

Corso Python

Modulo 4: Errori, Modularità e OOP

Collegamento al Modulo Precedente

Nel **Modulo 3** abbiamo imparato:

-  Controllo di flusso: `if`, `elif`, `else`
-  Cicli: `for`, `while`
-  Funzioni: `def`, parametri, `return`
-  Scope delle variabili

Ora il salto di qualità:

-  **Eccezioni:** Gestire gli errori elegantemente
-  **Moduli:** Organizzare codice in file
-  **Classi:** Creare i nostri tipi di dati

Agenda del Modulo (3 Ore)

In questo modulo faremo il salto di qualità verso la programmazione strutturata.

1. **Gestione Eccezioni:** Come evitare che il programma crashi (`try` , `except`).
2. **Moduli e Import:** Usare librerie esterne e organizzare i propri file.
3. **OOP (Object Oriented Programming):** Classi, Oggetti ed Ereditarietà.
4. **Laboratorio:** Sistema Bancario e altri esercizi.

1. Gestione delle Eccezioni

"Better safe than sorry"

Cos'è un'Eccezione?

Un'**eccezione** è un evento anomalo che interrompe il normale flusso di esecuzione del programma.

Differenza tra Errori ed Eccezioni:

- **Errori di sintassi:** Rilevati PRIMA dell'esecuzione (es. parentesi mancante)
- **Eccezioni:** Si verificano DURANTE l'esecuzione (runtime errors)

Perché esistono le eccezioni?

- Separano la logica principale dalla gestione degli errori
- Permettono di propagare errori attraverso più livelli del codice
- Forniscono informazioni dettagliate su cosa è andato storto

Senza gestione, il programma si **blocca** immediatamente mostrando un *traceback*.

Esempi di Eccezioni Comuni

```
# ZeroDivisionError: divisione per zero
print(10 / 0)

# ValueError: valore inappropriate per il tipo
int("ciao")

# IndexError: indice fuori dal range della lista
lista = [1, 2, 3]
print(lista[10])

# KeyError: chiave non presente nel dizionario
dizionario = {}
print(dizionario["x"])

# TypeError: operazione su tipi incompatibili
"ciao" + 5
```

Try-Except: La Base

Il blocco `try-except` permette di "catturare" gli errori e gestirli.

```
try:  
    # Codice che potrebbe generare errori  
    numero = int(input("Inserisci un numero: "))  
    risultato = 10 / numero  
    print(f"Risultato: {risultato}")  
except:  
    # Codice eseguito se c'è QUALSIASI errore  
    print("Si è verificato un errore!")
```

⚠️ Attenzione: Catturare tutti gli errori con `except: generico` è sconsigliato!

Catturare Eccezioni Specifiche

È buona pratica specificare il tipo di eccezione.

```
try:  
    numero = int(input("Inserisci un numero: "))  
    risultato = 10 / numero  
except ValueError:  
    # Scatta se l'utente scrive "ciao" invece di un numero  
    print("Errore: Devi inserire un numero intero!")  
except ZeroDivisionError:  
    # Scatta se l'utente inserisce 0  
    print("Errore: Non puoi dividere per zero!")
```

Ogni tipo di errore può essere gestito in modo diverso.

Catturare Più Eccezioni Insieme

```
try:  
    numero = int(input("Numero: "))  
    risultato = 10 / numero  
except (ValueError, ZeroDivisionError):  
    # Gestisce entrambi gli errori allo stesso modo  
    print("Input non valido!")  
  
# Oppure catturare l'errore in una variabile  
try:  
    numero = int(input("Numero: "))  
except ValueError as e:  
    print(f"Dettaglio errore: {e}")  
    # Output: invalid literal for int() with base 10: 'ciao'
```

Tipi di Eccezioni Comuni

Eccezione	Quando si verifica
ValueError	Valore inappropriate (es. <code>int("abc")</code>)
TypeError	Tipo sbagliato (es. <code>"2" + 2</code>)
IndexError	Indice lista fuori range
KeyError	Chiave dizionario non esistente
ZeroDivisionError	Divisione per zero
FileNotFoundException	File non trovato
AttributeError	Attributo/metodo non esistente
ImportError	Modulo non trovato

Il Blocco Else

Il blocco `else` viene eseguito **solo se NON ci sono eccezioni**.

Perché usare `else` invece di mettere il codice nel `try` ?

- Separa chiaramente il codice "a rischio" dal codice "sicuro"
- Evita di catturare eccezioni non previste
- Rende il codice più leggibile e intenzionale

```
try:  
    numero = int(input("Numero: "))  
    risultato = 10 / numero  
except ValueError:  
    print("Non è un numero!")  
except ZeroDivisionError:  
    print("Non dividere per zero!")  
else:  
    # Eseguito SOLO se tutto va bene  
    print(f"Risultato: {risultato}")  
    print("Operazione completata con successo!")
```

Il Blocco Finally

Il blocco `finally` viene eseguito **SEMPRE**, con o senza errori.

Quando viene eseguito `finally` ?

- Se il codice nel `try` ha successo
- Se viene sollevata un'eccezione (catturata o non)
- Anche se c'è un `return` nel `try` o `except`
- Anche se viene usato `break` o `continue` in un ciclo

```
try:  
    file = open("dati.txt", "r")  
    contenuto = file.read()  
except FileNotFoundError:  
    print("File non trovato!")  
finally:  
    # Eseguito SEMPRE (utile per chiudere risorse)  
    print("Tentativo di lettura completato.")
```

Struttura Completa Try-Except

```
try:  
    # Codice a rischio  
    risultato = operazione_rischiosa()  
except TipoErrore1:  
    # Gestione errore specifico 1  
    pass  
except TipoErrore2 as e:  
    # Gestione errore specifico 2 (con dettagli)  
    print(e)  
except Exception as e:  
    # Cattura tutti gli altri errori (ultima risorsa)  
    print(f"Errore imprevisto: {e}")  
else:  
    # Eseguito se NESSUN errore  
    print("Tutto OK!")  
finally:  
    # Eseguito SEMPRE  
    print("Fine operazione")
```

Raise: Sollevare Eccezioni

Possiamo **sollevarre** eccezioni volontariamente con `raise`.

```
def preleva(saldo, importo):
    if importo < 0:
        raise ValueError("L'importo non può essere negativo!")
    if importo > saldo:
        raise ValueError("Fondi insufficienti!")
    return saldo - importo

# Uso
try:
    nuovo_saldo = preleva(100, 150)
except ValueError as e:
    print(f"Errore: {e}")
# Output: Errore: Fondi insufficienti!
```

Eccezioni Personalizzate

Possiamo creare i nostri tipi di eccezione.

```
# Definizione eccezione personalizzata
class SaldoInsufficienteError(Exception):
    """Eccezione sollevata quando il saldo è insufficiente."""
    def __init__(self, saldo, importo):
        self.saldo = saldo
        self.importo = importo
        super().__init__(f"Saldo {saldo}€ insufficiente per prelevare {importo}€")

# Uso
def preleva(saldo, importo):
    if importo > saldo:
        raise SaldoInsufficienteError(saldo, importo)
    return saldo - importo

try:
    preleva(100, 150)
except SaldoInsufficienteError as e:
    print(e) # Saldo 100€ insufficiente per prelevare 150€
```

Assert: Verifiche di Debug

`assert` verifica una condizione e solleva `AssertionError` se falsa.

```
def calcola_media(numeri):
    assert len(numeri) > 0, "La lista non può essere vuota!"
    return sum(numeri) / len(numeri)

# Test
print(calcola_media([1, 2, 3])) # 2.0
print(calcola_media([])) # AssertionError: La lista non può essere vuota!
```

⚠️ Nota: Gli assert possono essere disabilitati con `python -O`. Non usarli per validazione input utente!

Pattern: Input Validation Loop

Pattern comune per ottenere input valido dall'utente.

```
def chiedi_numero(messaggio, minimo=None, massimo=None):
    while True:
        try:
            valore = int(input(messaggio))
            if minimo is not None and valore < minimo:
                print(f"Il valore deve essere ≥ {minimo}")
                continue
            if massimo is not None and valore > massimo:
                print(f"Il valore deve essere ≤ {massimo}")
                continue
            return valore
        except ValueError:
            print("Inserisci un numero intero valido!")

# Uso
eta = chiedi_numero("Inserisci età: ", minimo=0, massimo=120)
```



Esercizi Intermedi: Eccezioni

Pratica Prima di Proseguire

Esercizio 1.1: Conversione Sicura

Obiettivo: Crea una funzione che converte una stringa in numero gestendo gli errori.

```
def converti_numero(testo):
    """
    Converte una stringa in numero (int o float).
    Restituisce None se la conversione fallisce.
    """
    # Prova prima int, poi float
    # Gestisci ValueError e restituisci None in caso di errore
    pass
```

Esercizio 1.2: Accesso Sicuro a Dizionario

Obiettivo: Gestisci KeyError quando accedi a chiavi inesistenti.

```
utenti = {
    "mario": {"eta": 30, "email": "mario@email.it"},
    "luigi": {"eta": 25}
}
def get_email_utente(username):
    """
    Restituisce l'email dell'utente.
    Gestisce: utente non esistente, email non presente.
    """
    # Usa try-except per gestire KeyError
    # Restituisci un messaggio appropriato in ogni caso
    pass
```

Esercizio 1.3: Eccezione Custom

Obiettivo: Crea un'eccezione personalizzata per validare password.

```
class PasswordDebolError(Exception):
    """Eccezione per password non sicure."""
    pass
def valida_password(password):
    """
    Valida una password. Solleva PasswordDebolError se:
    - Meno di 8 caratteri
    - Nessun numero
    - Nessuna maiuscola
    """
    # Implementa le validazioni con raise
    pass
```

2. Moduli e Import

Non reinventare la ruota

Cos'è un Modulo?

Un **modulo** è un file `.py` contenente definizioni e istruzioni Python (funzioni, classi, variabili).

Perché usare i moduli?

1. **Organizzazione:** Dividere codice complesso in file più piccoli e gestibili
2. **Riutilizzo:** Scrivere una volta, usare ovunque (DRY - Don't Repeat Yourself)
3. **Namespace:** Evitare conflitti di nomi tra variabili/funzioni
4. **Manutenibilità:** Modificare una funzione in un solo posto
5. **Collaborazione:** Diversi sviluppatori possono lavorare su moduli diversi

Tipi di moduli

- **Moduli built-in:** Inclusi in Python (`math` , `os` , `sys`)
- **Moduli della libreria standard:** Installati con Python (`json` , `datetime`)
- **Moduli di terze parti:** Installabili con pip (`requests` , `numpy`)
- **Moduli propri:** Creati da te per il tuo progetto

Import Base

```
# Importa l'intero modulo
import math

# Uso: modulo.funzione()
print(math.sqrt(16))      # 4.0
print(math.pi)            # 3.141592...
print(math.ceil(4.2))     # 5
print(math.floor(4.8))    # 4

# Importa il modulo con un alias
import math as m

print(m.sqrt(16))         # 4.0
```

Import Selettivo

Importare solo ciò che serve.

```
# Importa funzioni/variabili specifiche
from math import sqrt, pi, pow

# Uso diretto (senza prefisso modulo)
print(sqrt(16))          # 4.0
print(pi)                 # 3.141592...
area = pi * pow(5, 2)     # 78.54...

# Import con alias
from math import sqrt as radice

print(radice(25))         # 5.0

# Import tutto (SCONSIGLIATO - inquina il namespace)
from math import *
```

Libreria Standard Python

Python ha una filosofia "Batteries Included".

```
import math      # Funzioni matematiche
import datetime # Date e orari
import os        # Sistema operativo
import sys       # Parametri sistema
import random   # Numeri casuali
import json      # Parsing JSON
import re        # Espressioni regolari
import collections # Strutture dati avanzate
import itertools # Strumenti per iterazione
import functools # Strumenti per funzioni
```

Modulo datetime

```
from datetime import datetime, date, timedelta

# Data e ora corrente
adesso = datetime.now()
print(adesso) # 2026-01-07 14:30:45.123456

# Solo data
oggi = date.today()
print(oggi) # 2026-01-07

# Formattazione
print(adesso.strftime("%d/%m/%Y %H:%M")) # 07/01/2026 14:30

# Parsing da stringa
data = datetime.strptime("25/12/2025", "%d/%m/%Y")

# Operazioni con date
domani = oggi + timedelta(days=1)
tra_una_settimana = oggi + timedelta(weeks=1)
```

Modulo os e pathlib

```
import os
from pathlib import Path

# os: operazioni su file e cartelle
print(os.getcwd())                      # Directory corrente
os.listdir(".")                          # Lista file nella cartella
os.path.exists("file.txt")              # Verifica esistenza
os.makedirs("nuova/cartella")          # Crea cartelle

# pathlib: approccio moderno (più pythonico)
cartella = Path(".")
for file in cartella.glob("*.py"):
    print(file.name)

# Costruire percorsi
percorso = Path("documenti") / "progetti" / "file.txt"
print(percorso.exists())
```

Modulo random

```
import random

# Numero casuale float tra 0 e 1
print(random.random())          # 0.7431...

# Numero intero in un range
print(random.randint(1, 100))    # Es: 42

# Scegliere elemento casuale
colori = ["rosso", "verde", "blu"]
print(random.choice(colori))    # Es: "verde"

# Mescolare una lista (in place)
numeri = [1, 2, 3, 4, 5]
random.shuffle(numeri)
print(numeri) # Es: [3, 1, 5, 2, 4]

# Campionare senza ripetizione
print(random.sample(range(100), 5)) # 5 numeri unici
```

Modulo json

```
import json

# Dizionario Python
dati = {
    "nome": "Mario",
    "eta": 30,
    "hobby": ["calcio", "lettura"]
}

# Python → JSON (serializzazione)
json_string = json.dumps(dati, indent=2)
print(json_string)
# JSON → Python (deserializzazione)
dati_recuperati = json.loads(json_string)

# Salvare su file
with open("dati.json", "w") as f:
    json.dump(dati, f, indent=2)
# Leggere da file
with open("dati.json", "r") as f:
    dati letti = json.load(f)
```

Creare i Propri Moduli

Crea un file `utils.py` :

```
# utils.py
"""Modulo con funzioni di utilità."""

PI = 3.14159

def saluta(nome):
    return f"Ciao, {nome}!"

def area_cerchio(raggio):
    return PI * raggio ** 2
```

Creare i Propri Moduli

Usalo in `main.py` :

```
# main.py
import utils

print(utils.saluta("Mario"))
print(utils.area_cerchio(5))
print(utils.PI)
```

name e main

Permette di distinguere se un file è eseguito direttamente o importato.

```
# utils.py
def saluta(nome):
    return f"Ciao, {nome}!"

# Questo blocco viene eseguito SOLO se il file è eseguito direttamente
if __name__ == "__main__":
    # Test del modulo
    print(saluta("Test"))
    print("Tutti i test passati!")
```

```
python utils.py      # Esegue il blocco __main__
python main.py      # Se main importa utils, il blocco NON viene eseguito
```

Pacchetti (Packages)

Un **pacchetto** è una cartella contenente moduli.

```
mio_progetto/
└── main.py
└── mio_pacchetto/
    ├── __init__.py      # Rende la cartella un pacchetto
    ├── modulo1.py
    └── modulo2.py
```

```
# main.py
from mio_pacchetto import modulo1
from mio_pacchetto.modulo2 import funzione_specifica

# Oppure
import mio_pacchetto.modulo1 as m1
```

pip: Anteprima

pip è il gestore di pacchetti di Python per scaricare librerie esterne.

```
# Comando base per installare una libreria  
pip install nome_libreria  
  
# Esempio  
pip install requests
```

Approfondiremo nel Modulo 5:

- Virtual environments (`venv`)
- `requirements.txt`
- Gestione completa delle dipendenze



Esercizi Intermedi: Moduli

Pratica Prima di Proseguire

Esercizio 2.1: Generatore di Password

Obiettivo: Usa i moduli `random` e `string` per generare password.

```
import random
import string
def genera_password(lunghezza=12, con_speciali=True):
    """
    Genera una password casuale.

    Args:
        lunghezza: numero di caratteri
        con_speciali: se includere caratteri speciali

    Returns:
        Stringa con la password generata
    """
    # Usa string.ascii_letters, string.digits, string.punctuation
    # Usa random.choice() o random.choices()
    pass
```

Esercizio 2.2: Statistiche su Date

Obiettivo: Usa il modulo `datetime` per calcolare statistiche.

```
from datetime import datetime, date

def giorni_al_compleanno(data_nascita_str):
    """
    Calcola i giorni mancanti al prossimo compleanno.

    Args:
        data_nascita_str: stringa formato "GG/MM/AAAA"

    Returns:
        Numero di giorni al prossimo compleanno
    """
    # Usa datetime.strptime() per parsare la data
    # Calcola il prossimo compleanno
    pass
```

Esercizio 2.3: Salvataggio Configurazione

Obiettivo: Usa il modulo `json` per salvare e caricare configurazioni.

```
import json

def carica_config(file_path):
    """
    Carica configurazione da file JSON.
    Restituisce dizionario vuoto se il file non esiste.
    """
    # Usa try-except per gestire FileNotFoundError
    pass
def salva_config(config, file_path):
    """Salva configurazione su file JSON."""
    pass
def aggiorna_config(file_path, chiave, valore):
    """Carica config, aggiorna una chiave, e salva."""
    pass
```

3. OOP: Object Oriented Programming

Classi e Oggetti

Cos'è la Programmazione Orientata agli Oggetti?

L'**OOP** è un paradigma di programmazione che organizza il software attorno a **oggetti** piuttosto che a funzioni e logica.

I 4 Pilastri dell'OOP:

1. **Incapsulamento:** Nascondere i dettagli interni, esporre solo l'interfaccia
2. **Astrazione:** Rappresentare concetti complessi in modo semplificato
3. **Ereditarietà:** Creare nuove classi basate su classi esistenti
4. **Polimorfismo:** Oggetti diversi rispondono allo stesso messaggio in modi diversi

Perché l'OOP?

La programmazione orientata agli oggetti permette di:

1. **Modellare** entità del mondo reale in modo naturale
2. **Organizzare** codice in modo logico e modulare
3. **Riutilizzare** codice tramite ereditarietà e composizione
4. **Incapsulare** dati e comportamenti insieme
5. **Manutenere** codice più facilmente nel tempo

```
# Senza OOP: dati separati e funzioni disconnesse
nome_utente = "Mario"
eta_utente = 30
email_utente = "mario@email.com"

# Con OOP: dati e comportamenti raggruppati logicamente
utente = Utente("Mario", 30, "mario@email.com")
utente.invia_email("Benvenuto!") # Il comportamento è legato ai dati
```

Classi vs Oggetti

L'OOP serve a modellare il mondo reale nel codice.

- **Classe (Class):** È il **progetto**, lo stampino, il "template"
 - Definisce la struttura (attributi) e il comportamento (metodi)
 - È un concetto astratto, non occupa memoria per i dati
- **Oggetto (Object/Istanza):** È la **realizzazione concreta** della classe
 - Ogni oggetto ha i propri valori per gli attributi
 - Può esistere un numero illimitato di oggetti dalla stessa classe

Analogia:

- Classe = Progetto architettonico di una casa
- Oggetto = Casa effettivamente costruita (ce ne possono essere molte)

Classi vs Oggetti: Esempio

Classe: Automobile

- Attributi: marca, modello, colore, km_percorsi
- Metodi: accendi(), frena(), accelera(), get_km()

Oggetti (istanze concrete):

- auto1 = Automobile("Fiat", "Panda", "Rosso")
- auto2 = Automobile("BMW", "Serie 3", "Nero")
- auto3 = Automobile("Fiat", "Panda", "Blu") # Stessa classe, oggetto diverso!

Ogni oggetto ha la **propria identità** e i **propri dati**, anche se creato dalla stessa classe.

Definire una Classe

```
class Cane:  
    # Attributo di CLASSE (condiviso da tutti)  
    specie = "Canis familiaris"  
  
    # Costruttore: inizializza l'oggetto  
    def __init__(self, nome, razza):  
        # Attributi di ISTANZA (specifici per ogni oggetto)  
        self.nome = nome  
        self.razza = razza  
  
    # Metodo (comportamento)  
    def abbaia(self):  
        print(f"{self.nome} dice: Woof!")  
  
# Creazione oggetti (istanziazione)  
fido = Cane("Fido", "Labrador")  
rex = Cane("Rex", "Pastore Tedesco")
```

self: Il Riferimento all'Istanza

`self` rappresenta l'oggetto stesso che sta chiamando il metodo.

```
class Contatore:  
    def __init__(self):  
        self.valore = 0  
  
    def incrementa(self):  
        self.valore += 1  
  
    def decrementa(self):  
        self.valore -= 1  
  
c1 = Contatore()  
c2 = Contatore()  
  
c1.incrementa()  
c1.incrementa()  
print(c1.valore) # 2  
print(c2.valore) # 0 (oggetto diverso!)
```

Attributi di Classe vs Istanza

```
class Studente:  
    # Attributo di CLASSE (condiviso)  
    scuola = "Liceo Einstein"  
    numero_studenti = 0  
  
    def __init__(self, nome):  
        # Attributo di ISTANZA (per ogni oggetto)  
        self.nome = nome  
        Studente.numero_studenti += 1  
  
s1 = Studente("Mario")  
s2 = Studente("Luigi")  
  
print(s1.scuola)          # "Liceo Einstein"  
print(s2.scuola)          # "Liceo Einstein"  
print(Studente.numero_studenti) # 2  
  
Studente.scuola = "Liceo Fermi" # Cambia per TUTTI  
print(s1.scuola)           # "Liceo Fermi"
```

Metodi Speciali (Dunder Methods)

Metodi con doppio underscore che definiscono comportamenti speciali.

```
class Punto:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __str__():  
        # Chiamato da print() e str()  
        return f"Punto({self.x}, {self.y})"  
  
    def __repr__():  
        # Rappresentazione "ufficiale" (per debug)  
        return f"Punto(x={self.x}, y={self.y})"  
  
p = Punto(3, 4)  
print(p)      # Punto(3, 4) - usa __str__  
print(repr(p)) # Punto(x=3, y=4) - usa __repr__
```

Metodi Speciali: Operatori

```
class Vettore:  
    def __init__(self, x, y):  
        self.x = x  
        self.y = y  
  
    def __add__(self, altro):  
        # Definisce v1 + v2  
        return Vettore(self.x + altro.x, self.y + altro.y)  
  
    def __eq__(self, altro):  
        # Definisce v1 == v2  
        return self.x == altro.x and self.y == altro.y  
  
    def __len__(self):  
        # Definisce len(v)  
        return int((self.x**2 + self.y**2)**0.5)  
  
v1 = Vettore(3, 4)  
v2 = Vettore(1, 2)  
v3 = v1 + v2      # Vettore(4, 6)  
print(len(v1))    # 5
```

Tabella Metodi Speciali

Metodo	Operatore/Funzione
<code>__init__</code>	Costruttore
<code>__str__</code>	<code>str()</code> , <code>print()</code>
<code>__repr__</code>	<code>repr()</code>
<code>__add__</code>	<code>+</code>
<code>__sub__</code>	<code>-</code>
<code>__mul__</code>	<code>*</code>
<code>__eq__</code>	<code>=</code>
<code>__lt__</code>	<code><</code>
<code>__len__</code>	<code>len()</code>
<code>__getitem__</code>	<code>obj[key]</code>

Incapsulamento: Il Concetto

L'**incapsulamento** è il principio di nascondere i dettagli implementativi e proteggere i dati interni.

Perché è importante?

- **Protezione:** Impedisce modifiche accidentali ai dati
- **Flessibilità:** Puoi cambiare l'implementazione interna senza rompere il codice esterno
- **Validazione:** Puoi controllare come i dati vengono modificati
- **Interfaccia chiara:** L'utente della classe sa esattamente cosa può usare

In Python: Non esiste un vero "private" come in Java/C++. Python usa **convenzioni** e **name mangling**.

Incapsulamento: Convenzioni Python

```
class ContoBancario:  
    def __init__(self, titolare, saldo):  
        self.titolare = titolare      # Pubblico: uso libero  
        self._saldo = saldo          # "Protetto": convenzione, non toccare  
        self.__pin = "1234"           # "Privato": name mangling attivo  
  
    def get_saldo(self):  
        return self._saldo  
  
    def deposita(self, importo):  
        if importo > 0:  # Validazione!  
            self._saldo += importo  
  
conto = ContoBancario("Mario", 100)  
print(conto.titolare)      # ✅ OK: pubblico  
print(conto._saldo)        # ⚠ Funziona ma SCONSIGLIATO  
# print(conto.__pin)        # ❌ AttributeError!  
print(conto._ContoBancario__pin)  # ⚠ Funziona (name mangling)
```

Property: Getter e Setter Eleganti

```
class Cerchio:
    def __init__(self, raggio):
        self._raggio = raggio
    @property
    def raggio(self):
        """Getter per raggio."""
        return self._raggio
    @raggio.setter
    def raggio(self, valore):
        """Setter con validazione."""
        if valore < 0:
            raise ValueError("Il raggio non può essere negativo!")
        self._raggio = valore
    @property
    def area(self):
        """Proprietà calcolata (solo getter)."""
        return 3.14159 * self._raggio ** 2

c = Cerchio(5)
print(c.raggio)      # 5 (usa getter)
c.raggio = 10         # Usa setter
print(c.area)        # 314.159 (calcolato)
```

Ereditarietà: Il Concetto

L'**ereditarietà** permette di creare nuove classi basate su classi esistenti.

Terminologia:

- **Classe Base/Padre/Superclasse:** La classe da cui si eredita
- **Classe Derivata/Figlia/Sottoclasse:** La classe che eredita

Cosa si eredita?

- Tutti gli attributi (di classe e istanza)
- Tutti i metodi
- La classe figlia può **aggiungere** nuovi attributi/metodi
- La classe figlia può **sovrascrivere** (override) metodi esistenti

Relazione "IS-A":

L'ereditarietà modella la relazione "è un": un Cane **è un** Animale.

Ereditarietà: Esempio

```
# Classe PADRE (Base)
class Animale:
    def __init__(self, nome):
        self.nome = nome

    def parla(self):
        print("...") # Comportamento generico

# Classe FIGLIA (Derivata) - eredita e specializza
class Cane(Animale):
    def parla(self): # Override del metodo
        print(f"{self.nome} dice: Woof!")

class Gatto(Animale):
    def parla(self): # Override del metodo
        print(f"{self.nome} dice: Miao!")

fido = Cane("Fido")
fido.parla() # Fido dice: Woof!
# fido.nome è ereditato da Animale!
```

super(): Chiamare il Padre

```
class Veicolo:
    def __init__(self, marca, modello):
        self.marca = marca
        self.modello = modello

    def descrivi(self):
        return f"{self.marca} {self.modello}"

class Auto(Veicolo):
    def __init__(self, marca, modello, num_porte):
        # Chiama il costruttore del padre
        super().__init__(marca, modello)
        self.num_porte = num_porte

    def descrivi(self):
        # Estende il metodo del padre
        base = super().descrivi()
        return f"{base} ({self.num_porte} porte)"

auto = Auto("Fiat", "Panda", 5)
print(auto.descrivi()) # Fiat Panda (5 porte)
```

Polimorfismo: Il Concetto

Il **polimorfismo** ("molte forme") permette a oggetti di classi diverse di rispondere allo stesso messaggio in modi diversi.

Perché è potente?

- **Flessibilità:** Scrivi codice che funziona con qualsiasi tipo che rispetta l'interfaccia
- **Estensibilità:** Aggiungi nuovi tipi senza modificare il codice esistente
- **Semplicità:** Non serve sapere il tipo esatto dell'oggetto

Duck Typing in Python:

"If it walks like a duck and quacks like a duck, then it must be a duck"

Python non controlla il tipo, controlla se l'oggetto ha il metodo richiesto.

Polimorfismo: Esempio

```
class Forma:  
    def area(self):  
        raise NotImplementedError("Implementa nelle sottoclassi")  
  
class Rettangolo(Forma):  
    def __init__(self, base, altezza):  
        self.base = base  
        self.altezza = altezza  
    def area(self):  
        return self.base * self.altezza  
  
class Cerchio(Forma):  
    def __init__(self, raggio):  
        self.raggio = raggio  
    def area(self):  
        return 3.14159 * self.raggio ** 2  
  
# Polimorfismo in azione: stesso codice, comportamenti diversi  
forme = [Rettangolo(10, 5), Cerchio(7)]  
for forma in forme:  
    print(f"Area: {forma.area()}") # Ogni forma sa calcolare la SUA area
```

Metodi di Classe e Statici

```
class Data:  
    def __init__(self, giorno, mese, anno):  
        self.giorno = giorno  
        self.mese = mese  
        self.anno = anno  
  
    @classmethod  
    def da_stringa(cls, stringa):  
        """Factory method: crea istanza da stringa."""  
        g, m, a = map(int, stringa.split("/"))  
        return cls(g, m, a) # cls = Data  
  
    @staticmethod  
    def is_bisestile(anno):  
        """Metodo statico: non usa self né cls."""  
        return anno % 4 == 0 and (anno % 100 != 0 or anno % 400 == 0)  
  
# Uso  
d1 = Data(25, 12, 2025)  
d2 = Data.da_stringa("01/01/2026") # Usa classmethod  
print(Data.is_bisestile(2024))      # True (staticmethod)
```

Classi Astratte: Il Concetto

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

- Non può essere istanziata direttamente
- Definisce un'**interfaccia** (contratto) che le sottoclassi devono rispettare
- Può contenere metodi astratti (senza implementazione) e metodi concreti

Quando usarle?

- Vuoi forzare le sottoclassi a implementare certi metodi
- Vuoi definire un comportamento comune ma lasciare i dettagli alle sottoclassi
- Vuoi creare una gerarchia di tipi con un'interfaccia comune

Classi Astratte: Esempio

```
from abc import ABC, abstractmethod

class Database(ABC): # Classe astratta
    @abstractmethod
    def connect(self): # DEVE essere implementato
        pass
    @abstractmethod
    def query(self, sql): # DEVE essere implementato
        pass
    def log(self, msg): # Metodo concreto (ereditato)
        print(f"[LOG] {msg}")

class MySQL(Database):
    def connect(self):
        print("Connesso a MySQL")
    def query(self, sql):
        print(f"MySQL esegue: {sql}")

# db = Database() # ✗ TypeError! Non si può istanziare
db = MySQL()      # ✅ OK: implementa tutti i metodi astratti
db.connect()
db.log("Operazione completata") # Metodo ereditato
```

Composizione vs Ereditarietà

Due modi per riutilizzare codice e costruire relazioni tra classi.

Ereditarietà: "È un" (is-a relationship)

- Un Cane **è un** Animale
- Crea una gerarchia di tipi
- Accoppiamento forte tra classi

Composizione: "Ha un" (has-a relationship)

- Un'Auto **ha un** Motore
- Oggetti contengono altri oggetti
- Accoppiamento più debole e flessibile

Regola pratica: "*Favor composition over inheritance*"

Preferisci la composizione quando possibile, usa l'ereditarietà solo per vere relazioni "è

Composizione: Esempio

```
# Composizione: un'Auto HA un Motore (non È un Motore)
class Motore:
    def __init__(self, cavalli):
        self.cavalli = cavalli
    def avvia(self):
        print("Motore avviato!")

class Auto:
    def __init__(self, marca, cavalli):
        self.marca = marca
        self.motore = Motore(cavalli) # COMPOSIZIONE: Auto contiene Motore
    def accendi(self):
        self.motore.avvia() # Delega al componente interno
        print(f"{self.marca} pronta a partire!")

auto = Auto("Ferrari", 800)
auto.accendi()
```

Vantaggio: Puoi cambiare il tipo di Motore senza toccare la classe Auto!

Dataclass (Python 3.7+)

Modo rapido per creare classi che contengono principalmente dati.

```
from dataclasses import dataclass

@dataclass
class Punto:
    x: float
    y: float

    def distanza_origine(self):
        return (self.x**2 + self.y**2)**0.5

# Genera automaticamente __init__, __repr__, __eq__
p1 = Punto(3, 4)
p2 = Punto(3, 4)

print(p1)          # Punto(x=3, y=4)
print(p1 == p2)    # True (confronta i valori)
print(p1.distanza_origine()) # 5.0
```

Dataclass Avanzate

```
from dataclasses import dataclass, field
from typing import List

@dataclass
class Studente:
    nome: str
    eta: int
    voti: List[int] = field(default_factory=list)
    matricola: str = field(default="N/A", repr=False)

    @property
    def media(self):
        return sum(self.voti) / len(self.voti) if self.voti else 0

@dataclass(frozen=True)  # Immutabile
class Coordinate:
    lat: float
    lon: float

s = Studente("Mario", 20)
s.voti.extend([28, 30, 25])
print(s.media)  # 27.67
```



Esercizi Intermedi: OOP

Pratica Prima di Proseguire

Esercizio 3.1: Classe Prodotto

Obiettivo: Crea una classe con metodi speciali.

```
class Prodotto:  
    """  
    Rappresenta un prodotto con nome, prezzo e quantità.  
  
    Implementa:  
    - __str__: rappresentazione leggibile  
    - __repr__: rappresentazione per debug  
    - __eq__: due prodotti sono uguali se hanno stesso nome  
    - valore_totale(): prezzo * quantità  
    """  
  
    def __init__(self, nome, prezzo, quantita=1):  
        # Completa  
        pass  
  
    # Test:  
p1 = Prodotto("Laptop", 999, 2)  
p2 = Prodotto("Laptop", 1200, 1)  
print(p1)                      # "Laptop: 999€ x 2"  
print(p1.valore_totale())       # 1998  
print(p1 == p2)                 # True (stesso nome)
```

Esercizio 3.2: Gerarchia Forme

Obiettivo: Usa ereditarietà e polimorfismo.

```
import math
class Forma:
    """Classe base per forme geometriche."""
    def area(self):
        raise NotImplementedError
    def perimetro(self):
        raise NotImplementedError
class Quadrato(Forma):
    def __init__(self, lato):
        # Completa
        pass
class Cerchio(Forma):
    def __init__(self, raggio):
        # Completa
        pass
# Test polimorfismo:
forme = [Quadrato(5), Cerchio(3)]
for f in forme:
    print(f"Area: {f.area():.2f}, Perimetro: {f.perimetro():.2f}")
```

Esercizio 3.3: Sistema di Utenti

Obiettivo: Usa property, ereditarietà e composizione.

```
from datetime import date
class Utente:
    """Utente base del sistema."""
    def __init__(self, username, email):
        self.username = username
        self._email = email
        self.data_registrazione = date.today()
    @property
    def email(self):
        return self._email
    @email.setter
    def email(self, valore):
        if "@" not in valore:
            raise ValueError("Email non valida")
        self._email = valore
class UtenteAdmin(Utente):
    """Utente con privilegi admin."""
    def __init__(self, username, email, livello=1):
        # Usa super() e aggiungi attributo livello
        pass
    def promuovi(self):
        """Aumenta il livello admin."""
        pass
```

4. Laboratorio Pratico

Esercizi su OOP e Eccezioni

Esercizio 1: Sistema Bancario

Obiettivo: Classe `ContoBancario` completa.

Requisiti:

1. Attributi: titolare, saldo, storico_operazioni
2. Metodi: deposita, preleva, trasferisci
3. Gestione errori con eccezioni personalizzate
4. Property per saldo (solo lettura)

Soluzione Esercizio 1 (Parte 1)

```
class SaldoInsufficientError(Exception):
    """Eccezione per saldo insufficiente."""
    pass

class ContoBancario:
    def __init__(self, titolare, saldo_iniziale=0):
        self.titolare = titolare
        self._saldo = saldo_iniziale
        self.storico = []

    @property
    def saldo(self):
        return self._saldo

    def deposita(self, importo):
        if importo < 0:
            raise ValueError("L'importo deve essere positivo!")
        self._saldo += importo
        self.storico.append(f"Deposito: +{importo}€")
```

Soluzione Esercizio 1 (Parte 2)

```
def preleva(self, importo):
    if importo <= 0:
        raise ValueError("L'importo deve essere positivo!")
    if importo > self._saldo:
        raise SaldoInsufficientError(
            f"Saldo {self._saldo}€ insufficiente per prelevare {importo}€"
        )
    self._saldo -= importo
    self.storico.append(f"Prelievo: -{importo}€")

def trasferisci(self, destinatario, importo):
    self.preleva(importo) # Può sollevare SaldoInsufficientError
    destinatario.deposita(importo)
    self.storico.append(f"Trasferimento a {destinatario.titolare}: -{importo}€")

# Test
conto1 = ContoBancario("Mario", 1000)
conto2 = ContoBancario("Luigi", 500)
conto1.trasferisci(conto2, 300)
print(conto1.saldo) # 700
print(conto2.saldo) # 800
```

Esercizio 2: Gerarchia Veicoli

```
class Veicolo:
    def __init__(self, marca, modello, anno):
        self.marca = marca
        self.modello = modello
        self.anno = anno
        self.acceso = False

    def accendi(self):
        self.acceso = True
        print(f"{self.marca} {self.modello} acceso")

    def spegni(self):
        self.acceso = False

class Auto(Veicolo):
    def __init__(self, marca, modello, anno, num_porte):
        super().__init__(marca, modello, anno)
        self.num_porte = num_porte

class Moto(Veicolo):
    def __init__(self, marca, modello, anno, cilindrata):
        super().__init__(marca, modello, anno)
        self.cilindrata = cilindrata
```

Esercizio 3: Sistema di Notifiche

```
from abc import ABC, abstractmethod

class Notificatore(ABC):
    @abstractmethod
    def invia(self, messaggio):
        pass

class EmailNotificatore(Notificatore):
    def __init__(self, email):
        self.email = email

    def invia(self, messaggio):
        print(f"Email a {self.email}: {messaggio}")

class SMSNotificatore(Notificatore):
    def __init__(self, telefono):
        self.telefono = telefono

    def invia(self, messaggio):
        print(f"SMS a {self.telefono}: {messaggio}")

# Polimorfismo
notificatori = [EmailNotificatore("test@email.com"), SMSNotificatore("123456")]
for n in notificatori:
    n.invia("Benvenuto!")
```

Esercizio 4: Inventario con JSON

```
import json

class Inventario:
    def __init__(self, file_path="inventario.json"):
        self.file_path = file_path
        self.prodotti = self._carica()

    def _carica(self):
        try:
            with open(self.file_path, "r") as f:
                return json.load(f)
        except FileNotFoundError:
            return {}

    def salva(self):
        with open(self.file_path, "w") as f:
            json.dump(self.prodotti, f, indent=2)

    def aggiungi(self, nome, quantita, prezzo):
        self.prodotti[nome] = {"quantita": quantita, "prezzo": prezzo}
        self.salva()
```

Riepilogo del Modulo

Concetto	Uso
try/except	Gestire errori senza crash
raise	Sollevare eccezioni volontariamente
import	Usare moduli esterni
Classe	Definire nuovi tipi di oggetti
init	Inizializzare attributi
Ereditarietà	Riutilizzare e estendere classi
@property	Getter/setter eleganti
@dataclass	Classi dati rapide

⚠ Errori Comuni: OOP e Eccezioni

Errore	Problema	Soluzione
<code>except: generico</code>	Cattura TUTTO, nasconde bug	Specifica il tipo <code>except ValueError</code>
Dimenticare <code>self</code>	Metodo non accede all'istanza	Primo parametro sempre <code>self</code>
Dimenticare <code>super()</code>	<code>__init__</code> padre non chiamato	<code>super().__init__(...)</code>
Attributo mutabile di classe	Condiviso tra istanze	Definisci in <code>__init__</code>

```
# ❌ Attributo mutabile di classe
class Squadra:
    membri = [] # CONDIVISO tra tutte le istanze!
```

```
# ✅ Corretto: definisci in __init__
class Squadra:
```



Progetto Cross-Modulo: Rubrica Contatti

Parte 4: Classe Contatto

```
class Contatto:  
    """Rappresenta un singolo contatto."""  
  
    def __init__(self, nome, telefono, email=""):  
        self.nome = nome  
        self.telefono = telefono  
        self.email = email  
  
    def __str__(self):  
        return f"{self.nome}: {self.telefono}"  
  
    def __repr__(self):  
        return f"Contatto('{self.nome}', '{self.telefono}', '{self.email}')"  
  
class Rubrica:  
    """Gestisce una collezione di contatti."""  
  
    def __init__(self):  
        self.contatti = []
```

Prossimo Modulo: File I/O, API e Pandas