Recap

Sintassi e Strumenti Base

Walrus Operator (`:=`)

Permette di assegnare e usare una variabile in un'unica espressione.

```
if (n := len([1,2,3])) > 2:
    print(f"Lista con {n} elementi")  # Lista con 3 elementi
```

Strutture Dati

- **Lista** (ordinata, mutabile)

```
l = [1,2,3]; l.append(4); print(l[0])
```

- **Tupla** (ordinata, immutabile)

```
t = (1,2,3); print(t[1])
```

- **Set** (non ordinato, no duplicati)

```
s = {1,2,2,3}; print(s)  # {1,2,3}
```

- **Dizionario** (mappa chiave-valore)

```
d = {"a":1, "b":2}; print(d["a"])
```

Mutabilità: Attenzione

```
a = [1,2]; b = a; b.append(3)
print(a)  # [1,2,3] (puntano allo stesso oggetto)
```

Stringhe

```
s = "Python"
print(s.lower(), s.upper(), s[::-1])
```

Cicli

```
for i in range(3): print(i)
while (x := input("stop?")) != "y": print("ancora")
```

Funzioni

```
def f(x, y=1): return x + y
print(f(2))  # 3
```

⚠️ Evitare valori mutabili come default:

```
def bad(l=[]): l.append(1); return l
print(bad(), bad())  # [1], [1,1]
```

✅ Usare `None`:

```
def good(l=None):
    if l is None: l = []
    l.append(1); return l
```

File I/O

```
with open("test.txt","w") as f: f.write("ciao")
with open("test.txt") as f: print(f.read())
```

Error Handling

```
try:
    1/0
except ZeroDivisionError:
```

```
    print("Errore: divisione per zero")
finally:
    print("Sempre eseguito")
```

Paradigmi

- **Procedurale**: funzioni + dati separati.
- **OOP**: classi/oggetti con attributi + metodi.
- Pilastri: **incapsulamento, ereditarietà, polimorfismo**.

```
# Procedurale
saldo = 0
def deposita(x):  global saldo; saldo += x
def preleva(x):   global saldo; saldo -= x
deposita(100); preleva(40); print(saldo)

# OOP
class BankAccount:
    def __init__(self): self.balance = 0
    def deposit(self, x): self.balance += x
    def withdraw(self, x): self.balance -= x
acc = BankAccount(); acc.deposit(100); acc.withdraw(40); print(acc.balanc
e)
```

Classi e Oggetti

- **Classe**: blueprint.
- **Oggetto**: istanza con stato proprio.

```
class MyClass:
    def greet(self): print("Hello!")
obj = MyClass(); obj.greet()
```

Attributi

- **Istanza**: in `__init__`, unici per oggetto.
- **Classe**: definiti fuori dai metodi, condivisi.

```
class Demo:
    count = 0
    def __init__(self, v):
        self.value = v; Demo.count += 1
a, b = Demo(10), Demo(20)
print(a.value, b.value, Demo.count)
```

Metodi

- **Istanza**: con `self`.
- **Classe**: `@classmethod`, primo argomento `cls`.
- **Statici**: `@staticmethod`, non ricevono `self/cls`.

```
class Util:
    def m_istanza(self): return "istanza"
    @classmethod
    def m_classe(cls): return f"classe: {cls.__name__}"
    @staticmethod
    def m_statico(): return "statico"
```

Costruttore `__init__`

```
class Persona:
    def __init__(self, nome, eta=0):
        self.nome, self.eta = nome, eta
```

Rappresentazione (`__str__`, `__repr__`)

- `__str__`: leggibile per utenti.
- `__repr__`: non ambiguo, debug.

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

Incapsulamento

- `_attributo`: convenzione → interno.
- `__attributo`: name mangling → `_Classe__attr`.
- `@property`: getter/setter pythonic.

```
class Account:
    def __init__(self, saldo): self._saldo = saldo
    @property
    def saldo(self): return self._saldo
    @saldo.setter
    def saldo(self, v):
        if v<0: raise ValueError("negativo"); self._saldo=v
```

Ereditarietà

- Singola e multipla (risolta con MRO).
- `super()` richiama logica della superclasse.

```
class Veicolo:   def avvia(self): print("Veicolo")
class Auto(Veicolo):
    def avvia(self): super().avvia(); print("Auto pronta")
Auto().avvia()
```

Duck Typing & Interfacce

- Conta ciò che un oggetto **sa fare**.
- Interfacce informali → protocollo atteso.
- Formali con `abc.ABC`.

```
class DB: def connect(self): print("DB")
class Socket: def connect(self): print("Socket")
def connetti(x): x.connect()
```

Polimorfismo

Stessa interfaccia, implementazioni diverse.

```
class Cerchio: def area(self): return 3.14*2**2
class Quadrato:def area(self): return 4*4
for f in (Cerchio(), Quadrato()): print(f.area())
```

Composizione

Relazione “ha-un”: più flessibile di ereditarietà.

```
class Motore: def start(self): print("Motore")
class Auto:
    def __init__(self): self.m = Motore()
    def start(self): self.m.start(); print("Auto")
```

Classi Astratte

```
from abc import ABC,abstractmethod
class Forma(ABC):
    @abstractmethod
    def area(self): ...
```

Mixin

- Classi leggere usate in ereditarietà multipla.
- Riuso orizzontale, non istanziate da sole.

```
class ReprMixin:
    def __repr__(self): return f"{self.__class__.__name__}({self.__dict__})"
class User(ReprMixin):
    def __init__(self,n): self.n=n
```

Metaclassi

- “Classi di classi”, default = `type`.
- Permettono di controllare la creazione.

```
class MyMeta(type):
    def __new__(mcls,n,b,a): print("Creo",n); return super().__new__(mcls,n,
b,a)
class Test(metaclass=MyMeta): pass
```

Magic Methods

- Operatori: `__add__`, `__sub__`.
- Comparazione: `__eq__`, `__lt__`.
- Collezioni: `__len__`, `__getitem__`, `__iter__`.
- Callable: `__call__`.
- Contesti: `__enter__`, `__exit__`.

```
class Vettore:
    def __init__(s,x,y): s.x,s.y=x,y
    def __add__(s,o): return Vettore(s.x+o.x,s.y+o.y)
    def __len__(s): return abs(s.x)+abs(s.y)
```

Design Patterns

- **Factory**: delega la creazione.
- **Singleton**: una sola istanza globale.
- **Strategy**: algoritmi intercambiabili.

- **Observer**: notifica cambiamenti.
- **Decorator**: aggiunge comportamento dinamico.
- Altri: Adapter, Template Method, Proxy, Iterator, State, Visitor.

```
# Factory
class Circle: def draw(self): print("Cerchio")
def factory(t): return {"cerchio":Circle}.get(t)()
factory("cerchio").draw()
```

```
# Singleton
class Config:
    _i=None
    def __new__(cls): cls._i=cls._i or super().__new__(cls); return cls._i
```

```
# Strategy
def s_upper(s): return s.upper()
class Formatter:
    def __init__(self,strat): self.strat=strat
    def fmt(self,s): return self.strat(s)
```

```
# Observer
class Subject:
    def __init__(self): self.obs=[]
    def sub(self,f): self.obs.append(f)
    def set(self,v): [f(v) for f in self.obs]
```

Schema riassuntivo dei Design Pattern

I design pattern classici (GoF) sono divisi in **3 categorie principali**: *Creazionali*, *Strutturali*, e *Comportamentali*. (refactoring.guru)

---

1. Pattern Creazionali

Gestiscono la creazione degli oggetti in modo flessibile e controllato.

Pattern	Scopo principale	Note rapide
Singleton	Garantire che esista una sola istanza di una classe	Controlla il costruttore / istanza unica (Wikipedia)
Factory Method	Definire un'interfaccia per creare oggetti, delegando alle sottoclassi quale classe concreta istanziare	Evita dipendenze dirette dal tipo concreto (Wikipedia)
Abstract Factory	Gruppo di factory che producono famiglie correlate di oggetti	Permette creare prodotti correlati senza specificare le classi concrete
Builder	Costruzione passo-passo di oggetti complessi	Separare rappresentazione da costruzione (Wikipedia)
Prototype	Creare nuovi oggetti clonando un prototipo esistente	Utile quando la creazione è costosa

2. Pattern Strutturali

Si occupano di **come comporre classi e oggetti** in strutture più grandi mantenendo flessibilità.

Esempi:

- **Adapter** – “adatta” l’interfaccia di una classe esistente a quella che serve.
- **Bridge** – separa l’astrazione dall’implementazione, permettendo di variare le due indipendentemente.
- **Composite** – tratta oggetti singoli e aggregati in modo uniforme.
- **Decorator** – aggiunge dinamicamente responsabilità a oggetti senza modificare la classe originale.
- **Facade** – fornisce un’interfaccia semplificata per un insieme di classi complesse.
- **Flyweight** – riduce l’uso della memoria condividendo stati tra oggetti simili (Wikipedia).
- **Proxy** – fornisce un sostituto o un “controllo di accesso” a un oggetto reale.

3. Pattern Comportamentali

Definiscono modalità di comunicazione e responsabilità tra oggetti.

Esempi:

- **Strategy** – incapsula algoritmi intercambiabili e li rende intercambiabili dinamicamente.
- **Observer** – notifica automaticamente a più oggetti quando lo stato di uno cambia (Wikipedia).
- **Command** – incapsula una richiesta come un oggetto, permettendo di parametrizzare client con differenti richieste.
- **Iterator** – fornisce un modo standard per accedere sequenzialmente agli elementi di un oggetto aggregato.
- **Mediator** – coordina le comunicazioni tra oggetti, evitando che si riferiscano direttamente.
- **Memento** – salva e ripristina lo stato interno di un oggetto senza violare l’incapsulamento.
- **State** – permette a un oggetto di alterare il suo comportamento quando cambia il suo stato interno.
- **Template Method** – definisce lo scheletro di un algoritmo nella classe base, lasciando alcuni passi alle sottoclassi.
- **Visitor** – separa un'operazione dalla struttura su cui opera, permettendo definire nuove operazioni senza modificare le classi delle strutture.