Programmazione a Oggetti (OOP) in Python

Slide 1: Paradigma Procedurale vs OOP

- **Programmazione Procedurale:** il codice è organizzato in funzioni e strutture dati separate. L'esecuzione avviene sequenzialmente chiamando funzioni che operano sui dati.
- **Programmazione Orientata agli Oggetti:** il codice è organizzato in **oggetti** che combinano dati (*attributi*) e funzioni (*metodi*). Gli oggetti modellano entità reali o concetti, con cui interagiamo chiamando metodi sull'oggetto stesso.
- L'OOP introduce concetti come incapsulamento, ereditarietà e polimorfismo per gestire la complessità e favorire riuso e modularità rispetto allo stile procedurale.

Slide 2: Esempio – Procedurale vs OOP

Lo stesso semplice compito implementato con approccio procedurale e con approccio OOP:

```
# Approccio procedurale: funzioni e dati separati
saldo = 0
def deposita(importo):
    global saldo
    saldo += importo

def preleva(importo):
    global saldo
    saldo -= importo

# Utilizzo dello stile procedurale
deposita(100)
preleva(40)
print(saldo)  # 60

# Approccio OOP: dati e metodi incapsulati in una classe
class BankAccount:
    def __init__(self):
        self.balance = 0
    def deposit(self, amount):
        self.balance += amount
    def withdraw(self, amount):
        self.balance -= amount

# Utilizzo dello stile OOP
account = BankAccount()
account.deposit(100)
account.withdraw(40)
print(account.balance)  # 60
```

Slide 3: Classi e Oggetti

- **Classe:** definisce un nuovo tipo di oggetto, specificando un insieme di attributi (dati) e metodi (funzionalità). È come uno stampo o blueprint da cui creare oggetti.
- **Oggetto (istanza):** entità creata da una classe. Ogni oggetto possiede i dati e i comportamenti definiti dalla classe, ma con stato proprio. Si possono creare più istanze di una stessa classe, ognuna con il proprio stato.

Slide 4: Definire una Classe in Python

La sintassi per definire una classe utilizza la keyword `class` seguita dal nome della classe e dai due punti. Ad esempio:

```
class MyClass:
    def greet(self):
        print("Hello!")
```

Slide 5: Creare un Oggetto (Istanza)

Per creare un oggetto (istanza) si richiama la classe come fosse una funzione, ottenendo un'istanza su cui chiamare i metodi:

```
obj = MyClass()    # crea un'istanza di MyClass
obj.greet()        # stampa: Hello!
```

Slide 6: Attributi di Istanza

- Gli **attributi di istanza** sono variabili che appartengono al singolo oggetto. Vengono tipicamente inizializzati all'interno di `__init__` con `self`, ad esempio `self.nome = "Mario"`.
- Ogni istanza ha i propri valori per questi attributi (stato interno). Modificare un attributo di istanza influenza solo quell'oggetto, non gli altri.
- Esempio: in una classe `Persona`, attributi come `nome` o `età` saranno attributi di istanza (ogni persona ha il proprio nome ed età).

Slide 7: Attributi di Classe

- Gli **attributi di classe** sono variabili definite **direttamente nella classe**, fuori da qualsiasi metodo. Sono condivisi da *tutte* le istanze della classe.
- Un attributo di classe esiste una sola volta e il suo valore è comune a tutti gli oggetti di quella classe. Ad esempio, un contatore di istanze creato.
- Se si modifica il valore di un attributo di classe, la modifica sarà visibile da tutte le istanze (a meno che un'istanza abbia un attributo di istanza con lo stesso nome, che "oscurerebbe" quello di classe).

Slide 8: Esempio – Attributi di Istanza vs Classe

```
class MyClass:
    shared = 0              # attributo di classe
    def __init__(self, val):
        self.instance = val  # attributo di istanza

a = MyClass(5)
b = MyClass(10)
print(a.instance, b.instance)  # 5 10 (ogni oggetto ha il suo valore)
print(a.shared, b.shared)      # 0 0 (valore condiviso, definito dalla
classe)

MyClass.shared = 99            # modifica dell'attributo di classe
print(a.shared, b.shared)      # 99 99 (tutte le istanze vedono il nuovo
valore)
```

Slide 9: Metodi di Istanza

- I **metodi di istanza** sono le funzioni definite all'interno di una classe che operano su un'istanza specifica. Il primo parametro è sempre `self`, che rappresenta l'istanza su cui il metodo è invocato.
- Chiamando un metodo sull'oggetto (es. `obj.metodo()`), Python passa automaticamente l'istanza come `self` al metodo.
- I metodi di istanza possono leggere o modificare lo stato dell'oggetto (ovvero gli attributi di istanza) e chiamare altri metodi all'interno della classe tramite `self`.

Slide 10: Metodi di Classe

- I **metodi di classe** sono definiti con il decoratore `@classmethod`. Ricevono come primo argomento `cls` invece di `self`, che fa riferimento alla classe stessa.
- Un metodo di classe può quindi operare sui dati a livello di classe, ad esempio per implementare costruttori alternativi o logica che riguarda tutte le istanze.
- Si invoca un metodo di classe tramite la classe (es. `MiaClasse.metodo_di_classe()`), ma può essere chiamato anche da un'istanza (passando comunque la classe come `cls`).

Slide 11: Metodi Statici

- I **metodi statici** sono definiti con `@staticmethod`. Non ricevono automaticamente né `self` né `cls` come parametro.
- Sono essenzialmente funzioni normali messe all'interno della definizione di una classe per organizzazione. Non accedono allo stato dell'istanza né della classe.
- Si chiamano attraverso la classe (es. `MiaClasse.metodo_statico()`) o un'istanza. Sono utili per funzioni di utilità correlate alla classe, ma che non richiedono l'accesso a `self` o `cls`.

Slide 12: Esempio – Metodi di Istanza, Classe, Statici

```
class Demo:
    def instance_method(self):
        print(f"Istanza: {self}")
    @classmethod
    def class_method(cls):
        print(f"Classe: {cls}")
    @staticmethod
    def static_method():
        print("Metodo statico")

obj = Demo()
obj.instance_method()    # Istanza: <__main__.Demo object at 0x...>
Demo.class_method()      # Classe: <class '__main__.Demo'>
Demo.static_method()     # Metodo statico
```

Slide 13: Il Costruttore `__init__`

`__init__` è il metodo speciale (detto costruttore) che Python chiama subito dopo la creazione di un nuovo oggetto di una classe. Serve per inizializzare lo stato dell'oggetto: - Viene definito all'interno della classe con il nome esatto `__init__` e parametro `self` (oltre ad eventuali altri parametri necessari per l'inizializzazione). - All'interno di `__init__` si assegnano di solito gli attributi di istanza iniziali (es. `self.attributo = valore`). - Esempio tipico: in una classe `Persona`, `__init__` può accettare parametri come nome e età e assegnarli all'oggetto appena creato.

Slide 14: Esempio – Costruttore `__init__`

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

p = Punto(2, 3)
print(p.x, p.y)  # output: 2 3
```

Slide 15: Metodi `__str__` e `__repr__`

- `__str__` e `__repr__` sono metodi speciali per fornire una rappresentazione testuale dell'oggetto.
- `__str__` dovrebbe restituire una stringa "leggibile" e di uso generico, destinata ad utenti finali o log output. È quello che viene usato da `print(obj)`.
- `__repr__` dovrebbe restituire una stringa che rappresenta l'oggetto in modo non ambiguo, tipicamente utile per debug e sviluppatori (idealmente, un codice Python che ricostruisce l'oggetto). È usato quando si valuta l'oggetto in una console interattiva o si chiama esplicitamente `repr(obj)`.

- Buona pratica: definire sempre `__repr__` e, se serve una rappresentazione diversa più user-friendly, definire anche `__str__`. In assenza di `__str__`, Python usa `__repr__` come fallback.

Slide 16: Esempio – `__str__` vs `__repr__`

```
class Punto:
    def __init__(self, x, y):
        self.x = x; self.y = y
    def __repr__(self):
        return f"Punto({self.x}, {self.y})"
    def __str__(self):
        return f"({self.x}, {self.y})"

p = Punto(2, 3)
print(p)        # output: (2, 3)        -> __str__
print(repr(p))  # output: Punto(2, 3)  -> __repr__
```

Slide 17: Incapsulamento

- **Incapsulamento** significa nascondere i dettagli interni di un oggetto e mostrare solo un'interfaccia pubblica essenziale per il suo utilizzo.
- In una classe, l'incapsulamento si ottiene raggruppando dati (attributi) e metodi che operano su di essi. L'oggetto gestisce i propri dati interni e fornisce metodi per interagire con essi, prevenendo usi non validi dello stato interno.
- Questo approccio migliora la modularità e la manutenibilità: i cambiamenti interni all'implementazione di una classe non impattano il codice esterno finché l'interfaccia (metodi pubblici) rimane invariata.

Slide 18: Attributi Privati in Python

- In Python non esiste un vero meccanismo di accesso privato agli attributi come in altri linguaggi (es. `private` in Java). Per convenzione, un attributo il cui nome inizia con `_` (singolo underscore) è considerato **interno** (uso riservato all'implementazione).
- Se un nome inizia con `__` (doppio underscore), Python applica il *name mangling* (es. `__x` diventa `_NomeClasse__x`), rendendo più difficile l'accesso accidentale dall'esterno. Questo è usato per attributi davvero interni alla classe.
- Tuttavia, nulla impedisce di accedere comunque a tali attributi “pseudo-privati” (sono solo offuscati, non realmente nascosti). Pertanto, l'incapsulamento in Python si basa più sulla disciplina degli sviluppatori e sulle convenzioni.

Slide 19: Getter e Setter

- In molti linguaggi OOP si utilizzano metodi **getter** e **setter** per accedere e modificare attributi privati. In Python, spesso questo non è necessario: se un attributo può essere pubblico, si accede ad esso direttamente (`obj.attr`).
- Se serve logica aggiuntiva durante la lettura o scrittura di un attributo (validazione, calcolo lazy, notifica, ecc.), si può utilizzare un getter/setter. Ma invece di esporli come `get_x()` / `set_x()`, in Python si preferisce usare le *property*.

- Le property permettono di utilizzare la sintassi di accesso agli attributi (`obj.attr`) pur eseguendo in background codice personalizzato per il get/set.

Slide 20: Decoratore `@property`

- Il decoratore `@property` consente di definire attributi *managed* senza rompere l'interfaccia. Si definisce un metodo getter che viene usato come se fosse un attributo.
- Per definire anche la scrittura, si aggiunge un metodo con `@nomedellaProperty.setter`. In questo modo, assegnare a `obj.attr = valore` invoca il metodo setter.
- Le property aiutano a implementare l'incapsulamento *pythonic*: permettono di cambiare l'implementazione interna senza modificare il codice che usa la classe (l'accesso resta tramite attributo).
- Esempio d'uso: validare valori, calcolare un attributo al volo la prima volta che viene richiesto, ecc.

Slide 21: Esempio – `@property` (Getter/Setter)

```
class Persona:
    def __init__(self, nome):
        self._nome = nome  # attributo interno (convenzione underscore)
    @property
    def nome(self):
        return self._nome.capitalize()
    @nome.setter
    def nome(self, nuovo_nome):
        if len(nuovo_nome) >= 3:
            self._nome = nuovo_nome
        else:
            print("Nome troppo corto")

p = Persona("mario")
print(p.nome)   # output: Mario  (chiamata al getter)
p.nome = "lu"   # Nome troppo corto (chiamata al setter con valore non
valido)
p.nome = "luigi"
print(p.nome)   # output: Luigi
```

Slide 22: Ereditarietà – Concetti di Base

- **Ereditarietà**: meccanismo per cui una classe (detta *sottoclasse* o classe derivata) può **ereditare** attributi e metodi da un'altra classe (detta *superclasse* o classe base).
- La sottoclasse riutilizza il codice della superclasse, promuovendo il riuso. Può inoltre aggiungere nuovi attributi/metodi o **sovrascrivere** quelli esistenti per modificarne il comportamento.
- L'ereditarietà modella una relazione "è-un": la sottoclasse rappresenta una specializzazione della superclasse. Ad esempio, se `Veicolo` è una classe base, una classe `Auto` può ereditarne le caratteristiche di base ed estenderle.

Slide 23: Ereditarietà Singola

- In molti linguaggi OOP l'ereditarietà è **singola**, ovvero una sottoclasse estende al più una superclasse. Python supporta anche l'ereditarietà multipla (più avanti), ma il caso comune è la singola.
- La sintassi per definire una sottoclasse in Python è: `class Sottoclasse(Superclasse):` nel nome della classe.
- Ogni classe che non specifica una superclasse eredita implicitamente da `object` (tutte le classi in Python derivano in ultima analisi da `object`).
- Si possono avere gerarchie multilevel: es. C sottoclasse di B, B sottoclasse di A.

Slide 24: Ereditarietà Multipla

- Python consente una classe di ereditare da **più** classi base: `class C(A, B): ...` indica che C estende sia A che B.
- L'ereditarietà multipla può essere potente per combinare funzionalità, ma introduce complessità: se le classi base hanno metodi con lo stesso nome, bisogna capire quale viene ereditato.
- Python risolve questi conflitti usando l'**MRO (Method Resolution Order)**, un ordine lineare che determina la precedenza delle classi base quando si cerca un attributo/metodo. Si può vedere l'ordine con `Classe.__mro__`.
- I mixin (come vedremo) sono un caso d'uso comune e controllato di ereditarietà multipla per aggiungere funzionalità.

Slide 25: Esempio – Ereditarietà Semplice

```
class Veicolo:
    def avvia(self):
        print("Veicolo in moto")
class Auto(Veicolo):
    def apri_portiere(self):
        print("Portiere aperte")

auto = Auto()
auto.avvia()         # output: Veicolo in moto  (metodo ereditato da Veicolo)
auto.apri_portiere() # output: Portiere aperte  (metodo definito in Auto)
```

Slide 26: Esempio – Ereditarietà Multipla

```
class Schedulabile:
    def schedula(self):
        print("Operazione schedulata")
class Loggable:
    def log(self):
        print("Log operazione")
class Servizio(Schedulabile, Loggable):
    def esegui(self):
        print("Servizio eseguito")
```

```
s = Servizio()
s.esegui()    # Servizio eseguito  (definito in Servizio)
s.schedula()  # output: Operazione schedulata  (ereditato da Schedulabile)
s.log()       # output: Log operazione  (ereditato da Loggable)
```

Slide 27: Overriding (Sovrascrittura) di Metodi

- Una sottoclasse può **sovrascrivere** (override) un metodo della superclasse definendone uno con lo stesso nome. L'implementazione nella sottoclasse prende il posto di quella originale quando il metodo è chiamato sull'istanza della sottoclasse.
- L'overriding permette di adattare il comportamento ereditato alle esigenze della sottoclasse. Ad esempio, una classe base `Forma` può avere un metodo `area()` che ritorna 0, e le sottoclassi specifiche (`Rettangolo`, `Cerchio`) sovrascrivono `area()` per calcolarla correttamente.
- Nota: l'override riguarda i metodi d'istanza e di classe; gli attributi di classe possono essere sovrascritti assegnando un nuovo valore nella sottoclasse, se necessario.

Slide 28: Funzione `super()`

- La funzione built-in `super()` fornisce un modo per riferirsi alla superclasse all'interno di una classe derivata. Viene usata tipicamente per chiamare la versione originale di un metodo che è stato sovrascritto nella sottoclasse.
- Caso comune: dentro `__init__` della sottoclasse, chiamare `super().__init__(args)` per eseguire la logica di inizializzazione definita dalla superclasse (impostando parti dello stato ereditato).
- `super().metodo(args...)` può essere usato anche in altri metodi sovrascritti, per riutilizzare la logica della superclasse e poi estenderla.
- In scenari di ereditarietà multipla, `super()` segue l'MRO, permettendo chiamate "cooperative" attraverso le varie superclassi.

Slide 29: Esempio – Override e `super()`

```
class Person:
    def __init__(self, name):
        self.name = name
    def greet(self):
        print(f"Ciao, sono {self.name}")
class Employee(Person):
    def __init__(self, name, company):
        super().__init__(name)      # chiama __init__ di Person
        self.company = company
    def greet(self):
        super().greet()             # chiama greet di Person
        print(f"Lavoro in {self.company}")

e = Employee("Alice", "ACME")
e.greet()
# Output:
```

```
# Ciao, sono Alice
# Lavoro in ACME
```

Slide 30: Duck Typing

- **Duck typing:** "Se sembra un'anatra e starnazza come un'anatra, allora è un'anatra". In altri termini, in Python non importa la **classe** di appartenenza di un oggetto, ma **quali metodi/attributi offre**.
- Questo consente di scrivere funzioni/metodi che funzionano con qualsiasi oggetto purché abbia certi metodi, senza richiedere che appartenga a una particolare gerarchia.
- Esempio: se una funzione richiede che l'oggetto passato abbia un metodo `connetti()`, qualsiasi oggetto con `connetti()` (indipendentemente dalla sua classe) sarà accettato e funzionerà.
- Il duck typing aumenta la flessibilità del codice, ma richiede che lo sviluppatore documenti chiaramente le **interfacce attese** (insieme di metodi necessari) per evitare errori a runtime.

Slide 31: Interfacce Informali in Python

- In Python non esiste una sintassi dedicata per dichiarare un'interfaccia come nei linguaggi statici. Un'interfaccia può essere intesa come un insieme di metodi che un oggetto dovrebbe implementare per essere "conforme".
- Spesso le interfacce in Python sono **informali**: si stabilisce tramite documentazione o convenzione che una classe debba fornire certi metodi. Ad esempio, una classe "simil-lista" dovrebbe implementare `__len__` e `__getitem__` per essere iterabile e indicizzabile.
- Le **classi astratte** (ABC) possono essere usate per creare interfacce formali (definendo metodi abstract), ma molte librerie Python si affidano a duck typing puro.
- In pratica, se due classi non correlate forniscono lo stesso metodo (es. `speak()`), possono essere usate in modo intercambiabile senza una classe base comune – questo è un'interfaccia informale basata sul protocollo atteso.

Slide 32: Polimorfismo

- **Polimorfismo** significa "molte forme": in OOP indica la possibilità di utilizzare in modo uniforme oggetti di classi diverse, purché rispettino un'interfaccia comune.
- Ad esempio, si può avere una funzione `stampa_area(figura)` che chiama `figura.area()`. Grazie al polimorfismo, questa funzione può accettare oggetti di classi diverse (`Cerchio`, `Rettangolo`, `Triangolo`, ecc.) che abbiano tutti un metodo `area()` implementato.
- In Python, il polimorfismo è supportato sia attraverso l'ereditarietà (tutte le sottoclassi di una classe base possono essere trattate come istanze della base) sia attraverso il duck typing (ogni oggetto con i metodi richiesti può essere usato al posto di un altro).
- Vantaggio: il codice polimorfico è più generico e facilmente estendibile, perché aggiungere nuove classi compatibili non richiede modifiche alle funzioni che le usano.

Slide 33: Esempio – Polimorfismo

```
class Cat:
    def speak(self):
        return "Meow"
class Dog:
```

```
    def speak(self):
        return "Woof"

animali = [Cat(), Dog()]
for animale in animali:
    print(animale.speak())
# Output:
# Meow
# Woof
```

In questo esempio, `Cat` *e* `Dog` *non condividono una classe base esplicita, ma entrambe implementano* `speak()`*. Grazie al duck typing, il loop può chiamare* `speak()` *su ogni elemento senza sapere se sia un gatto o un cane.*

Slide 34: Composizione

- **Composizione:** relazione "ha-un". Una classe può *contenere* al suo interno istanze di altre classi, per delegare a esse parte della funzionalità. Invece di ereditare da una classe per riutilizzarne il codice, si può includere un oggetto di quella classe come attributo.
- Esempio: una classe `Auto` può *contenere* un'istanza di classe `Motore` come attributo, anziché ereditare da `Motore`. L'auto può delegare al suo motore la funzionalità di `avvia()`, pur non essendo essa stessa un motore.
- La composizione tende a produrre sistemi più modulari e flessibili rispetto all'uso estensivo dell'ereditarietà, perché le componenti possono essere combinate, sostituite o estese indipendentemente.

Slide 35: Composizione vs Ereditarietà

- **Ereditarietà**: stabilisce una relazione forte tra classi (sottotipo). La sottoclasse è un tipo specifico di superclasse. Vantaggiosa quando c'è un chiaro rapporto gerarchico "è-un". Può però portare a gerarchie rigide e meno riusabili se usata eccessivamente.
- **Composizione**: stabilisce una relazione di utilizzo. Una classe include un'altra ("ha-un") per delegare compiti. Offre maggiore flessibilità: si possono cambiare le parti interne senza impattare l'interfaccia esterna. Favorisce il riuso senza necessità di una relazione di tipo.
- Principio generale: *"Favorire la composizione rispetto all'ereditarietà"* quando appropriato. L'ereditarietà è indicata quando la relazione naturale tra concetti è realmente gerarchica, altrimenti la composizione spesso porta a design più manutenibili.

Slide 36: Esempio – Composizione

```
class Engine:
    def start(self):
        print("Motore avviato")
class Car:
    def __init__(self):
        self.engine = Engine()  # composizione: Car possiede un Engine
    def start(self):
        # delega l'azione di avvio al sotto-oggetto Engine
        self.engine.start()
```

```
        print("Auto avviata")

auto = Car()
auto.start()
# Output:
# Motore avviato
# Auto avviata
```

Slide 37: Classi Astratte

- Una **classe astratta** è una classe che funge da modello generale per altre classi ma che non viene istanziata direttamente. Può contenere sia implementazioni comuni sia dichiarazioni di metodi che le sottoclassi dovranno obbligatoriamente implementare.
- I metodi dichiarati ma non implementati si chiamano **metodi astratti**. Definiscono un'interfaccia che le sottoclassi concrete dovranno fornire. Se una sottoclasse non implementa tutti i metodi astratti ereditati, anch'essa è astratta e non istanziabile.
- Le classi astratte servono per assicurare che un insieme di sottoclassi abbiano certi metodi/attributi, e per riunire implementazioni comuni. In Python si possono definire usando il modulo `abc`.

Slide 38: Il modulo `abc` e `@abstractmethod`

- Per dichiarare formalmente classi astratte in Python si usa il modulo `abc`. Una classe astratta eredita da `abc.ABC`.
- I metodi che devono essere implementati dalle sottoclassi si decorano con `@abstractmethod`. La presenza di almeno un metodo astratto rende la classe non istanziabile.
- Esempio: si può definire una classe base astratta `Forma` con metodo astratto `area()`. Le sottoclassi (`Cerchio`, `Rettangolo`, ...) dovranno implementare `area()`. Finché non lo fanno, non potranno essere istanziate.
- Le classi astratte possono anche fornire implementazioni default di alcuni metodi (non astratti) che le sottoclassi ereditano.

Slide 39: Esempio – Classe Astratta

```
from abc import ABC, abstractmethod

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

class Cerchio(Forma):
    def __init__(self, raggio):
        self.raggio = raggio
    def area(self):
        return 3.14 * (self.raggio ** 2)

c = Cerchio(5)
```

```
print(c.area())   # output: 78.5
f = Forma()       # ERRORE: classe astratta non istanziabile
```

(Tentare di istanziare `Forma` *solleva un TypeError, perché non ha implementato il metodo astratto* `area()`*.)*

Slide 40: Mixin – Concetto

- Un **mixin** è una classe pensata per fornire funzionalità aggiuntive ad altre classi tramite ereditarietà multipla, senza rappresentare da sola un concetto autonomo.
- I mixin tipicamente forniscono un insieme ristretto di metodi utili (e a volte attributi) che possono essere "mischiati" in più classi. Ad esempio, si può avere un `LogMixin` che fornisce un metodo `log()` da ereditare in varie classi differenti.
- Di solito un mixin non viene istanziato da solo; è progettato per essere usato insieme ad una classe principale. Serve a evitare duplicazione di codice implementando una funzionalità una volta sola e riutilizzandola in diverse classi.

Slide 41: Mixin in Python – Utilizzo

- Per usare un mixin, una classe lo include come una delle sue superclassi. Esempio: `class MiaClasse(LogMixin, BaseClass): ...`. In questo modo `MiaClasse` eredita i metodi di `LogMixin`.
- È buona pratica far terminare il nome di una classe mixin con *Mixin* per riconoscerla. Inoltre, la documentazione dovrebbe chiarire come va usata (es: "questa classe va ereditata insieme a una classe principale che...").
- L'ordine delle superclassi è importante: secondo l'MRO di Python, i metodi delle classi più a sinistra nella tupla di ereditarietà hanno precedenza in caso di nomi in conflitto. Mixin che forniscono override di metodi dovrebbero essere posti a sinistra della classe principale.
- I mixin favoriscono un riuso orizzontale del codice (tra classi sibling) complementare all'ereditarietà verticale classica.

Slide 42: Esempio – Mixin

```
class PrintableMixin:
    def __str__(self):
        return str(self.__dict__)
class Person:
    def __init__(self, name):
        self.name = name
class Employee(PrintableMixin, Person):
    def __init__(self, name, company):
        super().__init__(name)
        self.company = company

e = Employee("Mario", "XYZ")
print(e)  # output: {'name': 'Mario', 'company': 'XYZ'}
```

Qui `Employee` *eredita sia da* `Person` *che da* `PrintableMixin`*. Grazie al mixin, l'istanza di* `Employee` *supporta* `str()`*, che restituisce un dizionario con gli attributi dell'oggetto.*

Slide 43: Metaclassi – Concetto

- In Python, le classi stesse sono oggetti e vengono create da **metaclassi**. Una *metaclasse* è la "classe di una classe": definisce come vengono costruite le classi.
- La metaclasse predefinita è `type`. Quando scriviamo `class MyClass: ...`, Python in realtà crea l'oggetto classe `MyClass` chiamando `type("MyClass", basi, dizionario_attributi)`.
- Definendo una propria metaclasse (sottoclasse di `type`), possiamo personalizzare la creazione di classi. Questo permette, ad esempio, di registrare automaticamente classi, aggiungere attributi/metodi extra, applicare restrizioni, ecc. al momento della definizione.
- Le metaclassi sono un argomento avanzato e vanno usate con cautela, ma sono potenti per implementare funzionalità di "metaprogrammazione" (codice che manipola definizioni di classi).

Slide 44: Metaclassi in Python – Utilizzo

- Per specificare una metaclasse per la propria classe, si usa la sintassi: `class NomeClasse(metaclass=NomeMetaclasse):`.
- All'interno di una metaclasse, si possono sovrascrivere metodi speciali:
- `__new__(mcls, name, bases, attrs)`: crea e ritorna il nuovo oggetto classe.
- `__init__(mcls, name, bases, attrs)`: inizializza l'oggetto classe appena creato.
- (Altri metodi meno comuni possono essere sovrascritti per comportamento avanzato.)
- Un esempio d'uso: definire una metaclasse che registra tutte le sottoclassi create (utile per tenere traccia di tipi disponibili). Oppure una metaclasse che modifica automaticamente alcuni attributi (come aggiungere metodi di logging).
- Molti framework Python utilizzano metaclassi "sotto il cofano" per semplificare l'uso delle classi da parte degli sviluppatori (ad es. Django ORM per definire modelli).

Slide 45: Esempio – Metaclasse

```
class MyMeta(type):
    def __new__(cls, name, bases, attrs):
        print(f"Creazione della classe {name}")
        return super().__new__(cls, name, bases, attrs)

class Test(metaclass=MyMeta):
    pass

# Definizione della classe Test innesca la chiamata a MyMeta.__new__:
# Output: Creazione della classe Test

obj = Test()  # Creazione di un'istanza (nessun output aggiuntivo dalla
metaclasse)
```

Slide 46: Metodi Magici (Magic Methods)

- I **metodi magici** (o "metodi dunder", double underscore) sono metodi speciali che iniziano e finiscono con `__`. Python li chiama automaticamente in certe situazioni, permettendoci di definire il comportamento di oggetti con operatori e funzioni built-in.
- Esempi: `__add__` per l'operatore `+`, `__len__` per la funzione `len()`, `__iter__` per rendere un oggetto iterabile, `__call__` per renderlo chiamabile, `__enter__/__exit__` per il contesto `with`, ecc.
- Implementando questi metodi, possiamo far sì che gli oggetti di una classe si comportino in modo personalizzato nelle espressioni o costrutti corrispondenti. Ad esempio, definendo `__eq__`, due oggetti della classe possono essere confrontati con `==`.
- I metodi magici vengono invocati implicitamente da Python; non si chiamano direttamente. Ad esempio `obj1 == obj2` chiama `obj1.__eq__(obj2)`.

Slide 47: Metodi di Confronto: `__eq__` e `__lt__`

- `__eq__(self, other)`: definisce l'uguaglianza (`==`) tra oggetti. Dovrebbe restituire True/False. Spesso conviene verificare che `other` sia dello stesso tipo di `self` prima di confrontare attributi.
- `__lt__(self, other)`: definisce il confronto "minore di" (`<`). Similmente, dovrebbe restituire True/False confrontando i campi appropriati.
- Implementando questi (e altri come `__le__`, `__gt__`, `__ne__`), è possibile usare gli oggetti nelle comparazioni ordinarie e nelle funzioni di ordinamento (`sorted`, `min`, `max`).
- Esempio tipico: oggetti che rappresentano entità con un ordine naturale (numeri complessi rispetto al loro modulo, punti 2D rispetto alla distanza dall'origine, ecc.) possono implementare `__lt__` per essere ordinati.

Slide 48: Esempio – `__eq__` e `__lt__`

```
class Punto:
    def __init__(self, x, y):
        self.x = x; self.y = y
    def __eq__(self, other):
        return isinstance(other, Punto) and self.x == other.x and self.y ==
other.y
    def __lt__(self, other):
        return (self.x**2 + self.y**2) < (other.x**2 + other.y**2)

p1 = Punto(1, 1)
p2 = Punto(1, 1)
p3 = Punto(2, 2)
print(p1 == p2)  # True  (usa Punto.__eq__)
print(p1 < p3)   # True  (usa Punto.__lt__)
```

Slide 49: Metodo `__len__`

Implementando `__len__` in una classe, le istanze diventano compatibili con la funzione built-in `len()`. Il metodo dovrebbe restituire un intero che rappresenta la "lunghezza" o dimensione dell'oggetto.

```
class MiaLista:
    def __init__(self, dati):
        self.dati = dati
    def __len__(self):
        return len(self.dati)

ml = MiaLista([1, 2, 3])
print(len(ml))  # output: 3
```

Slide 50: Metodo `__call__`

Se una classe implementa `__call__`, le sue istanze diventano **chiamabili come funzioni**.

```
class Moltiplicatore:
    def __init__(self, fattore):
        self.fattore = fattore
    def __call__(self, x):
        return x * self.fattore

raddoppia = Moltiplicatore(2)
print(raddoppia(5))  # output: 10  (equivalente a raddoppia.__call__(5))
```

In questo esempio, `raddoppia` *è un oggetto chiamabile: usando la sintassi* `raddoppia(5)` *Python invoca internamente* `raddoppia.__call__(5)`*. Questo pattern è utile per oggetti funzione o configurabili.*

Slide 51: Altri Metodi Magici Comuni

- **Operatori aritmetici:** `__add__` (somma `+`), `__sub__` (sottrazione `-`), `__mul__` (moltiplicazione `*`), ecc., per definire operatori matematici tra oggetti.
- **Accesso collezioni:** `__getitem__` (accesso indicizzato `obj[key]`), `__setitem__` (assegnazione indicizzata `obj[key] = val`), `__iter__` (iterazione in `for`), `__contains__` (operatore `in`), per far comportare l'oggetto come una collezione/sequenza.
- **Costrutti di contesto:** `__enter__` e `__exit__` permettono a un oggetto di essere usato con `with` (es: gestione automatica delle risorse, come file).
- **Altri:** `__new__` (controlla la creazione di nuove istanze, usato raramente), `__del__` (chiamato alla distruzione di un oggetto, anch'esso raro in Python moderno). In generale, Python offre un metodo magico per quasi ogni operazione o built-in, permettendo di adattare le proprie classi a vari protocolli.

Slide 52: Design Pattern in Python

- I **Design Pattern** sono schemi progettuali, soluzioni generiche a problemi comuni nello sviluppo software orientato agli oggetti. Esempi noti: Factory, Singleton, Strategy, Observer, Decorator, ecc.
- In Python molti pattern classici si possono implementare in modo più semplice grazie alle caratteristiche del linguaggio (funzioni di prima classe, tipizzazione dinamica). Tuttavia, conoscerli aiuta a strutturare il codice in maniera pulita e riconoscibile.
- Di seguito vedremo alcuni pattern applicati in Python con esempi pratici.

Slide 53: Factory Pattern (Fabbrica)

- **Factory** è un pattern creazionale che fornisce un metodo per creare oggetti, nascondendo la logica di istanziazione al chiamante. Invece di usare direttamente `Classe()` nel codice, si chiama una *factory* che decide quale classe concreta istanziare.
- Utile quando il tipo esatto dell'oggetto da creare dipende da condizioni runtime, o per evitare di esporre dettagli di creazione. Il chiamante riceve un oggetto tramite la factory senza conoscere la classe specifica.
- In Python, una factory può essere una semplice funzione o un metodo di classe/statico. Spesso si usa per istanziare una delle diverse sottoclassi in base a un parametro (ad esempio, creare una forma geometrica specifica in base a un identificatore stringa).

Slide 54: Esempio – Factory Pattern

```
class Shape:
    def draw(self):
        raise NotImplementedError
class Circle(Shape):
    def draw(self):
        print("Disegna Cerchio")
class Square(Shape):
    def draw(self):
        print("Disegna Quadrato")

class ShapeFactory:
    @staticmethod
    def create_shape(tipo):
        if tipo == "cerchio":
            return Circle()
        elif tipo == "quadrato":
            return Square()
        else:
            return None

# Uso della factory:
s = ShapeFactory.create_shape("cerchio")
s.draw()  # output: Disegna Cerchio
```

Slide 55: Singleton Pattern (Unico)

- **Singleton** è un pattern creazionale che garantisce che di una certa classe esista **una sola istanza** in tutto il programma, fornendo un punto di accesso globale a quell'istanza.
- È utile quando un singolo oggetto condiviso deve coordinare le operazioni (es: un logger centrale, gestione della configurazione, connessione unica a database).
- In Python, il Singleton può essere implementato sovrascrivendo `__new__` in modo da riusare sempre la stessa istanza. Tuttavia, spesso si ricorre a soluzioni più semplici: ad esempio usare direttamente variabili di modulo (i moduli Python sono singletons naturali una volta importati). Qui mostriamo l'approccio con `__new__`.

Slide 56: Esempio – Singleton Pattern

```
class Singleton:
    _istanza = None
    def __new__(cls, *args, **kwargs):
        if cls._istanza is None:
            cls._istanza = super().__new__(cls)
        return cls._istanza

a = Singleton()
b = Singleton()
print(a is b)  # output: True
```

Nel codice sopra, la prima volta che `Singleton()` *viene invocato,* `_istanza` *è None e viene creata una nuova istanza. Le chiamate successive restituiscono sempre la stessa* `_istanza`*, assicurando che* `a` *e* `b` *siano lo stesso oggetto.*

Slide 57: Strategy Pattern (Strategia)

- **Strategy** è un pattern comportamentale che incapsula diverse **strategie** (algoritmi intercambiabili) e le rende sostituibili l'una con l'altra in modo trasparente al cliente che le usa. Permette di scegliere algoritmi in modo flessibile a runtime.
- Struttura: si definisce un'interfaccia comune per le strategie (ad es. un metodo `esegui()` o simili). Le implementazioni concrete delle strategie incapsulano diversi algoritmi. Un oggetto *contesto* mantiene un riferimento a una strategia corrente e la usa per svolgere un compito.
- In Python, grazie alle funzioni come oggetti di prima classe, spesso il pattern Strategy può essere implementato semplicemente passando funzioni diverse a un oggetto, invece di definire molte classi separate.

Slide 58: Esempio – Strategy Pattern

```
def strategia_maiuscole(testo):
    return testo.upper()
def strategia_minuscole(testo):
    return testo.lower()

class FormattatoreTesto:
```

```
    def __init__(self, strategia):
        self.strategia = strategia
    def formatta(self, testo):
        return self.strategia(testo)

f1 = FormattatoreTesto(strategia_maiuscole)
f2 = FormattatoreTesto(strategia_minuscole)
print(f1.formatta("Py"))  # output: PY
print(f2.formatta("Py"))  # output: py
```

Nell'esempio, `FormattatoreTesto` *applica una strategia di formattazione sul testo. Passando una funzione differente al costruttore, l'oggetto può comportarsi in modo diverso (ma con la stessa interfaccia* `formatta`*).*

Slide 59: Altri Design Pattern (cenni)

- **Decorator (Decoratore):** permette di aggiungere dinamicamente funzionalità a un oggetto, avvolgendolo in un altro oggetto dello stesso tipo che estende il comportamento (da non confondere con i decoratori di funzione in Python, concetto diverso).
- **Observer (Osservatore):** definisce una dipendenza uno-a-molti tra oggetti, in modo che quando uno stato cambia, tutti i suoi osservatori vengano notificati e aggiornati automaticamente (es. implementazione di un sistema di eventi).
- **Adapter (Adattatore):** consente a classi con interfacce incompatibili di collaborare, traducendo l'interfaccia di una classe in un'altra attesa dal client.
- **Template Method:** definisce lo scheletro di un algoritmo in un metodo della classe base, delegando alle sottoclassi la realizzazione di alcuni passi senza modificare la struttura generale.
- **Altri pattern:** Command, Proxy, Iterator, State, Visitor, ecc.

Slide 60: Conclusioni

- La programmazione a oggetti in Python offre un ricco insieme di funzionalità: dalla definizione semplice di classi e oggetti, fino a meccanismi avanzati come metaclassi e decoratori di classe.
- Comprendere i concetti base (incapsulamento, ereditarietà, polimorfismo) e come si applicano in Python è fondamentale per progettare codice ben organizzato e manutenibile.
- I pattern di design classici sono applicabili in Python, ma il linguaggio spesso offre idiomi più diretti. È utile conoscere entrambi: sfruttare la flessibilità di Python senza perdere di vista la struttura offerta dai principi OOP.
- In sintesi, l'OOP in Python permette di modellare soluzioni complesse in modo naturale e modulare, facilitando la crescita e la scalabilità dei progetti software.
