Lezione 2: Modulo 1 – Revisione e Best Practices Python (parte 2) + Inizio Modulo 2 – OOP

Uso efficace della libreria standard (Slide 1)

Sfruttare moduli e funzioni integrate, "don't reinvent the wheel"

Una caratteristica che distingue un programmatore Python esperto è la capacità di **riutilizzare ciò che il linguaggio già offre** invece di reinventare soluzioni da zero. Python ha una libreria standard ricchissima e comunità di pacchetti esterni molto attiva. Alcuni consigli:

- **Funzioni built-in e moduli standard**: Prima di scrivere una funzione da zero, chiedersi se esiste già. Ad esempio, per operazioni matematiche c'è il modulo `math` (con funzioni come `math.sqrt`, costanti come `math.pi`), per manipolare date c'è `datetime`, per operare su file CSV c'è il modulo `csv`, per interagire con il sistema operativo c'è `os` e `pathlib`, e così via. Usare queste librerie rende il codice più affidabile (sono ben testate) e spesso più efficiente. Esempio:

```
import math
numeri = [5.5, 2.3, 7.1]
# Calcolare la parte frazionaria di ciascun numero usando math.modf
frazioni = [math.modf(x)[0] for x in numeri]
print(frazioni)  # Output: [0.5, 0.2999999999999998,
0.09999999999999964]
```

  In questo esempio utilizziamo `math.modf` per ottenere parte intera e frazionaria dei numeri. Potevamo calcolarla con `x - int(x)`, ma fidarci di una funzione standard riduce errori (gestisce anche valori negativi correttamente) e comunica chiaramente l'intento.
- **Non duplicare funzionalità**: Se stiamo scrivendo codice per, ad esempio, ordinare una lista in base a un criterio, ricordiamo che esiste già `sorted()` o il metodo `.sort()` delle liste che accettano anche una chiave di ordinamento. Idem per ricerche all'interno di iterabili, esiste la costruzione `in` (es. `if "abc" in lista_stringhe:`) anziché scrivere un loop manuale. Questo approccio porta a codice più conciso e spesso più veloce (le implementazioni in C sono molto ottimizzate).

- **Esempio pratico – gestione configurazione**: Immaginate di dover salvare e caricare impostazioni di un'applicazione. Invece di creare un formato proprietario e scrivere parser a mano, è conveniente usare formati standard come JSON o YAML. Python ha `json` integrato e librerie esterne per YAML. Ad esempio:

```
import json
# Salvataggio di un dizionario di configurazione su file JSON
config = {"utente": "admin", "tema": "scuro", "lingua": "it"}
with open("config.json", "w") as f:
    json.dump(config, f, indent=4)
```

In poche righe salviamo un dict in formato JSON. Allo stesso modo, per caricarlo useremo `json.load`. Questo evita errori e segue standard che altre applicazioni possono leggere. **Best practice**: preferire standard aperti e librerie collaudate per interoperabilità e robustezza.

- **Utilizzare le librerie esterne quando appropriato**: per compiti più complessi, spesso esistono librerie create dalla community (ad esempio, `requests` per le chiamate HTTP, `pandas` per analisi dati, etc.). In un contesto professionale, non esitare a dipendere da librerie affidabili invece di implementare tutto in casa – a patto di valutare le dipendenze (maturità, licenza, supporto). Questo corso è focalizzato sul linguaggio, ma è bene tenere a mente di non "reinventare la ruota" quando non serve.

Comprensioni di liste e generatori (Slide 2)

Liste, dizionari e generatori per codice più pulito ed efficiente

Abbiamo già introdotto le **list comprehension** nella lezione precedente. Approfondiamo e aggiungiamo due concetti correlati: le *dictionary comprehension* e i *generatori*.

- **List & dict comprehension**: Oltre alle liste, Python permette di creare facilmente anche dizionari e set con sintassi simile. Esempio di **dictionary comprehension** – supponiamo di avere una lista di prodotti e vogliamo un dizionario che mappi il nome prodotto al suo prezzo scontato:

```
prodotti = {"telefono": 1000, "tablet": 700, "portatile": 1200}
# Creiamo un nuovo dict con prezzi scontati del 10%
prezzi_scontati = {nome: prezzo * 0.9 for nome, prezzo in
prodotti.items()}
print(prezzi_scontati)
# Output: {'telefono': 900.0, 'tablet': 630.0, 'portatile': 1080.0}
```

Questa comprensione itera sulle coppie chiave-valore di `prodotti.items()` e calcola il nuovo dict. In generale, le comprensioni permettono di esprimere in modo conciso trasformazioni di collezioni. Sono particolarmente utili in **analisi dati** per preparare dataset filtrati o trasformati.

- **Generatori**: Un *generatore* è un oggetto iterabile che produce i valori *su richiesta* invece che crearli tutti insieme in memoria. Si definisce come una comprensione tra parentesi tonde invece che quadre, oppure usando la parola chiave `yield` in una funzione (generator function). Esempio:

```
# Lista di quadrati (comprensione normale, crea subito la lista
completa)
quadrati = [x*x for x in range(1000000)]
# Generatore di quadrati (non crea subito la lista, genera un valore
alla volta)
quadrati_gen = (x*x for x in range(1000000))
```

Nel primo caso `quadrati` occuperà memoria proporzionale a un milione di elementi. Nel secondo caso `quadrati_gen` occupa pochissima memoria e calcola ogni valore man mano che viene iterato.
Quando usare i generatori: se lavoriamo con insiemi di dati molto grandi (es. file di log da diversi MB, streaming di dati) e non abbiamo bisogno di accedere ai valori più di una volta, i

generatori permettono di risparmiare memoria e iniziare l'elaborazione prima (non si aspetta che l'intera collezione sia pronta). Un esempio pratico: leggere un file riga per riga senza caricarlo interamente:

```
def leggi_righe(file_path):
    with open(file_path) as f:
        for riga in f:
            yield riga.strip()  # restituisce una riga alla volta

# Uso del generatore
for linea in leggi_righe("grande_file.txt"):
    processa(linea)
```

Qui la funzione genera le righe una alla volta; possiamo processarle in streaming.

- **Trade-off**: Le comprehension (liste, dict) sono ottime se il dataset non è gigantesco e ci serve avere accesso casuale o iterare più volte. I generatori sono preferibili per streaming di dati o quando vogliamo calcolare valori pigro (*lazy evaluation*). Spesso si inizia con una comprehension per semplicità e, se si incontrano limiti di memoria, si passa a un generatore.

Gestione delle eccezioni (Slide 3)

EAFP vs LBYL e gestione robusta degli errori

Il robusto trattamento degli errori è parte integrante del codice professionale. Python offre il meccanismo di **eccezioni** per intercettare e gestire situazioni anomale. Alcune linee guida e best practice:

- **EAFP (Easier to Ask Forgiveness than Permission)**: Questo mantra Pythonista suggerisce di tentare un'operazione direttamente, *catturando* l'errore se avviene, piuttosto che controllare preventivamente le condizioni per evitare l'errore. Ad esempio, invece di:

```
# LBYL (Look Before You Leap): controllare prima
if os.path.exists("config.yaml"):
    config_data = open("config.yaml").read()
else:
    config_data = {}
```

È spesso preferibile:

```
# EAFP: provare e gestire eventuali errori
try:
    with open("config.yaml") as f:
        config_data = f.read()
except FileNotFoundError:
    config_data = {}
```

Nel secondo approccio, non c'è una finestra di tempo tra il controllo e l'uso (nell'esempio LBYL, il file poteva essere cancellato dopo il check e prima dell'apertura causando comunque un errore). Il codice EAFP è generalmente più compatto e sicuro da race condition.

- **Catch mirati, non generici**: Quando si intercettano eccezioni, è best practice **specificare il tipo di eccezione** da gestire, invece di usare un generico `except Exception:` o peggio un nudo `except:` che cattura tutto (inclusi segnali di interrupt, es. Ctrl+C). Identificare la specifica eccezione (es. `FileNotFoundError`, `KeyError`, `ValueError`, ecc.) aiuta a non mascherare bug inaspettati. Se necessario, si possono elencare più eccezioni in un'unica linea o usare più blocchi except.
- **Pulizia in caso di errore**: Se si acquisiscono risorse che non sono gestite da context manager, assicurarsi di rilasciarle anche in caso di errore. Ad esempio, chiudere connessioni o file in un `finally:` oppure usare context manager appositi. Con `with`, come visto, questo è automatizzato.
- **Loggare le eccezioni**: in applicazioni reali, spesso **logghiamo** l'errore invece di stamparlo, magari con `logging.error("Messaggio", exc_info=True)` per avere anche lo stacktrace. Questo permette di analizzare i problemi a posteriori senza interrompere bruscamente il programma (salvo situazioni critiche).

Esempio: gestione robusta dell'input

```
import logging

def leggi_int(prompt):
    while True:
        try:
            val = int(input(prompt))
            return val
        except ValueError as e:
            logging.warning(f"Input non valido. Inserisci un numero intero.
Errore: {e}")
            # loop ricomincia chiedendo di nuovo
```

In questo esempio, un eventuale valore non convertibile a int genera un `ValueError` che viene catturato; il codice logga un warning e chiede di nuovo l'input all'utente invece di crashare. Notare l'uso di `logging.warning` al posto di `print`: nelle applicazioni reali è preferibile usare il modulo `logging` per questi messaggi (vedi slide successiva).

- **Rilanciare o tradurre eccezioni**: a volte ha senso intercettare un'eccezione e rilanciarne un'altra più significativa per il contesto. Ad esempio, se una funzione di alto livello che elabora un file CSV riceve un `KeyError` da una funzione interna, si potrebbe intercettarlo e rilanciare una nostra eccezione custom tipo `DatiMancantiErrore` con un messaggio più chiaro. La definizione di **eccezioni personalizzate** (ereditando da Exception) può aiutare a gestire errori in modo più semantico, ma assicurarsi di non esagerare (usare i tipi built-in va bene nella maggior parte dei casi).

Logging e debug (Slide 4)

Logging al posto di print, uso di debugger

Scrivere **log** significativi è una best practice fondamentale in contesti professionali. Durante lo sviluppo magari si usano i `print` per capire cosa succede, ma in produzione è meglio usare il modulo

`logging`, che permette di regolare il livello di dettaglio e dirigere l'output (console, file, etc.) in modo flessibile.

- **Uso del modulo** `logging`:

```
import logging
logging.basicConfig(level=logging.INFO)  # Configurazione base del
logging
```

Una volta configurato, si possono emettere messaggi:

```
logging.debug("Messaggio di debug, dettagliato")
logging.info("Informazione generale sul flusso del programma")
logging.warning("Qualcosa di inaspettato ma non critico")
logging.error("Si è verificato un errore")
logging.critical("Errore grave! Applicazione compromessa")
```

I vantaggi di `logging` rispetto a `print` sono molti: possiamo disabilitare o filtrare messaggi sotto un certo livello senza toccare il codice (ad es. mostrare solo warning ed errori in produzione), aggiungere timestamp automatici, loggare su file per analisi successive, ecc. Ad esempio, in un sistema di **automazione** schedulato, i log permettono di capire cosa è successo durante esecuzioni notturne senza dover essere presenti.

- **Non lasciare print di debug nel codice**: è sintomo di codice non pulito vedere stampe che non siano strettamente necessarie. Se servono info per debug, implementarle con logging debug. Così non "sporcano" l'output utente e possono essere attivate solo quando serve.
- **Uso del debugger**: Oltre ai log, un programmatore avanzato sa utilizzare il **debugger** interattivo (ad esempio `pdb` o i debugger integrati negli IDE come PyCharm, VSCode). Questo permette di eseguire il codice passo passo, esaminare lo stato delle variabili in runtime, e trovare bug in maniera più efficiente che con mille stampe. *Best practice*: imparare a mettere breakpoints e a ispezionare il codice interattivamente per velocizzare il troubleshooting.

- **Esempio pratico di logging**:

```
import logging
logging.basicConfig(filename="app.log", level=logging.INFO)

def processa_dati(dati):
    logging.info(f"Inizio elaborazione di {len(dati)} elementi")
    try:
        risultato = algoritmo_complesso(dati)
        logging.info("Elaborazione completata con successo")
        return risultato
    except Exception as e:
        logging.error(f"Errore durante l'elaborazione: {e}",
exc_info=True)
        raise  # rilanciamo l'eccezione dopo averla loggata
```

In questo snippet, i messaggi di log verranno scritti nel file `app.log`. Abbiamo registrato quante entità stiamo per processare, confermato quando finito e, in caso di errore, loggato l'eccezione con `exc_info=True` (che include lo stacktrace). Rilanciamo poi l'errore dopo averlo loggato, decisione da prendere in base alla logica dell'app (qui presumiamo che l'errore sia critico e vada gestito altrove o fermare il programma).

- **if `__name__ == "__main__":**: Un breve cenno a questa convenzione: quando scriviamo script che possono anche essere importati come moduli, incapsulare la logica eseguibile all'avvio sotto questo controllo impedisce che venga eseguita se il file è importato altrove. Ad esempio:`

```
def funzione_utilitaria():
    ...

if __name__ == "__main__":
    # Codice di test o esecuzione script
    dati = leggi_input()
    risultato = funzione_utilitaria(dati)
    print("Risultato:", risultato)
```

`In questo modo la parte dentro l'if gira solo quando eseguiamo direttamente il file, ma non quando lo importiamo in un altro modulo (magari per riutilizzare` **funzione_utilitaria`). Questa è considerata buona pratica per rendere i moduli riusabili e mantenere separata la logica di esecuzione dallo** namespace** globale del modulo.

Paradigma OOP in Python (Slide 5)

Concetti base di programmazione a oggetti

Iniziamo ora il **Modulo 2 – OOP (Object-Oriented Programming)**. La programmazione a oggetti è un paradigma che consente di modellare entità come **oggetti** software contenenti dati (*attributi*) e comportamenti (*metodi*). Molti probabilmente hanno già familiarità con OOP in altri linguaggi; qui ci focalizzeremo sulle peculiarità di Python.

Principali concetti OOP da tenere a mente:
- **Classe**: definisce il *tipo* di oggetto, funge da stampo (template) descrivendo quali attributi e metodi quell'oggetto avrà.
- **Oggetto/istanza**: è un singolo elemento (istanza) creato a partire da una classe, con propri dati. Ad esempio, se `Cliente` è una classe, `cliente1` e `cliente2` possono essere due oggetti distinti con i propri attributi (nome, età, etc.).
- **Incapsulamento**: la classe raggruppa dati e funzioni che operano su di essi. In Python l'incapsulamento è più "soft" che in altri linguaggi – non esiste una vera keyword per attributi privati, si lavora per convenzioni (come vedremo). L'idea è che l'oggetto gestisce il suo stato tramite i metodi, fornendo un'interfaccia pubblica e nascondendo i dettagli interni.
- **Ereditarietà**: una classe può **ereditar**e attributi e metodi da un'altra (classe base), permettendo riuso e specializzazione. Ad esempio possiamo avere una classe base `Veicolo` con attributi generici

(velocità, posizione) e classi derivate `Auto` e `Bicicletta` che ereditano da `Veicolo` ma aggiungono comportamenti specifici.
- **Polimorfismo**: oggetti di classi diverse possono essere usati in modo intercambiabile se condividono un'interfaccia comune. Ad esempio, se sia `Auto` che `Bicicletta` hanno un metodo `muovi()`, posso scrivere codice che chiama `veicolo.muovi()` senza preoccuparsi se `veicolo` sia un'auto o una bici – l'oggetto sa come muoversi secondo la propria implementazione. In Python in particolare si parla spesso di **duck typing**: "se qualcosa cammina come un'anatra e starnazza come un'anatra, probabilmente è un'anatra". In altre parole, conta che l'oggetto fornisca i metodi giusti, non la sua stretta appartenenza a una classe.

Python implementa l'OOP in maniera flessibile: tutto è oggetto (anche i tipi base, le funzioni stesse sono oggetti), le classi sono di fatto oggetti di tipo `type`. Non serve definire tutto in anticipo – si possono aggiungere attributi a runtime agli oggetti – ma ciò va usato con cautela e generalmente si preferisce definire le strutture in modo chiaro all'interno delle classi.

Nei prossimi slide vedremo come **definire una classe**, creare oggetti e utilizzare i principi suddetti in pratica.

Definire classi e oggetti in Python (Slide 6)

init, self e attributi di istanza

Per definire una classe in Python si usa la sintassi: `class NomeClasse: ...`. All'interno, definire un metodo speciale `__init__` (inizializzatore) per specificare cosa avviene alla creazione degli oggetti. Vediamo un esempio concreto:

```
class Config:
    """Classe per gestire una configurazione semplice di un'applicazione."""
    def __init__(self, filepath):
        # Attributi di istanza
        self.filepath = filepath    # percorso del file di config
        self.parametri = {}
        # All'inizializzazione carichiamo subito la configurazione
        try:
            with open(filepath, 'r') as f:
                import json
                self.parametri = json.load(f)
        except FileNotFoundError:
            # Se il file non esiste, iniziamo con config vuota
            self.parametri = {}
            print(f"File di config {filepath} non trovato, uso parametri di
default.")

    def get(self, nome, default=None):
        """Ottiene il valore di un parametro di configurazione, o default se
non esiste."""
        return self.parametri.get(nome, default)

    def set(self, nome, valore):
        """Imposta un parametro di configurazione e salva su file."""
```

```
        self.parametri[nome] = valore
        # Salva immediatamente su file ad ogni modifica
        with open(self.filepath, 'w') as f:
            import json
            json.dump(self.parametri, f, indent=4)
```

Nel codice sopra:
- `class Config:` definisce una classe chiamata `Config`. Il docstring spiega lo scopo (sempre buona abitudine documentare la classe).
- Il metodo `__init__(self, filepath)` è il costruttore che Python chiama ogni volta che creiamo un nuovo oggetto `Config`. Il parametro `self` rappresenta l'istanza che si sta creando (come *this* in altri linguaggi). Non bisogna passarla esplicitamente quando si crea l'oggetto, Python la gestisce automaticamente.
- All'interno di `__init__`, creiamo attributi di istanza come `self.filepath` e `self.parametri`. Ogni oggetto `Config` avrà il proprio `filepath` e `parametri`.
- In questo esempio, al momento della creazione l'oggetto prova a caricare subito i parametri da file JSON. Se il file non c'è, inizializza un dict vuoto e stampa un messaggio (in un contesto reale, potremmo loggarlo invece di stamparlo).
- Abbiamo poi due metodi di utilità: `get` per leggere un parametro (ritorna un default se il parametro non esiste) e `set` per aggiornare un parametro e salvare immediatamente su file (in modo da mantenere sincronizzato il file di config).

Uso della classe (creazione di oggetti):

```
# Creazione di un oggetto Config
config_app = Config("impostazioni.json")
# Accesso ai dati tramite i metodi
lingua = config_app.get("lingua", default="it")
print("Lingua configurata:", lingua)
config_app.set("tema", "chiaro")
```

Quando facciamo `config_app = Config("impostazioni.json")`, Python:
1. Alloca un nuovo oggetto di tipo Config.
2. Chiama `Config.__init__(config_app, "impostazioni.json")` automaticamente, passando l'oggetto stesso come `self`.
3. Dopo `__init__`, l'oggetto è pronto all'uso.

Notare che possiamo creare molteplici oggetti `Config`, ciascuno indipendente con il proprio stato (filepath diverso, parametri diversi).

Attributi di istanza vs attributi di classe: Nel nostro esempio, `filepath` e `parametri` sono attributi legati all'istanza (`self.parametri` ecc.). Python però permette anche di definire attributi a livello di classe, cioè condivisi tra tutte le istanze. Ad esempio:

```
class MioEsempio:
    contatore = 0  # variabile di classe
    def __init__(self):
```

```
        MioEsempio.contatore += 1  # incrementa il contatore ogni volta che
si crea un oggetto
```

Qui `contatore` è definito fuori dai metodi, quindi appartiene alla classe. Lo si può accedere come `MioEsempio.contatore`. Nel `__init__` usiamo `MioEsempio.contatore` (alternativamente `self.__class__.contatore`) per incrementarlo. Questo contatore terrà il numero totale di istanze create, condiviso tra tutti.
Gli **attributi di classe** sono utili per costanti (ad es. un tasso di cambio fisso valido per tutti gli oggetti) o per tenere traccia di informazioni globali relative alla classe. Tuttavia, la maggior parte degli attributi che definiremo saranno di istanza, specifici di ogni oggetto.

Ereditarietà (introduzione) (Slide 7)

Riutilizzare codice con classi derivate

L'ereditarietà permette di creare una nuova classe **riusando** e **specializzando** una classe esistente. In Python si indica mettendo tra parentesi la classe base. Esempio semplice:

```
class Veicolo:
    def __init__(self, velocita=0):
        self.velocita = velocita
    def accelera(self, incremento):
        self.velocita += incremento

class Automobile(Veicolo):
    def __init__(self, velocita=0, marca="Anonima"):
        # chiama il costruttore della classe base Veicolo
        super().__init__(velocita)
        self.marca = marca
    def suona_clacson(self):
        print("Beep beep! Sono un'auto", self.marca)

class Bicicletta(Veicolo):
    def __init__(self, velocita=0, tipo="graziella"):
        super().__init__(velocita)
        self.tipo = tipo
    def impenna(self):
        if self.velocita > 20:
            print("Wow, impennata alla grande!")
        else:
            print("Pedala più forte per impennare!")
```

In questo esempio:
- `Veicolo` è la classe base, con un attributo `velocita` e un metodo `accelera`.
- `Automobile` e `Bicicletta` **ereditano** da `Veicolo`. Significa che automaticamente hanno l'attributo `velocita` e il metodo `accelera` come definiti in `Veicolo`, oltre a ciò che definiscono nel loro corpo.
- Dentro `Automobile.__init__` e `Bicicletta.__init__`, usiamo `super().__init__(velocita)` per chiamare il costruttore di `Veicolo` ed evitare di ripetere

codice di inizializzazione già fatto nella classe base (impostare `self.velocita`). `super()` rappresenta la classe base, e lo chiamiamo con i parametri che `Veicolo.__init__` si aspetta.
- `Automobile` aggiunge un attributo `marca` e un metodo `suona_clacson()`. `Bicicletta` aggiunge `tipo` e un metodo `impenna()`. Entrambe ereditano `accelera` senza bisogno di riscriverlo.

Uso delle classi derivate:

```
a = Automobile(velocita=50, marca="Fiat")
b = Bicicletta(tipo="mountain bike")
a.accelera(10)   # Metodo ereditato, velocita diventa 60
b.accelera(10)   # Bicicletta accelera di 10
a.suona_clacson()  # Output: "Beep beep! Sono un'auto Fiat"
b.impenna()        # Con velocita 10, Output: "Pedala più forte per
impennare!"
```

Notare come `Automobile` e `Bicicletta` possano essere trattate in modo polimorfo come `Veicolo`:

```
fleet = [Automobile(marca="Toyota"), Bicicletta(tipo="BMX"),
Automobile(marca="Tesla")]
for v in fleet:
    v.accelera(5)  # valido per entrambi i tipi, grazie all'ereditarietà
```

Possiamo chiamare `accelera` su `v` senza sapere se sia auto o bici, perché sappiamo che qualunque `Veicolo` (e quindi qualunque sottoclasse) lo possiede.

Override di metodi: le classi derivate possono *sovrascrivere* metodi della classe base per modificarne il comportamento. Ad esempio, se volessimo che `Automobile.accelera` aggiungesse un limite massimo di velocità, potremmo ridefinirlo in `Automobile` (eventualmente chiamando il metodo base via `super()` e poi applicando un limite). Python risolve i metodi cercandoli prima nella classe dell'istanza, poi su su nelle basi (questa è la *Method Resolution Order*, MRO).

Ereditarietà multipla: Python permette a una classe di ereditare da più classi (ad esempio `class C(A, B): ...`). Questo è un meccanismo avanzato che richiede cautela per evitare ambiguità e complessità nel MRO. Si usa in casi specifici, spesso per mescolare funzionalità (mixin). Nel nostro corso accenneremo soltanto alla cosa: la maggior parte dei design OOP può essere soddisfatta con ereditarietà singola e buona composizione tra oggetti.

Polimorfismo e Duck Typing (Slide 8)

Interfacce comuni e flessibilità degli oggetti Python

In linguaggi statici spesso si utilizzano **interfacce** o classi base astratte per definire un insieme di metodi che varie classi concrete devono implementare, garantendo così intercambiabilità. In Python, grazie al duck typing, conta più la presenza dei metodi che la formale relazione di ereditarietà.

Esempio: supponiamo di avere due classi non correlate da ereditarietà, ma concettualmente simili:

```
class Stampante:
    def stampa(self, documento):
        print(f"Stampante: sto stampando {documento}")

class Plotter:
    def stampa(self, documento):
        print(f"Plotter: sto tracciando {documento}")
```

Entrambe hanno un metodo `stampa(documento)`, ma non c'è una classe base comune. Possiamo comunque scrivere una funzione che le utilizzi in modo polimorfo:

```
def invia_alla_stampante(periferica, documento):
    periferica.stampa(documento)

stamp = Stampante()
plot = Plotter()
invia_alla_stampante(stamp, "Relazione.pdf")  # Stampante: sto stampando
Relazione.pdf
invia_alla_stampante(plot, "Progetto.dwg")    # Plotter: sto tracciando
Progetto.dwg
```

Questo funziona perché sia `stamp` che `plot` hanno il metodo `.stampa`. Non ci interessa *di che classe sono*, ma *che cosa sanno fare*. Questo è il duck typing: "se un oggetto sa *stampa()*, allora può essere passato a `invia_alla_stampante`".

ABC (Abstract Base Classes): Python fornisce nel modulo `abc` la possibilità di creare classi base astratte, definendo metodi abstract (decorati con `@abstractmethod`) che le sottoclassi dovranno implementare. Questo serve principalmente come documentazione e per abilitare certe forme di controllo (non puoi istanziare la classe astratta, e le subclass incomplete danno errore se instantiate). Esempio minimale:

```
from abc import ABC, abstractmethod

class PerifericaStampa(ABC):
    @abstractmethod
    def stampa(self, documento):
        pass

class Stampante(PerifericaStampa):
    def stampa(self, documento):
        print(f"Stampante: {documento}")

# plotter che non implementa stampa darebbe errore se provi a istanziarlo
```

In un corso avanzato, potremmo usare ABC per definire delle interfacce formali (ad esempio un'interfaccia per plugin con metodi noti). Tuttavia, molti progetti Python si affidano semplicemente a convenzioni e duck typing senza la necessità di introdurre classi astratte, a meno che il caso d'uso non lo richieda espressamente.

Metodi speciali (dunder methods) (Slide 9)

str, repr, eq e altri per personalizzare il comportamento degli oggetti

I **metodi speciali** (spesso chiamati *dunder* da "double underscore") sono funzioni predefinite con nomi particolari che iniziano e finiscono con `__`. Implementandoli all'interno delle nostre classi, possiamo fare in modo che gli oggetti si comportino in modi specifici in certi contesti. Eccone alcuni importanti:

- `__str__` **e** `__repr__`: definiscono la rappresentazione in stringa dell'oggetto. `__str__` è pensato per una rappresentazione *umana* leggibile, mentre `__repr__` per una rappresentazione *tecnica* (che idealmente potrebbe essere usata per ricostruire l'oggetto). Quando facciamo `print(oggetto)`, Python chiama `__str__`; se non definito, usa `__repr__`. Esempio:

```
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)  -> usa __str__
print([p])          # Output: [Punto(2, 3)] -> in lista mostra __repr__
```

  Definire questi metodi è molto utile per il debug e il logging: avere oggetti che si stampano con informazioni utili aiuta a capire lo stato del programma.
- `__eq__` **(e altri confronti)**: per definire come confrontare due oggetti della nostra classe con `==`. Di default, `a == b` per oggetti personalizzati verifica semplicemente se sono lo stesso oggetto (stessa posizione in memoria). Implementando `__eq__`, possiamo far sì che confronti i valori interni. Ad esempio, per `Punto` potremmo definire:

```
def __eq__(self, other):
    if not isinstance(other, Punto):
        return False
    return self.x == other.x and self.y == other.y
```

  Così due punti con stesse coordinate risulteranno `==` anche se sono oggetti distinti. Se si implementa `__eq__`, è buona norma implementare anche `__ne__` (not equal) o Python dedurrà il contrario di `__eq__` in modo automatico nella maggior parte dei casi. Altri metodi di confronto includono `__lt__`, `__le__`, etc., per `<`, `<=` se ha senso ordinare gli oggetti.
- `__add__`, `__iter__`, `__len__`, **...**: ce ne sono molti. `__add__` permette di definire il comportamento per l'operatore `+` (somma) tra oggetti della classe. `__iter__` permette di rendere l'oggetto iterabile (deve restituire un iteratore, spesso `self` e implementare `__next__` oppure delegare a un attributo iterabile interno). `__len__` definisce cosa restituisce `len(oggetto)`. Ad esempio, se avessimo una classe `Documento` con una lista di pagine, `__len__` potrebbe restituire il numero di pagine.

- **Quando usarli**: i metodi speciali rendono le nostre classi più integrate nel linguaggio, permettono di usare sintassi e funzioni built-in in modo naturale con gli oggetti custom. Non vanno implementati tutti a priori, solo quelli utili. Un caso comune in ambito professionale: implementare `__str__` / `__repr__` per loggare oggetti, `__eq__` per confronti logici (ad esempio confrontare due config), `__iter__` se la classe è una collezione di elementi e vogliamo iterarci sopra.

Esempio pratico: riprendiamo la classe `Config` definita in precedenza. Potremmo aggiungere un metodo speciale `__repr__` per facilitarne il debug:

```
class Config:
    # ... come definito prima ...
    def __repr__(self):
        return f"Config(filepath='{self.filepath}',
num_parametri={len(self.parametri)})"
```

Così, stampando una lista di config o loggando un oggetto Config, vedremo qualcosa come `Config(filepath='impostazioni.json', num_parametri=5)` invece di `<__main__.Config object at 0x...>`, il che è molto più informativo.

Metodi di classe e metodi statici (Slide 10)

Alternative ai costruttori e utilità legate alla classe

Oltre ai normali metodi d'istanza (quelli che abbiamo visto finora, che operano su `self`), Python supporta:

- **Metodi di classe** (`@classmethod`): sono metodi che ricevono come primo argomento la *classe* (`cls`) invece dell'istanza. Si definiscono decorando la funzione con `@classmethod`. Tipicamente si usano per fornire costruttori alternativi o operazioni che riguardano la classe intera, non un singolo oggetto. Ad esempio, potremmo volere un modo rapido per creare un oggetto `Config` con parametri di default senza passare un file:

  ```
  class Config:
      # ... resto della classe ...
      @classmethod
      def config_di_default(cls):

  """Crea un oggetto Config con parametri predefiniti (non salvato su
  file)."""
          obj = cls(filepath="config_default.json")
          obj.parametri = {"lingua": "it", "tema": "scuro"}  # impostiamo
  qualche default
          return obj

  # Utilizzo:
  cfg = Config.config_di_default()
  ```

In questo metodo, usiamo `cls` per referenziare la classe `Config` stessa. Chiamiamo `cls(...)` per costruire un nuovo oggetto. In questo modo, se `Config` venisse sottoclassata, `cls` assicurerebbe di creare un'istanza della sottoclasse, non necessariamente di `Config` base, rendendo il metodo valido anche per classi figlie.
Un altro uso comune di classmethod è ad esempio implementare un metodo `from_file` per creare un oggetto a partire da un file (es: `Config.from_file("path")` come alternativa a chiamare il costruttore normalmente).

- **Metodi statici** (`@staticmethod`): sono semplicemente funzioni definite dentro la classe, ma che non ricevono né `self` né `cls`. In pratica sono utili per definire utility correlate alla classe, che però non necessitano di accedere né all'istanza né alla classe. Si potrebbero definire come funzioni libere nel modulo, ma metterle come staticmethod nella classe aiuta a raggrupparle logicamente con essa.
  Esempio: nella nostra classe `Config`, potremmo definire un metodo statico di utilità per validare i nomi dei parametri:

```
class Config:
    # ... resto ...
    @staticmethod
    def _valida_nome_parametro(nome):
        # convenzione: i nomi validi sono non vuoti e senza spazi
        return isinstance(nome, str) and nome != "" and " " not in nome
```

  Qui `_valida_nome_parametro` è un metodo statico (notare l'uso di un nome con underscore iniziale, convenzione per indicare uso interno). Possiamo chiamarlo sia dalla classe che da un'istanza: `Config._valida_nome_parametro("tema")` oppure `config_app._valida_nome_parametro("tema")` – il risultato è lo stesso.
- **Quando usarli**: I classmethod sono molto potenti per fornire istanze pre-configurate o per operare su dati condivisi. Ad esempio, in un contesto di analisi dati, si potrebbe avere una classe `Dataset` con un classmethod `from_csv(path)` che legge un file CSV e restituisce un oggetto Dataset. I staticmethod si usano più raramente; spesso si preferisce definire funzioni libere a livello di modulo. Si usano come detto per utilities legate concettualmente alla classe (ad esempio una funzione che calcola qualcosa relativo ai dati della classe ma che non necessita di un'istanza).

Properties e incapsulamento in Python (Slide 11)

Uso di @property per getter/setter Pythonic

In molti linguaggi OOP è comune dichiarare attributi privati e fornire **getter** e **setter** per controllarne l'accesso. In Python, la filosofia è diversa: per convenzione, un nome che inizia con `_` (singolo underscore) indica "implementazione interna" che non dovrebbe essere usata all'esterno della classe. Python non impedisce di accedere a `_attributo`, ma è un segnale per il programmatore. Per casi in cui è necessario proteggere l'accesso con logica aggiuntiva (validazione, calcolo pigro, ecc.), si usano le **property**.

- **Property**: Una property in Python è un attributo *calcolato* che appare dall'esterno come un normale attributo. Si definisce usando il decoratore `@property` su un metodo senza parametri (oltre a self). Il nome del metodo sarà il nome della property. Si possono poi definire metodi setter e deleter associati.
  Esempio: immaginiamo una classe `Dipendente` dove vogliamo memorizzare lo stipendio

annuale ma anche avere un attributo che ci dica lo stipendio mensile. Possiamo implementarlo come property:

```
class Dipendente:
    def __init__(self, nome, stipendio_annuo):
        self.nome = nome
        self._stipendio_annuo = stipendio_annuo  # attributo "privato"

    @property
    def stipendio_mensile(self):
        """Calcola lo stipendio mensile dividendo quello annuo per
12."""
        return self._stipendio_annuo / 12

    @stipendio_mensile.setter
    def stipendio_mensile(self, valore):
        """Permette di impostare lo stipendio mensile, aggiornando
quello annuo di conseguenza."""
        self._stipendio_annuo = valore * 12
```

Uso:

```
emp = Dipendente("Alice", 36000)
print(emp.stipendio_mensile)
# Accesso come attributo, chiama il getter -> 3000.0
emp.stipendio_mensile = 3500  # Uso come se fosse un attributo, chiama
il setter
print(emp._stipendio_annuo)   # Ora stipendio annuo è 42000 (3500*12)
```

Come si vede, dall'esterno `stipendio_mensile` si usa come un campo, ma dietro le quinte invoca i metodi definiti. Questo ci consente di incapsulare la logica (divisione/moltiplicazione per 12) e ad esempio aggiungere validazioni (potremmo non permettere stipendi negativi nel setter). Senza property, avremmo dovuto usare metodi tipo `get_stipendio_mensile()` e `set_stipendio_mensile(val)`, meno eleganti e meno integrati con lo stile Python.

- **Quando usare property**: quando l'accesso o modifica di un attributo richiede logica extra (validazione, calcolo derivato, notifica di eventi, etc.). Se l'accesso è diretto e sicuro, non serve wrapping – possiamo esporre direttamente l'attributo. Le property sono utili per mantenere un'interfaccia stabile mentre magari l'implementazione interna cambia. Ad esempio, potremmo decidere di tenere solo stipendio mensile internamente, calcolando l'annuo su richiesta: il codice esterno può continuare ad usare `.stipendio_mensile` e `.stipendio_mensile = x` e l'implementazione interna può essere modificata senza impatto esterno (grazie alle property il nome e l'uso restano uguali).
- **Proteggere attributi**: se vogliamo **veramente** evitare accessi diretti a un attributo interno, Python offre la convenzione del doppio underscore `__nome` all'interno di una classe. Questo attiva il *name mangling*: l'attributo `__dato` in realtà sarà accessibile come `_NomeClasse__dato` dall'esterno, rendendo meno probabile un accesso accidentale o una sovrascrittura da parte di sottoclassi. Tuttavia, l'uso di `__` è raro; nella pratica ci si affida a `_` singolo e al buon senso degli sviluppatori.

Conclusione Modulo 2 (parte 2)

In questa lezione abbiamo esplorato concetti avanzati di OOP in Python: ereditarietà, polimorfismo, metodi speciali, metodi di classe/statici e property. Abbiamo visto come Python permetta di modellare oggetti in modo potente e flessibile, mantenendo però convenzioni più leggere rispetto ad altri linguaggi (es. niente dichiarazioni formali di private/public, niente necessità di get/set per tutto). La chiave è usare queste funzionalità dove servono, per scrivere codice orientato agli oggetti che sia **chiaro, riusabile e manutenibile**.

Mettete in pratica i principi OOP appresi con i seguenti esercizi guidati:

1. **Classe semplice con property**:
   Creare una classe `Rettangolo` che rappresenti un rettangolo con **larghezza** e **altezza**.
2. Definire il costruttore `__init__(self, larghezza, altezza)` che imposta i valori.
3. Aggiungere una property `area` (uso di `@property`) che calcola l'area del rettangolo (larghezza × altezza) al volo. Questa sarà di sola lettura (non serve il setter).
4. Aggiungere una property `perimetro` di sola lettura che calcola il perimetro (2*(larghezza+altezza)).
5. Aggiungere un metodo `scala(fattore)` che moltiplica larghezza e altezza per un certo fattore (float o int).

6. **Verifica**: istanziare un paio di rettangoli e stampare area e perimetro, poi scalare e verificare che area e perimetro cambino in modo corretto.

7. **Ereditarietà e override**:
   Si realizzi una classe base `Dipendente` con attributi `nome` e `stipendio_annuo`. Include un metodo `descrivi()` che stampa "Dipendente [nome], stipendio annuo [stipendio]".

8. Creare una sottoclasse `Manager` che eredita da `Dipendente`. Oltre ai campi del Dipendente, aggiunge un attributo `bonus` (percentuale di bonus annuale). Sovrascrivere il metodo `descrivi()` affinché includa anche il bonus nella descrizione, ad esempio: "Manager [nome], stipendio annuo [stipendio], bonus [bonus]%".
9. In `Manager.__init__`, utilizzare `super().__init__(nome, stipendio)` per inizializzare nome e stipendio dalla classe base, e poi impostare il bonus.

10. **Verifica**: creare un oggetto `Dipendente` e uno `Manager`, chiamare `descrivi()` su entrambi e controllare l'output. Provare anche a aggiungere i due oggetti in una lista e iterare chiamando `descrivi()` (polimorfismo: entrambe le versioni dovrebbero attivarsi correttamente a seconda dell'oggetto).

11. **Metodi speciali __str__ e __eq__**:

12. Prendere la classe `Rettangolo` dell'esercizio 1. Implementare `__str__` in modo da restituire una stringa del tipo `Rettangolo [larghezza=x, altezza=y]`.
13. Implementare `__eq__` in modo che due rettangoli siano considerati uguali se hanno stessa larghezza e altezza.

14. **Verifica**: creare due rettangoli con gli stessi valori e verificarne l'uguaglianza con `==` e stamparli per vedere il formato definito da `__str__`. Mettere questi oggetti in un set per vedere che i duplicati (in base a eq) non vengono aggiunti (nota: perché ciò funzioni pienamente bisognerebbe anche definire `__hash__`, opzionale per bonus).

15. **Factory method (metodo di classe)**:

16. Aggiungere alla classe `Dipendente` un metodo di classe `da_stringa(cls, testo)` che riceve una stringa con formato `"Nome Cognome,stipendio"` (es: `"Mario Rossi,30000"`) e restituisce un oggetto `Dipendente` inizializzato con quei valori (attenzione al tipo di stipendio, va convertito a numero).
17. Utilizzare questo metodo di classe per creare un paio di oggetti Dipendente da stringhe di input e verificare che `descrivi()` mostri i dati correttamente.
18. **Extra**: far override di `da_stringa` in `Manager` per gestire eventualmente una stringa con bonus (es: `"Luigi Bianchi,50000,bonus:10"`), mostrando la flessibilità dei classmethod in ereditarietà. (Questo punto extra è avanzato: richiede di interpretare la stringa e capire a quale classe si riferisce, potete anche semplicemente mostrare come si potrebbe fare).