendo così la costruzione di una gerarchia di classi, in cui le classi derivate possono estendere o modificare i comportamenti delle classi base.



Classe object

La classe object è la classe base di Python da cui derivano tutti i tipi di dati built-in: dalla classe object derivano anche tutte le classi definite in Python, utilizzando il meccanismo di ereditarietà.



Dichiarazione di classe derivata

L'ereditarietà permette di creare nuove classi in due modalità:

- per estensione;
- per ridefinizione.

Una classe **derivata per estensione** indica una classe che aggiunge ai metodi e agli attributi della classe base nuovi attributi e metodi che vengono definiti nella classe derivata.

Una classe **derivata per ridefinizione** indica una classe che modifica o ridefinisce i metodi ereditati dalla classe base.

```
class BevandeAnalcoliche(Bevande):
    #attributi
    ….
    #metodi
    ….
```

Le classi derivate ereditano attributi e metodi dalla superclasse e per poterli utilizzare è necessario fare riferimento a essi usando la funzione super(), che consente di fare riferimento ad attributi e metodi (anche il costruttore e l'inizializzatore) della classe da cui deriva.

Polimorfismo

Con l'ereditarietà è possibile creare nuovi classi derivandole da altre già esistenti, per estensione della classe base oppure per ridefinizione; il polimorfismo rappresenta per un oggetto la possibilità di assumere forme diverse in base alla sua implementazione specifica.

Il **polimorfismo degli oggetti** rappresenta la capacità degli oggetti di assumere comportamenti diversi in base a una diversa implementazione dei suoi metodi.

Il polimorfismo facilita il riutilizzo del software; un esempio è la possibilità di usare lo stesso metodo di una classe, ma con tipi diversi di dati. Facilita inoltre la scrittura di codice in modo estensibile, poiché l'aggiunta, a una gerarchia di classi già predisposta, di una nuova classe che implementa i metodi in modo differente non influenza il codice precedentemente scritto.

Overriding

La derivazione di una classe che effettua la ridefinizione dei metodi della classe base si chiama **overriding** (o sovrascrittura del metodo).
L'overriding non cambia il nome del metodo né il numero dei parametri (la sua firma), ma modifica le sue istruzioni.
L'**overriding** è un polimorfismo che prevede di sovrascrivere nella classe derivata un metodo della classe base, cambiando le istruzioni e dunque cambiando il suo comportamento.

```python
class Libro(object):
    #attributi
    titolo = ""
    autore = ""
    anno_pubblicazione = 0
    #metodi
    def __init__(self, titolo, autore, anno_pubblicazione):
        self.titolo = titolo
        self.autore = autore
        self.anno_pubblicazione = anno_pubblicazione

    def descrivi(self):
        '''usato per descrivere l'istanza del libro'''
        return f"{self.titolo} scritto da {self.autore}, pubblicato nel {self.anno_pubblicazione}"

```

```python
class Fantasy(Libro):
    #attributi aggiuntivi
    genere = ""
    #metodi
    def __init__(self, titolo, autore, anno_pubblicazione, genere):
        super().__init__(titolo, autore, anno_pubblicazione)
        self.genere = genere

    def descrivi(self):
        return f"{super().descrivi()}, genere: {self.genere}"

```

overriding nel metodo descrivi della classe Fantasy: anche la classe Libro ha un metodo che si chiama descrivi, ma il nuovo metodo descrivi della classe Fantasy lo sovrascrive aggiungendo al suo comportamento la possibilità di stampare anche il nuovo attributo genere che abbiamo aggiunto alla classe derivata.

Overloading

Un secondo tipo di polimorfismo è l'overloading, cioè la possibilità di definire più di una versione di una funzione o di un operatore con lo stesso nome ma con diversi tipi di parametri o un numero diverso di parametri.
In altre parole, le funzioni o gli operatori possono avere lo stesso nome ma comportarsi in modo diverso in base al tipo o al numero di parametri passati.
L'**overloading** è un polimorfismo che permette, in una classe derivata, di aggiungere parametri al metodo ereditato dalla classe base e di poter modificare le istruzioni.
In fase di chiamata del metodo, il compilatore o l'interprete del linguaggio identifica quale versione del metodo deve essere eseguita in base al numero dei parametri passati.

Overloading degli operatori e Python

Python non supporta l'overloading delle funzioni, mentre supporta solo quello degli operatori.
Infatti in Python se ci sono più definizioni di funzioni con lo stesso nome ma con parametri diversi, la dichiarazione dell'ultima funzione sovrascrive le precedenti e il tentativo di usarne una che non abbia il numero di parametri previsto da quella che è stata dichiarata per ultima provoca un errore.

Un esempio di overloading degli operatori è quello della funzione print: in genere viene fatto un overloading con il metodo __str__ che permette di restituire la rappresentazione dell'oggetto in forma di stringa, cioè in forma leggibile.

Librerie di classi

Attraverso l'ereditarietà viene reso possibile il riutilizzo del software creato senza dover partire da zero nella scrittura di un nuovo programma: è sufficiente utilizzare un insieme di classi già definite che hanno dei comportamenti simili a quelli che vogliamo realizzare nel nuovo programma ed estenderle per aggiungere i comportamenti desiderati alla nuova applicazione.

Nel nuovo programma le classi già esistenti non verranno riscritte, portando un vantaggio in termini di tempo e di affidabilità:

- in termini di tempo, poiché non è necessario ricodificare le classi esistenti;
- in termini di affidabilità, poiché le classi già scritte sono state collaudate da altri programmi che le hanno usate in precedenza e quindi sono quasi sicuramente prive di errori.