Appunti di Programmazione Python

15 novembre 2025

Indice

Capitolo 1

Fondamenti Python

1.1 Introduzione

Questo modulo introduce i concetti fondamentali del linguaggio Python: esecuzione di script, tipi di dato di base, variabili, input/output, e buone pratiche di stile secondo PEP 8.

Inquadreremo quando usare l'interprete rispetto agli script, come scegliere i tipi corretti per rappresentare dati reali e come applicare un primo livello di ordine e leggibilità al codice. La descrizione è pensata per chi parte da zero o proviene da un altro linguaggio, con esempi brevi e ripetibili.

Al termine sarai in grado di impostare un ambiente funzionante, comprendere la sintassi di base e produrre piccoli programmi che leggono input, elaborano dati e presentano un output chiaro.

1.2 Obiettivi di Apprendimento

Al termine del modulo lo studente sarà in grado di:

- Eseguire script Python da riga di comando e in ambienti interattivi.
- Dichiarare e utilizzare variabili e tipi di dato nativi (int, float, str, bool, list).
- Gestire input da tastiera e output formattato.
- Applicare convenzioni di stile PEP 8 di base.

1.3 Concetti Fondamentali

Setup Ambiente

Installa Python (3.10+) e verifica con `python -version`. Usa un IDE (VS Code, PyCharm) o un editor leggero. Avvia l'interprete con `python` o esegui file con `python main.py`.

Stile e Convenzioni (PEP 8)

Nomi di variabili in minuscolo con underscore (*snake_ case*), righe massimo 79 caratteri, indentazione con 4 spazi, docstring per funzioni e moduli.

Concetti chiave

Tipi dinamici e conversioni (`int()`, `float()`, `str()`), validazione dell'input e gestione dei casi limite (vuoti, valori fuori range). Comprendere gli effetti collaterali (stampa, scrittura su file) rispetto a funzioni pure facilita test e manutenzione.

1.4 Esempi Pratici

1.4.1 Hello, World

```
print("Hello, World!")
```

Spiegazione del codice

- **Stampa**: ‘print‘ scrive su stdout e aggiunge newline finale per default. - **Stringhe**: le virgolette doppie delimitano un ‘str‘ Unicode; immutabile. - **Side-effect**: questo è I/O; separare la logica dai side-effect facilita i test.

1.4.2 Variabili e Tipi

```
nome = "Ada"
anni = 27
altezza = 1.68
studente = True

print(type(nome), type(anni), type(altezza), type(studente))
```

Spiegazione del codice

- **Binding**: variabili assegnano riferimenti a oggetti; il tipo è dell'oggetto. - **Tipi**: ‘str‘, ‘int‘, ‘float‘, ‘bool‘ sono built-in fondamentali. - **Ispezione**: ‘type(obj)‘ mostra la classe dell'oggetto; utile per debug rapido.

1.4.3 Input e Output Formattato

```
eta = int(input("Quanti anni hai? "))
print(f"Tra 5 anni avrai {eta + 5} anni.")
```

Spiegazione del codice

- **Input**: ‘input()‘ legge una riga da stdin e ritorna ‘str‘. - **Conversione**: ‘int(...)‘ traduce ‘str‘ a intero; solleva ‘ValueError‘ se non numerico. - **f-string**: interpolazione e formato inline (‘...‘); qui espressione aritmetica.

Casi d'uso

- Script CLI che raccolgono input e producono report.
- Piccoli tool per conversioni (temperature, valute, unità).
- Esempi didattici per introdurre sintassi e interazione.

1.5 Esercizi

1. Scrivi uno script che stampa il tuo nome e la tua età.
2. Dato un raggio `r`, calcola l'area del cerchio (πr^2).
3. Chiedi all'utente due numeri e stampa somma, differenza, prodotto, divisione.
4. Converte una temperatura da Celsius a Fahrenheit.
5. Crea una lista di tre stringhe e stampale una per riga.

1.6 Riepilogo

Hai visto la sintassi di base e come eseguire semplici programmi in Python, con variabili, tipi di dato e input/output.

1.7 Contesto e Applicazioni

Contesto e Applicazioni

- Script rapidi per automazione quotidiana. - Pulizia e normalizzazione di dati semplici. - Prototipi di funzioni e micro-tool didattici. - Problem solving e esercizi algoritmici.

1.8 Approfondimenti

Spiegazioni dettagliate

Questo modulo introduce le basi del linguaggio e del runtime Python. Alcuni punti essenziali:

- **Tipi di base e immutabilità**: interi e stringhe sono immutabili; liste e dizionari sono mutabili. Comprendere le implicazioni per aliasing e copie.
- **Verità implicita (truthiness)**: oggetti vuoti (`[]`, `{}`, `""`) e lo zero sono valutati come `False` in contesti booleani.
- **Slice e passi**: la notazione `seq[start:end:step]` è potente; usare `[:]` per ottenere una copia superficiale.
- **Script vs funzione**: isolare la logica in funzioni rende il codice testabile e riusabile; usare il guardiano `if __name__ == "__main__"` negli script.
- **Stile e leggibilità**: seguire PEP8 per nomi, indentazione, lunghezza delle righe; preferire nomi esplicativi rispetto ad abbreviazioni.
- **Docstring e aiuto**: documentare funzioni e moduli con docstring; `help(obj)` e `obj.__doc__` facilitano l'esplorazione.
- **Ambiente**: usare ambienti virtuali per isolare dipendenze; non installare globalmente salvo necessità.

Per chi viene da C/Java/JS

Esecuzione: Nessun `main()` obbligatorio; il file viene eseguito dall'alto verso il basso. Usa il guardiano `if __name__ == "__main__"` per l'entrypoint. **Variabili**: Non si dichiara il tipo (`int a`); il tipo è dell'oggetto. Usa annotazioni opzionali `x: int = 0` per comunicare intenti. **Stringhe**: Sono immutabili; preferisci f-string `f"{var}"` a concatenazioni. **Collezioni**: `list`/`dict`/`set` coprono gran parte degli use case. **Errori**: Eccezioni al posto di codici di ritorno; gestisci con `try/except`.
Esempio: traduzione di uno snippet JS in Python.

```
// JS
function sumPositives(arr) {
  let s = 0;
  for (const x of arr) if (x > 0) s += x;
  return s;
}

# Python
def sum_positives(arr):
    s = 0
    for x in arr:
        if x > 0:
            s += x
    return s

# Idiomatico
def sum_positives_fast(arr):
    return sum(x for x in arr if x > 0)
```

Idiomi e differenze pratiche

- **Enumerare**: ‘for i, x in enumerate(xs)‘ al posto di contatori manuali. - **Comprensioni**: trasformazioni compatte ‘[f(x) for x in xs if cond]‘. - **Packing/Unpacking**: ‘a, b = pair‘ e ‘*rest‘ per raccogliere. - **Path e I/O**: usa ‘pathlib‘ e context manager ‘with‘ per risorse. - **Test**: prediligi funzioni pure e ‘pytest‘ con fixture isolanti.

Copy vs alias Assegnazioni come `b = a` condividono il riferimento; modifiche su oggetti mutabili si riflettono su tutte le variabili alias. Per copiare: `list(a)`, `a[:]` , oppure `copy.copy`/`copy.deepcopy`.

Suggerimenti pratici Sfruttare il REPL per esplorazione rapida, creare snippet brevi, e salvare esempi riproducibili. Integrare l'uso di `timeit` per confrontare approcci alternativi. Consulta la documentazione ufficiale: https://docs.python.org/3/. Esplora `help(str)` e `dir(str)` nell'interprete per conoscere metodi utili.

Guida rapida per programmatori non-Python

Indentazione come sintassi: i blocchi (funzioni, cicli, condizioni) sono definiti da indentazione a 4 spazi. Niente parentesi graffe. **Tipi dinamici e duck typing**: il tipo è associato all'oggetto, non alla variabile; l'API attesa conta più della classe. **Truthiness**: oggetti vuoti e zero sono `False`; tutto il resto è `True`. **Eccezioni**: si gestiscono con `try/except`; evitare codici di errore. **Liste e dizionari**: strutture fondamentali per raccolte e mapping.

Esempio commentato:

```python
# Variabili e tipi
count = 0             # int
msg = "ok"           # str
values = [1, 2, 3]    # list
meta = {"k": 1}      # dict

# Condizioni basate su truthiness
if values:            # True se non vuota
    print("ha elementi")

# Funzione e slicing
def head(seq):
    """Ritorna il primo elemento o None se vuota."""
    return seq[0] if seq else None

print(head(values))  # 1
print(values[:2])    # copia dei primi 2 elementi

# Eccezioni per gestione errori
try:
    x = int("10")
except ValueError:
    x = 0
```

Spiegazione del codice

- **Collezioni**: ‘list‘ e ‘dict‘ sono mutabili; adatti a raccolte e mapping. - **Truthiness**: ‘if values:‘ verifica non-vuoto in modo idiomatico. - **Funzione**: ‘head‘ usa ritorno condizionale; il docstring descrive contratto. - **Eccezioni**: ‘try/except‘ evita codici di errore; ‘ValueError‘ in parsing.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 2

Controllo del Flusso

2.1 Introduzione

Questo modulo copre le strutture di controllo fondamentali in Python: condizioni, cicli e controllo del flusso dell'esecuzione.

La descrizione mette in evidenza quando preferire costrutti semplici e leggibili, come ridurre la complessità dei rami condizionali e come evitare loop inefficienti. Vedremo pattern comuni (guard clauses, early return) e insidie tipiche (condizioni annidate, off-by-one).

Capire bene il controllo del flusso è essenziale per scrivere codice chiaro, testabile e performante in scenari come elaborazione di liste, validazione di input e automazione di compiti ripetitivi.

2.2 Obiettivi di Apprendimento

- Utilizzare `if`, `elif`, `else` con operatori di confronto e logici.
- Scrivere cicli `for` e `while`, usare `break`, `continue`, `pass`.
- Iterare su sequenze e range in modo idiomatico.

2.3 Concetti Fondamentali

Operatori

Confronto: `==`, `!=`, `<`, `>`, `<=`, `>=`. Logici: `and`, `or`, `not`. Appartenenza: `in`, `not in`.

Precedenza e short-circuit

Gli operatori logici valutano da sinistra a destra e possono interrompere la valutazione (*short-circuit*). Usa parentesi per evitare ambiguità e preferisci condizioni semplici con guard clauses ed *early return* per migliorare leggibilità.

2.4 Esempi Pratici

2.4.1 Condizioni

```
1 eta = 18
2 if eta < 18:
3     print("Minorenne")
4 elif eta == 18:
5     print("Appena maggiorenne")
```

```
else:
    print("Maggiorenne")
```

Spiegazione del codice

- **Branching**: catene ‘if/elif/else‘ gestiscono casi esclusivi. - **Confronto**: ‘==‘ vs ‘<‘; preferire condizioni semplici e nomi espliciti. - **I/O**: la stampa è l’effetto; separare logica dal lato utente quando possibile.

2.4.2 Cicli for e while

```
# for su range
for i in range(3):
    print(i)

# while con condizione
count = 0
while count < 3:
    print(count)
    count += 1
```

Spiegazione del codice

- **range**: genera 0..2; preferire ‘enumerate(seq)‘ quando si itera su collezioni. - **while**: usa condizione booleana e aggiorna stato (‘count += 1‘). - **Chiarezza**: evitare ‘while True‘ con break se un range o un iterabile è sufficiente.

2.4.3 break e continue

```
for n in range(10):
    if n == 5:
        break
    if n % 2 == 0:
        continue
    print(n)  # stampa solo dispari prima del 5
```

Spiegazione del codice

- **break**: interrompe il ciclo quando ‘n == 5‘; utile per early exit. - **continue**: salta iterazioni per numeri pari; riduce annidamento. - **Intento**: il commento esplicita comportamento; preferire condizioni leggibili a logica implicita.

Casi d’uso

- Validazione di input con condizioni chiare e rami ben definiti.
- Elaborazione di liste/dizionari con loop e filtri.
- Workflow di automazione con controlli su stati e errori.

2.5 Esercizi

1. Data una temperatura, stampa "Freddo", "Tiepido" o "Caldo" usando `if/elif/else`.

2. Stampa i numeri da 1 a 100, sostituisci multipli di 3 con "Fizz" e di 5 con "Buzz".
3. Calcola la somma dei numeri pari da 1 a 50.
4. Chiedi numeri all'utente finché digita 0, poi stampa il totale.
5. Itera su una lista di stringhe e stampa solo quelle con lunghezza > 3.

2.6 Riepilogo

Hai visto le strutture di controllo principali per governare il flusso del programma.

2.7 Contesto e Applicazioni

Contesto e Applicazioni

- Orchestrare logiche di business e validazioni. - Implementare state machine e menu interattivi. - Gestire retry/backoff e circuit breaker semplificati. - Costruire pipeline di elaborazione condizionale.

2.8 Approfondimenti

Spiegazioni dettagliate

- **Condizionali**: usare `if/elif/else` con condizioni semplici; scomporre condizioni complesse in variabili intermedie per leggibilità.
- **Short-circuit**: `and` e `or` interrompono la valutazione non appena l'esito è determinato; sfruttarlo con attenzione.
- **Cicli**: `for` su sequenze/iterabili, `while` per condizioni generali; utilizzare `break`/`continue` con parsimonia.
- **Clausola else su for/while**: eseguita se il ciclo termina senza `break`; utile per rilevare mancate condizioni.
- **Range**: ricordare che `range(n)` esclude `n`; preferire enumeration con `enumerate` al mantenere contatori manuali.
- **Operatore walrus (:=)**: può rendere più espressivi alcuni pattern di lettura/validazione, evitare abusi che riducono la chiarezza.

Per chi viene da C/Java/JS: tradurre i pattern

for: in Python itera su elementi, non sugli indici; usa ‘enumerate(xs)‘ per avere coppie ‘(i, x)‘. **break/continue**: come in C/Java; la clausola ‘else‘ su ‘for/while‘ è eseguita se non avviene ‘break‘. **switch**: usa ‘if/elif‘ o ‘dict‘ di funzioni per dispatch. **Comprehension**: preferire ‘[f(x) for x in xs if cond]‘ a loop con accumulatore quando leggibile.
Esempi commentati:

```
# C/Java style (concettuale)
sum = 0
for (int i = 0; i < n; i++) {
    if (arr[i] % 2 == 0) continue;
```

```
    sum += arr[i];
}

# Python idiomatico
total = 0
for x in arr:
    if x % 2 == 0:
        continue
    total += x

# Python compatto: solo dispari
total = sum(x for x in arr if x % 2 == 1)

# for-else: cerca elemento; se non trovato, esegui fallback
target = 42
for x in arr:
    if x == target:
        print("trovato")
        break
else:
    print("non trovato")
```

Spiegazione del codice

- **Indice vs elemento**: il primo snippet mostra stile basato su indice; in Python si preferisce iterare su elementi (‘for x in arr‘). - **Filtri**: usare ‘continue‘ per saltare i pari; l’alternativa compatta usa generator expression in ‘sum(...)‘. - **for-else**: il blocco ‘else‘ si esegue solo se il ciclo non ha usato ‘break‘ (elemento non trovato); utile per ricerca con fallback. - **Stato**: ‘total‘ accumula; separare accumulo dalla logica di decisione semplifica i test.

Idiomi utili e anti-pattern

- **Guard clauses**: esci presto da funzioni per ridurre annidamento. - **Validazioni**: definisci predicate chiari, evita condizioni composte difficili da leggere. - **Accumulazione**: usa ‘sum‘, ‘any‘, ‘all‘ e ‘itertools‘ al posto di loop manuali quando opportuno. - **Anti-pattern**: loop con indice+lookup quando bastano iterabili; annidamenti profondi senza fattorizzare funzioni ausiliarie.

Pattern raccomandati Sostituire annidamenti profondi con return anticipati o funzioni ausiliarie. Separare la costruzione dei dati dalla loro validazione, riducendo la complessità cognitiva. Consulta: https://docs.python.org/3/tutorial/controlflow.html.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 3

Funzioni

3.1 Introduzione

Le funzioni sono blocchi riutilizzabili di codice. In Python si definiscono con `def`, possono avere parametri, valori di default, `*args` e `**kwargs`, e restituiscono valori con `return`.

Questa descrizione illustra come strutturare funzioni piccole e focalizzate, come documentarle con docstring e come scegliere le firme per rendere il codice facile da testare e riutilizzare. Discuteremo anche gli effetti dello scope e delle chiusure per evitare comportamenti inattesi.

Padroneggiare le funzioni consente di organizzare il progetto in unità logiche, ridurre duplicazioni e migliorare la leggibilità, con impatto diretto su manutenzione e qualità.

3.2 Obiettivi di Apprendimento

- Definire funzioni con parametri e valori di default.
- Usare `*args` e `**kwargs` per flessibilità negli argomenti.
- Comprendere la visibilità delle variabili (LEGB) e le docstring.

3.3 Concetti Fondamentali

Firma delle Funzioni

Esempio: `def greet(name: str, upper: bool = False) -> str: ...` Docstring: descrive comportamento, parametri e return.

Purezza ed effetti collaterali

Preferisci funzioni pure (stessa input → stessa output, senza effetti esterni) quando possibile. Riduci dipendenze globali e I/O nella logica, isolandole in layer dedicati per facilitare test e riuso.

3.4 Esempi Pratici

3.4.1 Funzione semplice

```
1 def add(a, b):
2     """Ritorna la somma di a e b."""
3     return a + b
4
```

```
print(add(2, 3))
```

Spiegazione del codice

- **Definizione**: ‘def add(a, b)‘ crea una funzione; il docstring descrive contratto. - **Return**: restituisce somma di due argomenti; pura, senza I/O. - **Invocazione**: ‘print(add(...))‘ compone I/O con logica; separare in pratica.

3.4.2 Default, *args e **kwargs

```
def greet(name, upper=False):
    msg = f"Ciao, {name}!"
    return msg.upper() if upper else msg

def total(*args):
    return sum(args)

def describe(**kwargs):
    return ", ".join(f"{k}={v}" for k, v in kwargs.items())

print(greet("Ada", upper=True))
print(total(1, 2, 3))
print(describe(lang="Python", version=3.11))
```

Spiegazione del codice

- **Default**: ‘upper=False‘ definisce parametro opzionale; evita default mutabili. - ***args**: raccoglie posizione variabile di argomenti in una tupla. - ****kwargs**: raccoglie coppie chiave/valore in un dizionario; utile per parametri nominati flessibili.

3.4.3 Scope e lambda

```
factor = 2
def mul(x):
    return x * factor  # legge dalla closure globale

double = lambda x: x * 2
print(mul(5), double(5))
```

Spiegazione del codice

- **Scope**: ‘mul‘ legge ‘factor‘ dallo scope esterno (E nel LEGB). - **Lambda**: funzioni anonime per espressioni semplici; preferire ‘def‘ per logiche complesse. - **Effetti**: nessun I/O dentro ‘mul‘; la stampa è separata.

Casi d’uso

- Librerie di utilità con funzioni riusabili e ben documentate.
- Calcoli e trasformazioni di dati separati dall’I/O.
- API interne con firme stabili e test unitari.

3.5 Esercizi

1. Scrivi una funzione che calcola il fattoriale di un numero (iterativa e ricorsiva).
2. Crea una funzione `safe_div(a, b)` che gestisce divisione per zero.
3. Implementa una funzione che normalizza una lista di numeri tra 0 e 1.
4. Usa `*args` per sommare un numero variabile di valori.
5. Scrivi una funzione che formatta una stringa con parametri via `**kwargs`.

3.6 Riepilogo

Hai imparato a definire funzioni, gestire parametri flessibili e comprendere scope e lambda.

3.7 Contesto e Applicazioni

Contesto e Applicazioni

- Progettare API di modulo e funzioni riutilizzabili. - Funzioni pure per testabilità e caching. - Higher-order functions per composizione. - Validazione e parsing input incapsulati.

3.8 Approfondimenti

Spiegazioni dettagliate

- **Parametri e firme**: preferire parametri nominali (keyword) per chiarezza; usare argomenti obbligatori per input essenziali.
- **Default mutabili**: evitare `def f(x=[])`; usare `None` e inizializzare internamente.
- **args e kwargs**: utili per passare collezioni di argomenti; documentare cosa viene accettato e quali chiavi sono supportate.
- **Docstring e annotazioni**: spiegare contratti, esempi d'uso e complessità; usare type annotations per favore l'inferenza e la comunicazione.
- **Purezza e effetti collaterali**: preferire funzioni pure quando possibile; isolare effetti (I/O, mutazioni) ai confini del sistema.
- **Closure e currying**: chiudere valori nell'ambiente per creare funzioni specializzate; attenzione alla cattura di variabili mutevoli.

Per chi viene da C/Java/JS: firme e scope

Annotazioni di tipo: sono opzionali e non vincolanti a runtime; servono a lettori e strumenti. **Keyword arguments**: chiamate come 'func(x=1, y=2)' aumentano la chiarezza rispetto agli argomenti posizionali. **LEGB scope**: Local, Enclosing, Global, Built-in: risoluzione dei nomi dall'interno verso l'esterno; evita dipendenze da globali. Esempi commentati:

```
# Default mutabile: anti-pattern
def append_item_bad(item, bucket=[]):
    bucket.append(item)
```

```
    return bucket

# Uso corretto
def append_item(item, bucket=None):
    if bucket is None:
        bucket = []
    bucket.append(item)
    return bucket

# Keyword arguments e type hint
from typing import Iterable, Callable

def transform(xs: Iterable[int], fn: Callable[[int], int]) -> list[
    int]:
    return [fn(x) for x in xs]

data = [1, 2, 3]
res = transform(xs=data, fn=lambda x: x * 2)
```

Spiegazione del codice

- **Default mutabile**: ‘bucket=[]‘ condivide lista tra chiamate; bug classico. - **Pattern corretto**: usa ‘None‘ e inizializza all’interno per isolare stato. - **Type hints**: ‘Iterable[int]‘ e ‘Callable‘ comunicano contratto; non impongono a runtime. - **Keyword args**: chiamata ‘transform(xs=data, fn=...)‘ aumenta chiarezza dei parametri.

Idiomi: funzioni piccole, composizione, map/filter

- **Composizione**: costruisci pipeline con funzioni pure e piccole. - **map/filter**: usa ‘map(f, xs)‘/‘filter(pred, xs)‘ oppure comprensioni per leggibilità. - **Docstring**: inserisci esempio d’uso e contratti; evita commenti superflui dentro la funzione.

```
def normalize(xs):
    """Riporta i valori in [0, 1]."""
    m = max(xs) or 1
    return [x / m for x in xs]

def pipeline(xs):
    return normalize([x for x in xs if x > 0])

print(pipeline([-2, 3, 6]))  # [0.5, 1.0]
```

Spiegazione del codice

- **Docstring**: descrive contratto e output atteso. - **max or 1**: fallback per evitare divisione per zero su lista vuota. - **Comprensione**: crea nuova lista normalizzata; ‘pipeline‘ mostra composizione filtrando positivi.

Composizione Creare pipeline di funzioni piccole e testabili; separare parsing, validazione e trasformazione, restituendo strutture dati chiare. Consulta: https://docs.python.org/3/tutorial/controlflow.html#defining-functions e PEP 8 per docstring.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘:‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 4

Strutture Dati

4.1 Introduzione

Le strutture dati di base in Python includono liste, tuple, dizionari e insiemi. Ogni tipo ha proprietà e operazioni utili per modellare e manipolare dati.

La descrizione guida alla scelta consapevole del tipo in base a mutabilità, costi operativi e chiarezza dell'intenzione: quando usare un dizionario per accesso rapido, un set per unicità o una tupla per dati immutabili. Vedremo idiomi Pythonici come comprensioni e unpacking.

Una buona padronanza delle strutture dati rende le soluzioni più semplici, efficienti e robuste in problemi quotidiani di elaborazione, trasformazione e aggregazione.

4.2 Obiettivi di Apprendimento

- Usare liste, tuple, dizionari e insiemi con metodi principali.
- Comprendere mutabilità/immutabilità e scegliere il tipo adeguato.
- Iterare e trasformare dati in modo idiomatico.

4.3 Concetti Fondamentali

Panoramica

Lista (mutabile), Tupla (immutabile), Dizionario (mappa chiave → valore), Insieme (collezione non ordinata di elementi unici).

Complessità e prestazioni

Liste: append/iterazione veloci; Dizionari/insiemi: lookup/inserimento mediamente O(1); Tuple: immutabili e leggere. Scegli in base a mutabilità, necessità di ordine e costi operativi.

4.4 Esempi Pratici

4.4.1 Liste e comprensioni

```
1 nums = [1, 2, 3, 4]
2 nums.append(5)
3 quadrati = [n*n for n in nums]
4 print(quadrati)
```

Spiegazione del codice

- **Lista mutabile**: ‘append‘ aggiunge in coda in O(1) amortizzato. - **Comprensione**: crea una nuova lista trasformando elementi (‘n*n‘). - **Separazione**: calcolo e I/O (‘print‘) sono distinti per chiarezza.

4.4.2 Tuple e unpacking

```
pt = (10, 20)
x, y = pt
print(x, y)
```

Spiegazione del codice

- **Tupla immutabile**: buona per record fissi e chiavi di dizionari. - **Unpacking**: assegna posizionalmente; solleva ‘ValueError‘ se la cardinalità non coincide. - **Chiarezza**: nomi ‘x, y‘ rendono esplicita la struttura dei dati.

4.4.3 Dizionari

```
info = {"nome": "Ada", "eta": 27}
info["lingua"] = "Python"
for k, v in info.items():
    print(k, v)
```

Spiegazione del codice

- **Insert/Update**: assegnazione su chiave crea/aggiorna coppie chiave → valore. - **items**: itera su coppie ‘(k, v)‘ evitando lookup ripetuti. - **Ordine**: dal 3.7 l’ordine d’inserimento è preservato; non confonderlo con ordinamento.

4.4.4 Insiemi

```
u = {1, 2, 3}
v = {3, 4}
print(u | v, u & v, u - v)
```

Spiegazione del codice

- **Unione/intersezione/differenza**: `|`, `&`, - operano su insiemi con semantica matematica. - **Unicità**: i set eliminano duplicati; elementi devono essere hashable. - **Prestazioni**: membership e operazioni insiemistiche sono mediamente O(1)/O(n).

Casi d’uso

- Rimozione di duplicati e membership veloce con set.
- Indici e lookup rapidi con dizionari.
- Rappresentazione compatta e immutabile con tuple.

4.5 Esercizi

1. Dato un elenco di parole, rimuovi duplicati mantenendo l'ordine.
2. Conta la frequenza dei caratteri in una stringa con un dizionario.
3. Data una lista di tuple (nome, punteggio), stampa il top 3.
4. Usa un set per trovare l'intersezione tra due liste.
5. Trasforma una lista di dizionari in un dizionario indicizzato per una chiave.

4.6 Riepilogo

Hai esplorato le principali strutture dati e le loro operazioni tipiche.

4.7 Contesto e Applicazioni

Contesto e Applicazioni

- Modellare dati di dominio (liste, dizionari, set). - Cache LRU e code con `deque`. - Rilevare duplicati e membership con `set`. - Mappare configurazioni e lookup con `dict`.

4.8 Approfondimenti

Spiegazioni dettagliate

- **Liste vs tuple**: liste per collezioni mutabili; tuple per record immutabili e chiavi di dizionari.
- **Insiemi**: utili per eliminare duplicati e test di appartenenza rapidi (hash-based); attenzione agli oggetti non hashable.
- **Dizionari**: mapping flessibile; sfruttare `dict.get`, `defaultdict` e `Counter` per pattern frequenti.
- **Comprensioni**: preferire list/dict/set comprehension a cicli espliciti per trasformazioni; mantenere espressioni leggibili.
- **Copia superficiale/profonda**: distinguere tra copia dei contenitori e copia degli elementi; usare `copy` e `deepcopy` consapevolmente.
- **Complessità**: conoscere i costi di operazioni chiave (append, pop, membership) per evitare colli di bottiglia.

Per chi viene da C/Java/JS: mapping, set e ordinamenti

dict: come una HashMap/oggetto JS ma con metodi ricchi (‘items‘, ‘keys‘, ‘values‘); le chiavi devono essere hashable (immutabili). **set**: come HashSet, elimina duplicati e offre operazioni insiemistiche (`|`, `&`, `-`). **ordinamenti**: ‘sorted(xs, key=...)‘ ordina senza mutare; ‘list.sort()‘ ordina in-place.
Esempi commentati:

```
1 from collections import defaultdict, Counter
```

```
2
3  # Conteggi frequenze
4  c = Counter("abracadabra")
5  print(c.most_common(2))  # [('a', 5), ('b', 2)]
6
7  # Indice per chiave
8  people = [
9      {"name": "Ada", "age": 27},
10     {"name": "Bob", "age": 33},
11 ]
12 by_name = {p["name"]: p for p in people}
13
14 # set per unione/intersezione
15 a = {1, 2, 3}
16 b = {3, 4}
17 print(a | b, a & b, a - b)
18
19 # defaultdict per accumuli
20 acc = defaultdict(list)
21 for p in people:
22     acc[p["age"]].append(p["name"])
23
24 # ordinare per chiave
25 print(sorted(people, key=lambda p: (p["age"], p["name"])) )
```

Spiegazione del codice

- **Counter**: conteggio frequenze e `most_common` per ranking rapido. - **Comprehension dict**: crea indice per chiave evitando loop manuali. - **defaultdict**: inizializza accumulatori riducendo boilerplate. - **sorted key**: ordinamento stabile su tuple di chiavi senza mutare l'input.

Idiomi: comprensioni, unpacking, generatori

- **Comprensioni**: ‘[f(x) for x in xs if cond]‘ per trasformazioni rapide. - **Unpacking**: ‘a, b = pair‘, ‘first, *rest = xs‘. - **Generatori**: ‘(... for x in xs)‘ per elaborazioni lazy e ridotto uso memoria; usa ‘sum‘, ‘any‘, ‘all‘ su generatori.

Idiomi utili Sfruttare `zip`, `sorted(key=...)` e generatori per elaborare sequenze in modo espressivo e efficiente. Consulta: https://docs.python.org/3/tutorial/datastructures.html.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘:‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 5

Stringhe e Formattazione

5.1 Introduzione

Le stringhe sono sequenze di caratteri immutabili. In Python si manipolano con operatori, metodi e formattazione moderna.

Questa descrizione copre casi d'uso realistici: pulizia dati, normalizzazione, parsing e produzione di messaggi leggibili per log e interfacce. Accenniamo alle problematiche di encoding Unicode, ai separatori locali e alla formattazione numerica e di date.

Saper lavorare bene con le stringhe è centrale in scripting, reportistica, trasformazioni CSV/JSON e interazione con utenti e sistemi.

5.2 Obiettivi di Apprendimento

- Usare metodi delle stringhe comuni (strip, split, join, replace).
- Formattare output con f-string e format.
- Comprendere slicing e manipolazioni di base.

5.3 Concetti Fondamentali

F-string

Esempio: `name = "Ada"; f"Ciao, name!"`

Specifiche di formattazione

Numeri: `:.2f` per due decimali, `:08d` per zero-padding; Testo: `:<20` allineamento a sinistra, `:.10` taglio; Date: usa `datetime.strftime`. Mantieni separata la logica di formattazione dalla logica di calcolo.

5.4 Esempi Pratici

5.4.1 Metodi principali

```
s = "  Python e' fantastico  "
print(s.strip().replace("fantastico", "potente"))
```

Spiegazione del codice

- **Immutabilità**: i metodi su ‘str‘ ritornano nuove stringhe; ‘s‘ resta invariata. - **Pulizia**: ‘strip‘ rimuove spazi ai bordi; ‘replace‘ sostituisce occorrenze. - **Pipeline**: concatenare metodi rende leggibile la trasformazione testuale.

5.4.2 Split/Join

```
frase = "uno,due,tre"
parts = frase.split(",")
joined = ";".join(parts)
print(joined)
```

Spiegazione del codice

- **Tokenizzazione**: ‘split‘ crea lista di parti; gestire separatori coerenti. - **Ricombinazione**: ‘";".join(parts)‘ unisce con separatore scelto; richiede elementi ‘str‘. - **CSV semplice**: valido per casi basici; preferire ‘csv‘ per casi complessi.

5.4.3 Formattazione

```
user = "Ada"; score = 99.5
print(f"{user} ha punteggio {score:.1f}")
print("{} ha punteggio {}".format(user, score))
```

Spiegazione del codice

- **f-string**: interpolazione con specifiche (‘:.1f‘ per decimali). - **format**: alternativa compatibile; utile con template riciclati. - **Separazione**: tenere formattazione vicino all’output, logica separata.

Casi d’uso

- Generazione di report e output leggibili per utenti.
- Normalizzazione di input/CSV/log per pipeline dati.
- Template di messaggi ed etichette in UI/CLI.

5.5 Esercizi

1. Normalizza una stringa: rimuovi spazi, porta a minuscolo.
2. Sostituisci tutte le vocali in una stringa con `*`.
3. Dato un CSV di parole, trasformalo in lista e riuniscilo con separatore `|`.
4. Estrai il dominio da un indirizzo email.
5. Verifica se una stringa è palindroma usando slicing.

5.6 Riepilogo

Hai visto metodi delle stringhe e tecniche moderne di formattazione.

5.7 Contesto e Applicazioni

Contesto e Applicazioni

- Pulizia e normalizzazione testi (log, CSV, input utente). - Formattazione report e messaggi per CLI/GUI. - Localizzazione e formattazione numeri/date. - Sanitizzazione input e rendering sicuro.

5.8 Approfondimenti

Spiegazioni dettagliate

- **f-string vs `format`**: preferire f-string per leggibilità; `format` resta utile per formattazioni avanzate.
- **Precisione e allineamento**: specificare `:.2f` per decimali, `:>10` per allineamento; gestire numeri grandi con separatori.
- **Localizzazione**: separare formattazioni dipendenti dalla lingua; evitare concatenazioni manuali, preferendo template.
- **Sanitizzazione**: evitare injections e errori; normalizzare input (trim, lower) e validare prima di formattare.
- **Unicode ed encoding**: usare `utf-8`; attenzione a normalizzazione (NFC/NFD) per confronti affidabili.

Per chi viene da C/Java/JS: stringhe, bytes ed encoding

Stringhe: sono immutabili e Unicode; operazioni come ‘replace‘, ‘split‘, ‘join‘ sono metodi sul tipo ‘str‘. **Formattazione**: preferisci f-string (‘f"x:.2f"‘) per chiarezza; ‘format‘ utile per template generici. **Bytes**: ‘bytes‘/‘bytearray‘ rappresentano dati binari; conversione via ‘s.encode("utf-8")‘ e ‘b.decode("utf-8")‘. **Regex**: usa ‘re‘ per parsing robusto; separa regole dai messaggi di output.
Esempi commentati:

```
# Normalizzazione testo
def normalize(s):
    s = s.strip().lower()
    return " ".join(s.split())  # collassa spazi multipli

# Formattazione numeri
amount = 1234.567
print(f"{amount:,.2f}")  # separatori dipendono dalla locale

# String vs bytes
data = "ciao".encode("utf-8")   # bytes
text = data.decode("utf-8")       # str

# Regex (semplice email)
import re
pat = re.compile(r"^[^@\s]+@[^@\s]+\.[^@\s]+$")
print(bool(pat.match("ada@example.com")))
```

Spiegazione del codice

- **Normalizzazione**: trim + lower + collapse spazi per confronti affidabili. - **Formattazione numeri**: specifiche di locale possono cambiare separatori. - **Stringhe/bytes**: ‘encode‘/‘decode‘ per confine testo → binario. - **Regex**: pattern minimo per email; preferire librerie se requisiti complessi.

Idiomi: split/join, f-string, template

- **split/join**: ‘";".join(parts)‘ e ‘s.split(",")‘ per CSV semplici. - **f-string**: espressioni inline e specifiche di formato; evita concatenazioni con ‘+‘. - **Template**: centralizza formattazioni e messaggi; separa logica dai testi.

Suggerimenti Per testo lungo in codice, valutare `\lstinline` per parti inline e spezzare stringhe multi-linea con triple quotes mantenendo indentazione chiara. Consulta: https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘:‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 6

File e Gestione I/O

6.1 Introduzione

Operazioni di lettura/scrittura su file e gestione risorse con context manager.

Descriviamo flussi tipici di I/O: aprire file in sicurezza, trattare encoding e newline multipiattaforma, leggere dati strutturati (CSV) e scrivere output affidabile. Evidenziamo l'uso di `with` per garantire rilascio delle risorse.

Queste competenze abilitano pipeline di dati, logging persistente e integrazione con altri sistemi (configurazioni, esportazioni, archiviazione).

6.2 Obiettivi di Apprendimento

- Aprire, leggere e scrivere file di testo e CSV.
- Usare `with` per gestire risorse.
- Gestire percorsi e encoding.

6.3 Concetti Fondamentali

Context Manager

Usa `with open("file.txt", "r", encoding="utf-8") as f: ...`

Percorsi e filesystem

Preferisci `pathlib` per percorsi portabili (`Path("dir") / "file.txt"`). Attenzione a separatori e encoding tra sistemi; valida l'esistenza dei file e gestisci directory in modo sicuro.

6.4 Esempi Pratici

6.4.1 Lettura file

```
with open("dati.txt", "r", encoding="utf-8") as f:
    for line in f:
        print(line.strip())
```

Spiegazione del codice

- **with open**: garantisce chiusura automatica del file, anche su eccezione. - **Iterazione**: il file è iterabile per riga; ‘strip‘ rimuove newline/spazi ai bordi. - **Encoding**: esplicitare ‘utf-8‘ evita problemi di piattaforma.

6.4.2 Scrittura file

```
with open("out.txt", "w", encoding="utf-8") as f:
    f.write("Hello\n")
```

Spiegazione del codice

- **Modalità**: ‘"w"‘ sovrascrive il file (crea se non esiste). - **Buffer**: ‘write‘ aggiunge al buffer; flush/chiusura avvengono alla fine del ‘with‘. - **Newline**: usa \n come separatore universale; LaTeX e Windows gestiscono conversioni.

6.4.3 CSV

```
import csv
rows = [("nome", "eta"), ("Ada", 27)]
with open("people.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerows(rows)
```

Spiegazione del codice

- **csv.writer**: gestisce quoting/separatori corretti; evitare ‘join‘ per CSV complessi. - **newline=""**: delega al modulo CSV la gestione dei terminatori di riga. - **Schema**: definire header coerenti e tipi attesi aumenta robustezza.

Casi d’uso

- Pipeline di elaborazione file (log, CSV, esportazioni).
- Gestione configurazioni e archiviazione semplice.
- Generazione di report testuali e tracciamento attività.

6.5 Esercizi

1. Conta linee e parole in un file di testo.
2. Copia il contenuto di un file in un altro.
3. Leggi un CSV e stampa solo le righe con età > 30.
4. Scrivi un log con timestamp per ogni riga letta.
5. Gestisci errori di file mancante in lettura.

6.6 Riepilogo

Hai praticato I/O su file e uso di context manager.

6.7 Contesto e Applicazioni

Contesto e Applicazioni

- Lettura/scrittura CSV per piccoli ETL locali. - Gestione log e rotazione file. - Backup e archiviazione di configurazioni. - Import/export per strumenti CLI.

6.8 Approfondimenti

Spiegazioni dettagliate

- **Modalità di apertura**: `'r'`, `'w'`, `'a'`, `'rb'`/`'wb'`; scegliere testo o binario in base ai dati.
- **Context manager**: usare `with open(...) as f` per chiusura automatica e gestione sicura delle risorse.
- **Encoding**: esplicitare `encoding='utf-8'`; gestire errori con `errors='replace'` quando necessario.
- **Buffering e newline**: controllare buffering per prestazioni; gestire newlines con `newline` su piattaforme diverse.
- **Pathlib**: preferire `pathlib.Path` a `os.path` per operazioni leggibili e composabili.

Per chi viene da C/Java/JS: filesystem e flussi

Percorsi: usa ‘pathlib.Path‘ per comporre percorsi portabili (‘Path("dir") / "file.txt"‘). **Testo vs binario**: ‘"r"/"w"‘ per testo con encoding, ‘"rb"/"wb"‘ per dati binari grezzi. **Sicurezza**: gestisci eccezioni come ‘FileNotFoundError‘, ‘PermissionError‘; evita perdita di risorse usando ‘with‘. **CSV/JSON**: usa moduli standard ‘csv‘ e ‘json‘; definisci schema e validazione separata dalla lettura/scrittura. **Temporary files**: usa ‘tempfile‘ per file temporanei sicuri.
Esempi commentati:

```
from pathlib import Path
import json, csv

base = Path("data")
base.mkdir(exist_ok=True)

# Scrivi JSON
payload = {"name": "Ada", "age": 27}
with (base / "user.json").open("w", encoding="utf-8") as f:
    json.dump(payload, f, ensure_ascii=False, indent=2)

# Leggi CSV
with (base / "people.csv").open("w", newline="", encoding="utf-8")
    as f:
    w = csv.writer(f)
    w.writerow(["name", "age"])
    w.writerow(["Ada", 27])

with (base / "people.csv").open("r", newline="", encoding="utf-8")
    as f:
```

```
19      r = csv.DictReader(f)
20      rows = list(r)
21      print(rows)
```

Spiegazione del codice

- **pathlib**: 'Path' compone percorsi portabili e leggibili. - **json.dump**: scrive JSON con `ensure_ascii=False` per Unicode e `indent` per leggibilità. - **csv.DictReader**: crea dizionari per riga, facilitando accesso per chiave. - **mkdir(exist_ok)**: idempotente; prepara cartella dati senza errori se già esiste.

Idiomi: pathlib, context manager, validazione

- **pathlib**: preferisci 'Path' per comporre, controllare e creare percorsi. - **context manager**: usa 'with' per file, socket e risorse; riduce boilerplate e bug. - **validazione**: verifica esistenza/permessi prima di elaborare; isola l'I/O dalla logica di trasformazione.

Pratiche Validare dimensione file, esistenza e permessi prima di elaborare. Isolare l'I/O dai trasformazioni per semplificare test e riuso. Consulta: https://docs.python.org/3/tutorial/inputoutput.html.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘:‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 7

Gestione Errori

7.1 Introduzione

Gestire eccezioni con `try/except`, `finally` e sollevare errori custom.

La descrizione enfatizza robustezza e chiarezza: quando intercettare e quando propagare, come loggare errori in modo utile e come definire gerarchie di eccezioni significative. Mostriamo pattern come "fail fast" e cleanup con `finally`.

Una buona gestione degli errori migliora affidabilità di script, servizi web, CLI e integrazioni con file e rete.

7.2 Obiettivi di Apprendimento

- Intercettare eccezioni comuni (ValueError, FileNotFoundError).
- Usare `try/except/else/finally`.
- Definire eccezioni personalizzate.

7.3 Concetti Fondamentali

Eccezioni

Esempio: `raise ValueError("messaggio")`

Best practice

Usa messaggi informativi, intercetta eccezioni specifiche, evita `except:` generici. Logga l'errore dove ha senso e decidi se propagare o tradurre in un fallback chiaro; pulisci risorse con `finally`.

7.4 Esempi Pratici

7.4.1 Try/Except

```
try:
    x = int("12a")
except ValueError:
    print("Input non valido")
```

Spiegazione del codice

- **Eccezione specifica**: ‘ValueError‘ per conversione fallita; evita ‘except:‘ generici. - **Messaggio**: fornisce feedback chiaro; separare validazione da logica di parsing. - **Propagazione**: se l’errore non è atteso, lasciare propagare.

7.4.2 Finally

```
f = None
try:
    f = open("file.txt")
    # ... usa f
finally:
    if f:
        f.close()
```

Spiegazione del codice

- **Cleanup garantito**: ‘finally‘ esegue sempre; evita leak di risorse. - **Preferenza**: usare ‘with open(...)‘ al posto del ‘try/finally‘ manuale. - **Stato**: inizializzare ‘f=None‘ evita ‘UnboundLocalError‘ se l’apertura fallisce.

7.4.3 Custom Exception

```
class NegativeError(Exception):
    pass

def sqrt(x):
    if x < 0:
        raise NegativeError("Valore negativo")
    return x ** 0.5
```

Spiegazione del codice

- **Gerarchia**: eredita da ‘Exception‘; personalizza il significato di dominio. - **Contratto**: precondizione ‘x >= 0‘; documentarla nel docstring e nelle annotazioni. - **Segnalazione**: usare messaggi informativi per diagnosi e UX.

Casi d’uso

- Validazione input e regole di dominio con eccezioni chiare.
- Gestione errori di rete/file con retry e fallback.
- Traduzione di errori tecnici in messaggi per utenti/CLI.

7.5 Esercizi

1. Scrivi una funzione che gestisce input numerici con retry.
2. Gestisci errori di rete simulati con `try/except`.
3. Definisci un’eccezione custom e usala in una validazione.

4. Usa `else` per eseguire codice solo senza eccezioni.
5. Registra errori in un file di log.

7.6 Riepilogo

Hai imparato a gestire e sollevare eccezioni correttamente.

7.7 Contesto e Applicazioni

Contesto e Applicazioni

- Resilienza di script (retry, gestione eccezioni prevedibili). - Validazione input e messaggi d'errore chiari. - Gestione risorse con context manager. - Strategie di fallback e logging di anomalie.

7.8 Approfondimenti

Spiegazioni dettagliate

- **try/except/else/finally**: `else` esegue se non ci sono eccezioni; `finally` sempre per cleanup.
- **Eccezioni personalizzate**: definire gerarchie chiare; documentare quando e perché vengono sollevate.
- **Raising e reraising**: `raise` per propagare; conservare il traceback con `raise` senza argomenti nel blocco `except`.
- **EAFP vs LBYL**: preferire "it's easier to ask forgiveness than permission" con gestione puntuale delle eccezioni.
- **Logging**: usare il modulo `logging` per diagnosi e telemetria, evitare `print` in codice di libreria.

Per chi viene da C/Java/JS: modello eccezioni, retry e logging

Eccezioni: Python usa eccezioni per condizioni anomale; evita codici di ritorno speciali. **Specificità**: cattura eccezioni specifiche (‘ValueError‘, ‘FileNotFoundError‘) e lascia propagare quelle inattese. **Cleanup**: preferisci context manager (‘with‘) rispetto a ‘try/finally‘ manuale per risorse. **Retry/backoff**: implementa retry con ‘time.sleep‘ o librerie; non nascondere errori irreparabili.
Esempi commentati:

```
import logging, time
log = logging.getLogger(__name__)

def parse_int(s):
    try:
        return int(s)
    except ValueError:
        log.warning("input non numerico: %s", s)
        return None

```

```python
def fetch_with_retry(fn, attempts=3, delay=0.5):
    for i in range(attempts):
        try:
            return fn()
        except TimeoutError:
            log.info("timeout, tentativo %s", i + 1)
            time.sleep(delay)
    raise RuntimeError("fallimento dopo retry")
```

Spiegazione del codice

- **Logging**: usa `getLogger(__name__)`; preferisci livelli (‘warning‘, ‘info‘) a ‘print‘. - **Retry**: intercetta eccezioni attese (‘TimeoutError‘) e applica backoff con ‘sleep‘. - **Fallback**: dopo tentativi, solleva errore aggregato chiaro; non nascondere failure. - **Parse**: ritorna ‘None‘ su input non numerico; scegliere ‘Optional[int]‘ nel contratto.

Idiomi: EAFP, traduzione errori, context manager

- **EAFP**: prova e gestisci l’eccezione dove ha senso, invece di pre-controllare ovunque. - **Traduzione**: traduci eccezioni tecniche in errori di dominio (con messaggi utili). - **Context manager**: usa ‘with‘ per risorse e per garantire rollback/cleanup automatici.

Consigli Controllare granularità degli `except`, evitare catture generiche; assicurare messaggi di errore chiari e azionabili. Consulta: https://docs.python.org/3/tutorial/errors.html.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 8

Moduli e Package

8.1 Introduzione

Organizzare codice in moduli e package, import e percorso di ricerca.

Questa descrizione spiega come strutturare un progetto in unità coerenti, evitare conflitti di nomi e adottare convenzioni di import leggibili. Tocchiamo `__init__.py`, percorsi relativi/assoluti e il ruolo di `sys.path`.

Un buon design dei package rende il codice estensibile e manutenibile, facilitando test, riuso e distribuzione.

8.2 Obiettivi di Apprendimento

- Creare e importare moduli.
- Strutturare package con `__init__.py`.
- Comprendere `sys.path` e import relativi.

8.3 Concetti Fondamentali

Import

`from pkg.mod import func` e `import pkg.mod as m`

Strutturazione e convenzioni

- Usa import assoluti per chiarezza (evita ambiguità dei relativi profondi). - Mantieni `__init__` minimale: re-export selettivo e costanti. - Progetta API di package: cosa esporre dal livello radice e cosa no. - Evita side-effect all'import: inizializza risorse su richiesta. - Documenta dipendenze interne tra moduli per ridurre accoppiamento.

8.4 Esempi Pratici

8.4.1 Modulo semplice

```
# file: util.py
def say(msg):
    print(msg)

# file: app.py
```

```
import util
util.say("Ciao")
```

Spiegazione

- Il modulo `util.py` definisce una funzione e viene importato in `app.py`. - `import util` carica il modulo e ne usa il namespace: `util.say(...)`. - Preferire import espliciti (funzioni o nomi) quando la UX lo richiede: `from util import say`. - Evitare side-effect all'import: l'esecuzione dovrebbe avvenire in `main()` o funzioni dedicate.

8.4.2 Package

```
# pkg/__init__.py
# pkg/tools.py
def add(a, b):
    return a + b

# main.py
from pkg.tools import add
print(add(2, 3))
```

Spiegazione

- Un package è una cartella (`pkg/`) con moduli; `__init__.py` può essere vuoto o fare re-export. - `from pkg.tools import add` importa simboli dal modulo `tools` all'interno del package. - Mantenere `__init__.py` minimale e documentare l'API pubblica con `__all__` se necessario. - Preferire import assoluti per chiarezza; usare relativi solo all'interno del package.

8.4.3 sys.path

```
import sys
print(sys.path)
```

Spiegazione

- `sys.path` è la lista di percorsi in cui Python cerca moduli e package. - Modificare `sys.path` a runtime è possibile ma va limitato a script/CLI, non a librerie. - Per la distribuzione, usare package installabili invece di aggiungere path manualmente.

Casi d'uso

- Libreria interna di utilità condivisa tra progetti scollegati. - Organizzare un'app in sottopackage (`core`, `api`, `cli`). - Separare plugin/estensioni caricati dinamicamente. - Re-export dell'API pubblica dal package root per UX migliore.

8.5 Esercizi

1. Crea un package con due moduli e import incrociati.
2. Implementa un modulo `mathutil` con funzioni di base.

3. Usa alias negli import e confronta leggibilità.
4. Esplora `sys.path` e aggiungi un percorso temporaneo.
5. Organizza un progetto in package e sottopackage.

8.6 Riepilogo

Hai organizzato codice con moduli e package in modo pulito.

8.7 Contesto e Applicazioni

Contesto e Applicazioni

- Organizzare progetti medio-grandi in package coerenti. - Definire API pubblica del package con re-export. - Isolare plugin/estensioni con moduli dedicati. - Preparare distribuzione e riuso interno.

8.8 Approfondimenti

Spiegazioni dettagliate

- **Import assoluti vs relativi**: preferire import assoluti per chiarezza; relativi utili all'interno di pacchetti.
- `__init__.py`: controlla l'esposizione del pacchetto; usare `__all__` per definire API pubblica.
- **Struttura dei moduli**: separare responsabilità e evitare cicli di dipendenze; predisporre sottopacchetti coerenti.
- **Ambienti e dipendenze**: congelare versioni con file di requisiti; documentare vincoli e compatibilità.

Distribuzione Valutare packaging con `pyproject.toml` e strumenti moderni; integrare test e lint nel processo di rilascio. Consulta: https://docs.python.org/3/tutorial/modules.html.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 9

Programmazione a Oggetti (OOP)

9.1 Introduzione

Classi, oggetti, ereditarietà, incapsulamento e polimorfismo in Python.

La descrizione chiarisce quando introdurre classi rispetto a funzioni e strutture dati, come modellare entità e relazioni e come usare metodi speciali per integrare con l'ecosistema Python. Accenniamo anche a `dataclasses` come alternativa leggera.

Applicare correttamente OOP aiuta a organizzare sistemi complessi, isolare responsabilità e facilitare evoluzione e test.

9.2 Obiettivi di Apprendimento

- Definire classi e metodi.
- Usare ereditarietà e overriding.
- Applicare proprietà e metodi speciali.

9.3 Concetti Fondamentali

Metodi speciali

`__init__`, `__repr__`, `__str__`, `__len__`, ecc.

Principi di design OOP

- SRP: ogni classe deve avere una responsabilità chiara. - Composizione *over* ereditarietà per flessibilità. - LSP: le sottoclassi devono rispettare i contratti della base. - Incapsula stato, esponi metodi chiari e invarianti. - Usa `dataclasses` per oggetti-dato semplici.

9.4 Esempi Pratici

9.4.1 Classe base

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

Spiegazione

- `__init__` inizializza lo stato; `__repr__` fornisce una rappresentazione utile al debug. - Evitare di mescolare responsabilità: la classe modella un punto, non gestisce I/O. - Per oggetti-dato, considerare `dataclasses` per ridurre boilerplate.

9.4.2 Ereditarietà

```
class Veicolo:
    def muovi(self):
        print("Mi muovo")

class Auto(Veicolo):
    def muovi(self):
        print("Guido")
```

Spiegazione

- `Auto` eredita da `Veicolo` e fa override di `muovi`. - Applicare LSP: la sottoclasse deve rispettare l'interfaccia/contratto della base. - Preferire composizione quando l'ereditarietà non rappresenta una relazione "è un".

9.4.3 Proprietà

```
class User:
    def __init__(self, name):
        self._name = name
    @property
    def name(self):
        return self._name
```

Spiegazione

- `@property` espone un attributo calcolato/validato mantenendo un'interfaccia pulita. - L'underscore (`_name`) indica convenzione di attributo "interno"; non è enforced. - Aggiungere setter solo se necessario per invarianti; altrimenti mantenere immutabilità.

Casi d'uso

- Modellare entità di dominio (`Ordine`, `Prodotto`, `Utente`). - Adapter/Strategy per variare comportamento in runtime. - Value Object immutabili per sicurezza e testabilità. - Astrazioni per driver (DB, API) con implementazioni intercambiabili.

9.5 Esercizi

1. Implementa una classe `Rettangolo` con area e perimetro.
2. Crea gerarchia `Animale` → `Cane`/`Gatto` con override.
3. Usa `@property` per validare attributi.
4. Implementa un contenitore con metodi speciali di sequenza.
5. Aggiungi un metodo di classe per creare da stringa.

9.6 Riepilogo

Hai applicato i concetti chiave di OOP in Python.

9.7 Contesto e Applicazioni

Contesto e Applicazioni

- Modellazione di dominio (ordini, utenti, prodotti). - UI e componenti riutilizzabili con pattern OOP. - Estensioni e plugin tramite interfacce e astrazioni. - Testare comportamenti con polimorfismo/mocking.

9.8 Approfondimenti

Spiegazioni dettagliate

- **Composizione vs ereditarietà**: privilegiare composizione per flessibilità; usare ereditarietà quando c'è una relazione "è un".
- **Dataclass**: generano boilerplate per classi di dati; controllare immutabilità con `frozen=True`.
- **Proprietà e incapsulamento**: `@property` per interfacce pulite; validare invarianti.
- **Dunder methods**: implementare `__repr__`, `__eq__` e `__hash__` in base all'identità del dominio.
- **Type hints**: agevolano strumenti statici; mantenere annotazioni aggiornate e coerenti.

Progettazione Definire responsabilità piccole e coese, evitare classi "tuttofare". Integrare test di unità per gli invarianti principali. Consulta: https://docs.python.org/3/tutorial/classes.html.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘:‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 10

Decoratori, Iteratori e Generatori

10.1 Introduzione

Pattern avanzati per estendere comportamento di funzioni e gestire serie di dati.

Questa descrizione mostra come comporre comportamenti senza modificare il codice esistente (decoratori), come iterare su flussi in modo lazy (iteratori) e generare sequenze efficienti (generatori) riducendo uso di memoria.

Questi strumenti rendono il codice più espressivo e performante in logging, validazione, pipeline di dati e elaborazioni su grandi volumi.

10.2 Obiettivi di Apprendimento

- Scrivere decoratori semplici.
- Implementare iteratori personalizzati.
- Usare generatori con `yield` e comprensioni generator.

10.3 Concetti Fondamentali

Decoratori

`@decorator` avvolge funzioni per aggiungere comportamento.

Iterazione e valutazione lazy

- Gli iteratori consumano elementi uno alla volta, riducendo memoria. - I generatori con `yield` modellano pipeline e flussi. - Componi trasformazioni con generator expression e `itertools`. - Backpressure: produci solo quanto richiesto dal consumatore.

10.4 Esempi Pratici

10.4.1 Decoratore

```
def log_calls(fn):
    def wrapper(*a, **k):
        print("chiamata", fn.__name__)
        return fn(*a, **k)
    return wrapper

```

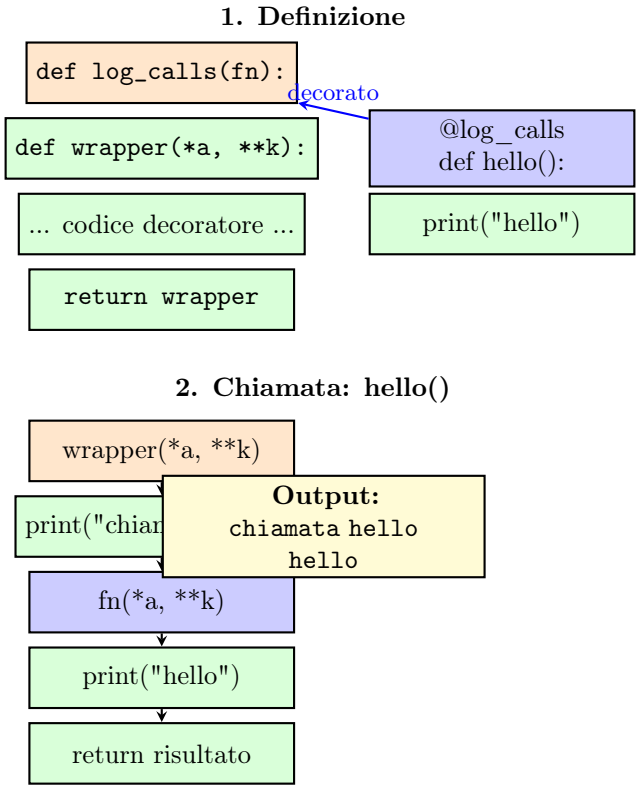
```
@log_calls
def hello():
    print("hello")
hello()
```

Spiegazione

- Il decoratore `log_calls` avvolge la funzione per eseguire codice prima/dopo. - Usare `functools.wraps` su `wrapper` per preservare metadati (nome, docstring). - I decoratori sono ottimi per logging, validazione, caching e autorizzazioni.

Flusso di esecuzione del decoratore

Ecco come funziona un decoratore visivamente:



Il decoratore **avvolge** la funzione originale:
wrapper → codice pre → funzione originale → codice post

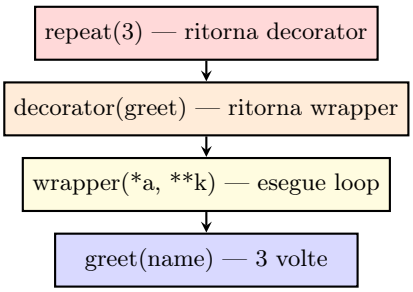
Decorator con parametri

```
def repeat(n):
    def decorator(fn):
        def wrapper(*a, **k):
            for _ in range(n):
                result = fn(*a, **k)
            return result
        return wrapper
    return decorator

```

```python
@repeat(3)
def greet(name):
    print(f"Ciao {name}!")

greet("Mario")  # Stampa 3 volte
```



Livello 1: Decoratori con parametri

Livello 2: Decoratore

Livello 3: Wrapper

Livello 4: Funzione originale

I decorator con parametri sono funzioni che ritornano decorator. La sintassi `@repeat(3)` chiama prima `repeat(3)`, che ritorna un decorator, che poi viene applicato a `greet`.

10.4.2 Iteratore

```python
class Count:
    def __init__(self, n):
        self.n = n; self.i = 0
    def __iter__(self): return self
    def __next__(self):
        if self.i >= self.n: raise StopIteration
        v = self.i; self.i += 1; return v
```

Spiegazione

- Implementa il protocollo iteratore con `__iter__` e `__next__`. - Solleva `StopIteration` per segnalare fine; consuma elemento per chiamata. - Preferire iteratori/generatori per flussi potenzialmente grandi o infiniti.

10.4.3 Generatore

```python
def gen(n):
    for i in range(n):
        yield i
print(list(gen(3)))
```

Spiegazione

- I generatori producono valori lazily con `yield`; consumati da `for` o `list(...)`. - Consentono pipeline efficienti e gestione di risorse con `try/finally`. - Comporre trasformazioni con generator expression e moduli come `itertools`.

Casi d'uso

- Logging/monitoring trasparente con decoratori. - Streaming di file/HTTP con trasformazioni lazy. - Validazione/autorità dei parametri senza cambiare funzioni originali. - Rate limiting e retry come decoratori riutilizzabili.

10.5 Esercizi

1. Crea un decoratore che misura il tempo di esecuzione.
2. Implementa un iteratore sui numeri pari fino a N.
3. Scrivi un generatore di Fibonacci.
4. Usa una generator expression per filtrare valori.
5. Implementa un decoratore che valida tipi di argomenti.

10.6 Riepilogo

Hai esplorato decoratori, iteratori e generatori in pratica.

10.7 Contesto e Applicazioni

Contesto e Applicazioni

- Strumentazione trasparente (logging, timing) con decoratori. - Pipeline dati lazy e streaming efficiente. - Validazioni, autorizzazioni e caching via decoratori. - Generator expression per filtri e trasformazioni.

10.8 Approfondimenti

Spiegazioni dettagliate

- **Protocollo iteratore**: implementare `__iter__` e `__next__` per oggetti iterabili.
- **Generatori**: usare `yield` per pipeline lazy e flussi di dati; gestire cleanup con `finally`.
- **Decoratori**: incapsulare cross-cutting concerns (logging, caching, autorizzazione); usare `functools.wraps`.
- **Caching**: `functools.lru_cache` per funzioni pure; definire dimensioni e politiche di invalidazione.

Pattern Combinare generatori e decoratori per pipeline efficienti e componibili; preferire iterazione lazy per dataset grandi. Consulta: https://docs.python.org/3/faq/programming.html#what-are-decorators.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 11

Standard Library Avanzata

Obiettivi del capitolo

Dopo questo capitolo saprai:

- Usare collections (Counter, defaultdict, deque, namedtuple)
- Applicare itertools per combinazioni e trasformazioni
- Ottimizzare con functools (lru_cache, partial, reduce)
- Manipolare date con datetime e timezone
- Gestire percorsi con pathlib
- Creare CLI con argparse
- Usare regex con il modulo re
- Serializzare dati con json e pickle
- Applicare type hints con typing

11.1 Teoria

La libreria standard Python offre moduli potenti per compiti comuni senza dipendenze esterne:

- **Evita reinventare la ruota**: soluzioni testate e ottimizzate
- **Portabilità**: funziona su tutte le piattaforme
- **Manutenibilità**: codice familiare per altri sviluppatori
- **Performance**: implementazioni in C per moduli critici

Filosofia: "Batteries included" — Python include tutto il necessario.

11.2 Collections

11.2.1 Counter

```
1 from collections import Counter
2
```

```
# Conta elementi
words = ["apple", "banana", "apple", "cherry", "banana", "apple"]
counts = Counter(words)
print(counts)  # Counter({'apple': 3, 'banana': 2, 'cherry': 1})

# Most common
print(counts.most_common(2))  # [('apple', 3), ('banana', 2)]

# Operazioni
text = "banana"
letter_counts = Counter(text)
print(letter_counts)  # Counter({'a': 3, 'n': 2, 'b': 1})

# Combine counters
c1 = Counter(a=3, b=1)
c2 = Counter(a=1, b=2)
print(c1 + c2)  # Counter({'a': 4, 'b': 3})
print(c1 - c2)  # Counter({'a': 2})
```

11.2.2 defaultdict

```
from collections import defaultdict

# Group by key
students = [
    ("Alice", "Math"),
    ("Bob", "Science"),
    ("Charlie", "Math"),
    ("David", "Science"),
]

# Con dict normale: controllare se key esiste
by_subject_manual = {}
for name, subject in students:
    if subject not in by_subject_manual:
        by_subject_manual[subject] = []
    by_subject_manual[subject].append(name)

# Con defaultdict: automatico!
by_subject = defaultdict(list)
for name, subject in students:
    by_subject[subject].append(name)

print(by_subject)
# defaultdict(<class 'list'>, {'Math': ['Alice', 'Charlie'],
#                              'Science': ['Bob', 'David']})

# Default values
counts = defaultdict(int)  # Default: 0
counts["apple"] += 1
counts["banana"] += 2
print(counts)  # defaultdict(<class 'int'>, {'apple': 1, 'banana': 2})
```

11.2.3 deque (Double-ended Queue)

```python
from collections import deque

# Efficient queue operations
queue = deque([1, 2, 3])

# Add/remove from both ends (O(1))
queue.append(4)         # Right: [1, 2, 3, 4]
queue.appendleft(0)     # Left: [0, 1, 2, 3, 4]
queue.pop()             # Remove right: [0, 1, 2, 3]
queue.popleft()         # Remove left: [1, 2, 3]

# Rotate
queue.rotate(1)   # [3, 1, 2]
queue.rotate(-1)  # [1, 2, 3]

# Max length (circular buffer)
buffer = deque(maxlen=3)
buffer.extend([1, 2, 3, 4, 5])  # Only keeps last 3
print(buffer)  # deque([3, 4, 5], maxlen=3)
```

11.2.4 namedtuple

```python
from collections import namedtuple

# Define structure
Point = namedtuple("Point", ["x", "y"])

# Create instances
p1 = Point(10, 20)
p2 = Point(x=5, y=15)

# Access by name (readable!)
print(p1.x, p1.y)  # 10 20

# Still a tuple (immutable, hashable)
print(p1[0], p1[1])  # 10 20
x, y = p1  # Unpacking works

# Use case: function returns
def get_coordinates():
    return Point(100, 200)

coord = get_coordinates()
print(f"X: {coord.x}, Y: {coord.y}")
```

11.3 Itertools

11.3.1 Combinazioni e Permutazioni

```python
import itertools

# Combinations (order doesn't matter)
items = ["A", "B", "C"]
print(list(itertools.combinations(items, 2)))
```

```
# [('A', 'B'), ('A', 'C'), ('B', 'C')]

# Permutations (order matters)
print(list(itertools.permutations(items, 2)))
# [('A', 'B'), ('A', 'C'), ('B', 'A'), ('B', 'C'), ('C', 'A'), ('C', 'B
    ')]

# Product (Cartesian product)
colors = ["red", "blue"]
sizes = ["S", "M", "L"]
print(list(itertools.product(colors, sizes)))
# [('red', 'S'), ('red', 'M'), ('red', 'L'),
#  ('blue', 'S'), ('blue', 'M'), ('blue', 'L')]
```

11.3.2 Chain, Groupby, Islice

```
import itertools

# Chain: concatenate iterables
list1 = [1, 2, 3]
list2 = [4, 5, 6]
print(list(itertools.chain(list1, list2)))  # [1, 2, 3, 4, 5, 6]

# Groupby: group consecutive items
data = [
    ("Alice", "Math"),
    ("Bob", "Math"),
    ("Charlie", "Science"),
    ("David", "Science"),
]

for subject, group in itertools.groupby(data, key=lambda x: x[1]):
    students = [name for name, _ in group]
    print(f"{subject}: {students}")
# Math: ['Alice', 'Bob']
# Science: ['Charlie', 'David']

# Islice: slice iterator (memory efficient)
numbers = range(1000000)  # Doesn't create list!
first_10 = list(itertools.islice(numbers, 10))
print(first_10)  # [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Every nth element
every_5th = list(itertools.islice(range(50), 0, None, 5))
print(every_5th)  # [0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
```

11.3.3 Cycle, Repeat, Accumulate

```
import itertools

# Cycle: infinite repetition
colors = itertools.cycle(["red", "green", "blue"])
for i, color in enumerate(colors):
    print(color)
    if i >= 5:
        break
```

```
# red, green, blue, red, green, blue

# Repeat: repeat element
for x in itertools.repeat("Hello", 3):
    print(x)  # Hello (3 times)

# Accumulate: running totals
numbers = [1, 2, 3, 4, 5]
print(list(itertools.accumulate(numbers)))  # [1, 3, 6, 10, 15]

# Accumulate with custom function
import operator
print(list(itertools.accumulate(numbers, operator.mul)))  # [1, 2, 6,
    24, 120]
```

11.4 Functools

11.4.1 lru_cache (Memoization)

```
from functools import lru_cache

# Without cache: slow (exponential time)
def fibonacci_slow(n):
    if n < 2:
        return n
    return fibonacci_slow(n-1) + fibonacci_slow(n-2)

# With cache: fast! (linear time)
@lru_cache(maxsize=128)
def fibonacci_fast(n):
    if n < 2:
        return n
    return fibonacci_fast(n-1) + fibonacci_fast(n-2)

print(fibonacci_fast(100))  # Instant!

# Check cache stats
print(fibonacci_fast.cache_info())
# CacheInfo(hits=98, misses=101, maxsize=128, currsize=101)

# Clear cache
fibonacci_fast.cache_clear()
```

11.4.2 partial (Partial Application)

```
from functools import partial

# Create specialized functions
def power(base, exponent):
    return base ** exponent

square = partial(power, exponent=2)
cube = partial(power, exponent=3)

print(square(5))  # 25
```

```
print(cube(5))    # 125

# Use case: callback with fixed arguments
import tkinter as tk

def on_button_click(button_id, event):
    print(f"Button {button_id} clicked")

root = tk.Tk()
for i in range(3):
    callback = partial(on_button_click, i)
    tk.Button(root, text=f"Button {i}", command=lambda e=None: callback(
        e)).pack()
```

11.4.3 reduce

```
from functools import reduce
import operator

# Reduce: apply function cumulatively
numbers = [1, 2, 3, 4, 5]

# Sum
total = reduce(operator.add, numbers, 0)
print(total)  # 15

# Product
product = reduce(operator.mul, numbers, 1)
print(product)  # 120

# Find max
max_val = reduce(lambda a, b: a if a > b else b, numbers)
print(max_val)  # 5

# Custom: concatenate strings
words = ["Hello", "World", "!"]
sentence = reduce(lambda a, b: a + " " + b, words)
print(sentence)  # "Hello World !"
```

11.5 Datetime

11.5.1 Basics

```
from datetime import datetime, timedelta, date, time

# Current datetime
now = datetime.now()
print(now)  # 2024-01-15 14:30:45.123456

# Create specific datetime
dt = datetime(2024, 12, 25, 18, 30, 0)
print(dt)  # 2024-12-25 18:30:00

# Date and time separately
today = date.today()
```

```
current_time = datetime.now().time()
print(today)          # 2024-01-15
print(current_time)   # 14:30:45.123456

# Formatting (strftime)
formatted = now.strftime("%Y-%m-%d %H:%M:%S")
print(formatted)  # "2024-01-15 14:30:45"

# Parsing (strptime)
date_string = "2024-01-15"
parsed = datetime.strptime(date_string, "%Y-%m-%d")
print(parsed)  # 2024-01-15 00:00:00
```

11.5.2 Timedelta (Durations)

```
from datetime import datetime, timedelta

now = datetime.now()

# Add/subtract time
tomorrow = now + timedelta(days=1)
next_week = now + timedelta(weeks=1)
two_hours_ago = now - timedelta(hours=2)

print(tomorrow)
print(next_week)

# Difference between dates
birthday = datetime(2024, 6, 15)
today = datetime.now()
days_until = (birthday - today).days
print(f"Days until birthday: {days_until}")

# Duration components
duration = timedelta(days=7, hours=3, minutes=30)
print(duration.total_seconds())  # 612600.0
```

11.6 Pathlib

11.6.1 Path Operations

```
from pathlib import Path

# Current directory
cwd = Path.cwd()
print(cwd)

# Home directory
home = Path.home()
print(home)

# Join paths (portable!)
config_file = home / ".config" / "app" / "settings.json"
print(config_file)

```

```
# Path components
path = Path("/home/user/documents/file.txt")
print(path.name)        # file.txt
print(path.stem)        # file
print(path.suffix)      # .txt
print(path.parent)      # /home/user/documents
print(path.parts)       # ('/', 'home', 'user', 'documents', 'file.txt')

# Check existence
if path.exists():
    print("File exists")

if path.is_file():
    print("It's a file")

if path.is_dir():
    print("It's a directory")
```

11.6.2 File I/O and Globbing

```
from pathlib import Path

# Read/write text
config = Path("config.txt")
config.write_text("Setting=Value\n")
content = config.read_text()
print(content)

# Read/write bytes
data_file = Path("data.bin")
data_file.write_bytes(b"\x00\xFF\x42")
data = data_file.read_bytes()

# Iterate directory
for item in Path(".").iterdir():
    if item.is_file():
        print(f"File: {item.name}")

# Glob patterns
for tex_file in Path(".").glob("*.tex"):
    print(tex_file)

# Recursive glob
for py_file in Path(".").rglob("*.py"):  # All .py in subdirs
    print(py_file)

# Create directories
new_dir = Path("output/data/processed")
new_dir.mkdir(parents=True, exist_ok=True)
```

11.7 Argparse (CLI Arguments)

```
import argparse
from pathlib import Path

```

```python
def main():
    parser = argparse.ArgumentParser(
        description="Process some data files"
    )

    # Positional argument
    parser.add_argument("input", type=Path, help="Input file path")

    # Optional arguments
    parser.add_argument(
        "-o", "--output",
        type=Path,
        default=Path("output.txt"),
        help="Output file path (default: output.txt)"
    )

    parser.add_argument(
        "-v", "--verbose",
        action="store_true",
        help="Enable verbose output"
    )

    parser.add_argument(
        "-n", "--count",
        type=int,
        default=10,
        help="Number of items to process"
    )

    # Choices
    parser.add_argument(
        "--format",
        choices=["json", "csv", "xml"],
        default="json",
        help="Output format"
    )

    args = parser.parse_args()

    # Use arguments
    if args.verbose:
        print(f"Processing {args.input}")
        print(f"Output: {args.output}")
        print(f"Format: {args.format}")
        print(f"Count: {args.count}")

    # Process file
    if args.input.exists():
        content = args.input.read_text()
        # ... process content ...
        args.output.write_text(f"Processed {args.count} items")

if __name__ == "__main__":
    main()

# Usage examples:
# python script.py input.txt
# python script.py input.txt -o result.txt --verbose
```

```
# python script.py input.txt --format csv -n 20
```

11.8 Regular Expressions (re)

```
import re

# Match pattern
text = "My email is john@example.com"
match = re.search(r"\w+@\w+\.\w+", text)
if match:
    print(match.group())  # john@example.com

# Find all
text = "Emails: alice@test.com, bob@demo.org"
emails = re.findall(r"\w+@\w+\.\w+", text)
print(emails)  # ['alice@test.com', 'bob@demo.org']

# Groups (capture parts)
pattern = r"(\w+)@(\w+)\.(\w+)"
match = re.search(pattern, "user@example.com")
if match:
    print(match.group(0))  # user@example.com (full match)
    print(match.group(1))  # user
    print(match.group(2))  # example
    print(match.group(3))  # com

# Named groups
pattern = r"(?P<user>\w+)@(?P<domain>\w+\.\w+)"
match = re.search(pattern, "alice@example.com")
if match:
    print(match.group("user"))    # alice
    print(match.group("domain"))  # example.com

# Substitute
text = "Phone: 123-456-7890"
cleaned = re.sub(r"[\s-]", "", text)
print(cleaned)  # Phone:1234567890

# Split
text = "apple,banana;cherry|date"
fruits = re.split(r"[,;|]", text)
print(fruits)  # ['apple', 'banana', 'cherry', 'date']

# Compile for reuse
email_pattern = re.compile(r"\w+@\w+\.\w+")
if email_pattern.search("Contact: test@mail.com"):
    print("Email found")
```

11.9 JSON e Pickle

11.9.1 JSON

```
import json
from pathlib import Path
```

```
3
4  # Python dict to JSON
5  data = {
6      "name": "Alice",
7      "age": 30,
8      "skills": ["Python", "SQL", "Docker"],
9      "active": True
10 }
11
12 # To JSON string
13 json_string = json.dumps(data, indent=2)
14 print(json_string)
15
16 # To file
17 Path("data.json").write_text(json.dumps(data, indent=2))
18
19 # From JSON string
20 parsed = json.loads(json_string)
21 print(parsed["name"])
22
23 # From file
24 loaded = json.loads(Path("data.json").read_text())
25 print(loaded)
26
27 # Custom objects (use default function)
28 from datetime import datetime
29
30 def serialize_datetime(obj):
31     if isinstance(obj, datetime):
32         return obj.isoformat()
33     raise TypeError(f"Type {type(obj)} not serializable")
34
35 data_with_date = {
36     "event": "Meeting",
37     "timestamp": datetime.now()
38 }
39
40 json_str = json.dumps(data_with_date, default=serialize_datetime)
41 print(json_str)
```

11.9.2 Pickle (Python Binary)

```
1  import pickle
2  from pathlib import Path
3
4  # Any Python object
5  data = {
6      "name": "Bob",
7      "skills": ["Python", "ML"],
8      "scores": [95, 87, 92]
9  }
10
11 # Save to file
12 with open("data.pkl", "wb") as f:
13     pickle.dump(data, f)
14
15 # Load from file
```

```python
with open("data.pkl", "rb") as f:
    loaded = pickle.load(f)

print(loaded)

# WARNING: Pickle is Python-specific and can execute code!
# Only unpickle trusted data
# For data exchange, prefer JSON
```

11.10 Type Hints (typing)

```python
from typing import List, Dict, Optional, Union, Tuple, Callable

# Basic types
def greet(name: str) -> str:
    return f"Hello, {name}"

# Lists and dicts
def process_scores(scores: List[int]) -> Dict[str, float]:
    return {
        "average": sum(scores) / len(scores),
        "max": max(scores)
    }

# Optional (can be None)
def find_user(user_id: int) -> Optional[Dict[str, str]]:
    # ... search logic ...
    if user_id == 1:
        return {"name": "Alice", "email": "alice@test.com"}
    return None

# Union (multiple types)
def format_id(id_value: Union[int, str]) -> str:
    return str(id_value).zfill(5)

# Tuple (fixed size)
def get_coordinates() -> Tuple[float, float]:
    return (10.5, 20.3)

# Callable (function type)
def apply_operation(
    numbers: List[int],
    operation: Callable[[int], int]
) -> List[int]:
    return [operation(n) for n in numbers]

result = apply_operation([1, 2, 3], lambda x: x * 2)
print(result)  # [2, 4, 6]

# Type aliases
Vector = List[float]
Matrix = List[Vector]

def multiply_matrix(m: Matrix) -> Matrix:
    # ... matrix multiplication ...
    pass
```

11.11 Progetto: Log Analyzer

```python
from pathlib import Path
from collections import Counter, defaultdict
from datetime import datetime
import re
import argparse
import json

class LogAnalyzer:
    def __init__(self, log_file: Path):
        self.log_file = log_file
        self.entries = []

    def parse_log(self):
        """Parse log entries"""
        # Pattern: 2024-01-15 14:30:45 [INFO] Message here
        pattern = r"(\d{4}-\d{2}-\d{2} \d{2}:\d{2}:\d{2}) \[(\w+)\] (.+)"

        for line in self.log_file.read_text().splitlines():
            match = re.match(pattern, line)
            if match:
                timestamp_str, level, message = match.groups()
                timestamp = datetime.strptime(
                    timestamp_str, "%Y-%m-%d %H:%M:%S"
                )
                self.entries.append({
                    "timestamp": timestamp,
                    "level": level,
                    "message": message
                })

    def analyze(self) -> dict:
        """Analyze log data"""
        # Count by level
        level_counts = Counter(e["level"] for e in self.entries)

        # Group by hour
        by_hour = defaultdict(int)
        for entry in self.entries:
            hour = entry["timestamp"].hour
            by_hour[hour] += 1

        # Find errors
        errors = [e for e in self.entries if e["level"] == "ERROR"]

        return {
            "total_entries": len(self.entries),
            "by_level": dict(level_counts),
            "by_hour": dict(by_hour),
            "error_count": len(errors),
            "errors": errors[:10]  # First 10 errors
        }

    def save_report(self, output: Path):
        """Save analysis report as JSON"""
        analysis = self.analyze()
```

```
56
57          # Serialize datetime objects
58          def serialize(obj):
59              if isinstance(obj, datetime):
60                  return obj.isoformat()
61              raise TypeError(f"Type {type(obj)} not serializable")
62
63          output.write_text(
64              json.dumps(analysis, indent=2, default=serialize)
65          )
66
67  def main():
68      parser = argparse.ArgumentParser(description="Analyze log files")
69      parser.add_argument("logfile", type=Path, help="Log file to analyze"
        )
70      parser.add_argument(
71          "-o", "--output",
72          type=Path,
73          default=Path("report.json"),
74          help="Output report file"
75      )
76
77      args = parser.parse_args()
78
79      if not args.logfile.exists():
80          print(f"Error: {args.logfile} not found")
81          return
82
83      analyzer = LogAnalyzer(args.logfile)
84      analyzer.parse_log()
85      analyzer.save_report(args.output)
86
87      print(f"Analysis complete. Report saved to {args.output}")
88
89  if __name__ == "__main__":
90      main()
```

Best Practices

- Preferire pathlib a os.path per operazioni su percorsi
- Usare type hints per documentare contratti di funzioni
- Compilare regex con re.compile() se riusate spesso
- defaultdict evita check "key in dict" ridondanti
- lru_cache per funzioni pure computazionalmente costose
- JSON per interoperabilità, pickle solo per Python
- argparse per CLI consistenti con –help automatico
- namedtuple per strutture dati immutabili leggibili
- itertools per pipeline di dati memory-efficient
- Timezone-aware datetime per app internazionali

Errori Comuni

- Usare os.path invece di pathlib (meno leggibile)
- Dimenticare maxsize in lru_cache (memory leak!)
- Pickle di dati non fidati (security risk!)
- Regex senza raw string (r"") — escape issues
- groupby senza ordinamento preventivo (funziona su consecutivi!)
- Naive datetime (senza timezone) per app globali
- JSON di datetime senza serializer custom
- Ignorare .cache_clear() per test con lru_cache
- Creare Path con stringhe hardcoded (non portabile!)
- reduce senza valore iniziale (crash su lista vuota)

11.12 Esercizi

11.12.1 Livello Base

1. Usa Counter per trovare le 5 parole più frequenti in un testo
2. Crea CLI con argparse che accetta file input/output
3. Usa pathlib per trovare tutti i file .txt in una directory

11.12.2 Livello Intermedio

1. Implementa cache LRU manuale con defaultdict + deque
2. Parser CSV con namedtuple per righe
3. Regex per validare email e estrarre dominio

11.12.3 Livello Avanzato

1. Log aggregator: analizza log da più file, report JSON
2. Data pipeline con itertools (filter, map, groupby)
3. CLI tool con argparse, pathlib, JSON per config file manager

11.13 Riferimenti

- Python Standard Library: https://docs.python.org/3/library/
- Collections: https://docs.python.org/3/library/collections.html
- Itertools recipes: https://docs.python.org/3/library/itertools.html#itertools-recipes
- Pathlib guide: https://realpython.com/python-pathlib/

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 12

GUI con Tkinter

Obiettivi del capitolo

Dopo questo capitolo saprai:

- Creare finestre e dialog con Tkinter
- Usare widgets base (Label, Button, Entry, Text)
- Gestire layout con pack, grid, place
- Implementare eventi e callbacks
- Usare widgets avanzati (Combobox, Scale, Progressbar)
- Creare menu e dialogs
- Usare Canvas per grafica
- Threading per operazioni lunghe
- Applicare pattern MVC per GUI

12.1 Teoria

Tkinter è la libreria GUI standard di Python (inclusa). Basata su Tk/Tcl, è:

- **Cross-platform**: Windows, macOS, Linux
- **Lightweight**: Nessuna dipendenza esterna
- **Event-driven**: Main loop gestisce eventi
- **Semplice**: Ideale per tool, prototipi, utility

Limitazioni:

- Look nativo solo con ttk (themed widgets)
- Meno widgets/features vs PyQt/wxPython
- Performance limitata con molti widgets

12.2 Finestra Base

```python
import tkinter as tk

# Crea finestra principale
root = tk.Tk()
root.title("My Application")
root.geometry("400x300")  # Width x Height

# Aggiungi widget
label = tk.Label(root, text="Hello, Tkinter!")
label.pack()

# Start event loop
root.mainloop()
```

12.3 Widgets Base

12.3.1 Label

```python
import tkinter as tk

root = tk.Tk()

# Label con testo
label1 = tk.Label(root, text="Simple Label")
label1.pack()

# Label con styling
label2 = tk.Label(
    root,
    text="Styled Label",
    font=("Arial", 16, "bold"),
    fg="blue",
    bg="yellow"
)
label2.pack()

# Label con variabile dinamica
text_var = tk.StringVar()
text_var.set("Dynamic Text")

label3 = tk.Label(root, textvariable=text_var)
label3.pack()

# Update text
text_var.set("Updated Text!")

root.mainloop()
```

12.3.2 Button

```python
import tkinter as tk

def on_click():
```

```
    print("Button clicked!")
    label.config(text="Button was clicked")

root = tk.Tk()

label = tk.Label(root, text="Click the button")
label.pack()

button = tk.Button(
    root,
    text="Click Me",
    command=on_click,
    bg="green",
    fg="white",
    font=("Arial", 12)
)
button.pack()

root.mainloop()
```

12.3.3 Entry (Text Input)

```
import tkinter as tk

def submit():
    text = entry.get()
    print(f"Input: {text}")
    result_label.config(text=f"You entered: {text}")
    entry.delete(0, tk.END)  # Clear

root = tk.Tk()

entry = tk.Entry(root, width=30)
entry.pack()

button = tk.Button(root, text="Submit", command=submit)
button.pack()

result_label = tk.Label(root, text="")
result_label.pack()

root.mainloop()
```

12.3.4 Text (Multi-line)

```
import tkinter as tk

root = tk.Tk()

text = tk.Text(root, height=10, width=40)
text.pack()

# Insert text
text.insert(tk.END, "Line 1\n")
text.insert(tk.END, "Line 2\n")

```

```
# Get text
def get_text():
    content = text.get("1.0", tk.END)  # From start to end
    print(content)

tk.Button(root, text="Get Text", command=get_text).pack()

root.mainloop()
```

12.4 Layout Managers

12.4.1 Pack

```
import tkinter as tk

root = tk.Tk()

# Stack vertically (default)
tk.Label(root, text="Top").pack()
tk.Label(root, text="Middle").pack()
tk.Label(root, text="Bottom").pack()

# Side by side
tk.Label(root, text="Left").pack(side=tk.LEFT)
tk.Label(root, text="Right").pack(side=tk.RIGHT)

# With padding
tk.Label(root, text="Padded").pack(padx=20, pady=10)

root.mainloop()
```

12.4.2 Grid

```
import tkinter as tk

root = tk.Tk()

# Grid layout (row, column)
tk.Label(root, text="Name:").grid(row=0, column=0, sticky=tk.W)
tk.Entry(root).grid(row=0, column=1)

tk.Label(root, text="Email:").grid(row=1, column=0, sticky=tk.W)
tk.Entry(root).grid(row=1, column=1)

tk.Label(root, text="Password:").grid(row=2, column=0, sticky=tk.W)
tk.Entry(root, show="*").grid(row=2, column=1)

tk.Button(root, text="Submit").grid(row=3, column=0, columnspan=2)

root.mainloop()
```

12.5 Eventi e Bindings

```python
import tkinter as tk

def on_key(event):
    print(f"Key pressed: {event.char}")

def on_mouse_click(event):
    print(f"Mouse clicked at ({event.x}, {event.y})")

root = tk.Tk()

# Bind keyboard
root.bind("<Key>", on_key)
root.bind("<Return>", lambda e: print("Enter pressed"))

# Bind mouse
canvas = tk.Canvas(root, width=300, height=200, bg="white")
canvas.pack()
canvas.bind("<Button-1>", on_mouse_click)  # Left click

root.mainloop()
```

12.6 Widgets Avanzati (ttk)

```python
import tkinter as tk
from tkinter import ttk

root = tk.Tk()

# Combobox
combo = ttk.Combobox(root, values=["Python", "Java", "C++"])
combo.set("Python")
combo.pack()

# Progressbar
progress = ttk.Progressbar(root, length=200, mode="determinate")
progress["maximum"] = 100
progress["value"] = 50
progress.pack()

# Spinbox
spinbox = tk.Spinbox(root, from_=0, to=100, increment=5)
spinbox.pack()

# Scale (Slider)
scale = tk.Scale(root, from_=0, to=100, orient=tk.HORIZONTAL)
scale.pack()

def show_value():
    print(f"Scale value: {scale.get()}")

tk.Button(root, text="Show Value", command=show_value).pack()

root.mainloop()
```

12.7 Menu

```
import tkinter as tk
from tkinter import messagebox

def new_file():
    messagebox.showinfo("New", "New file created")

def open_file():
    messagebox.showinfo("Open", "Open file dialog")

def exit_app():
    root.quit()

root = tk.Tk()

# Create menubar
menubar = tk.Menu(root)
root.config(menu=menubar)

# File menu
file_menu = tk.Menu(menubar, tearoff=0)
menubar.add_cascade(label="File", menu=file_menu)
file_menu.add_command(label="New", command=new_file)
file_menu.add_command(label="Open", command=open_file)
file_menu.add_separator()
file_menu.add_command(label="Exit", command=exit_app)

# Edit menu
edit_menu = tk.Menu(menubar, tearoff=0)
menubar.add_cascade(label="Edit", menu=edit_menu)
edit_menu.add_command(label="Cut", command=lambda: print("Cut"))
edit_menu.add_command(label="Copy", command=lambda: print("Copy"))

root.mainloop()
```

12.8 Dialogs

```
import tkinter as tk
from tkinter import messagebox, filedialog, simpledialog

root = tk.Tk()

def show_info():
    messagebox.showinfo("Info", "This is info")

def show_warning():
    messagebox.showwarning("Warning", "This is warning")

def show_error():
    messagebox.showerror("Error", "This is error")

def ask_question():
    result = messagebox.askyesno("Question", "Do you like Python?")
    print(f"Answer: {result}")

```

```
def open_file():
    filename = filedialog.askopenfilename(
        title="Select file",
        filetypes=[("Text files", "*.txt"), ("All files", "*.*")]
    )
    print(f"Selected: {filename}")

def ask_name():
    name = simpledialog.askstring("Input", "Enter your name:")
    print(f"Name: {name}")

tk.Button(root, text="Info", command=show_info).pack()
tk.Button(root, text="Warning", command=show_warning).pack()
tk.Button(root, text="Error", command=show_error).pack()
tk.Button(root, text="Question", command=ask_question).pack()
tk.Button(root, text="Open File", command=open_file).pack()
tk.Button(root, text="Ask Name", command=ask_name).pack()

root.mainloop()
```

12.9 Canvas: Grafica

```
import tkinter as tk

root = tk.Tk()

canvas = tk.Canvas(root, width=400, height=300, bg="white")
canvas.pack()

# Draw shapes
canvas.create_line(0, 0, 400, 300, fill="red", width=2)
canvas.create_rectangle(50, 50, 150, 100, fill="blue", outline="black")
canvas.create_oval(200, 50, 300, 150, fill="green")
canvas.create_polygon(100, 200, 150, 250, 50, 250, fill="yellow")

# Draw text
canvas.create_text(200, 270, text="Hello Canvas!", font=("Arial", 16))

root.mainloop()
```

12.10 Threading per Operazioni Lunghe

```
import tkinter as tk
import threading
import time

def long_task():
    """Simula operazione lunga"""
    button.config(state=tk.DISABLED, text="Working...")

    def worker():
        time.sleep(3)  # Simula lavoro
        # Update UI dal main thread
        root.after(0, task_completed)
```

```
13
14      thread = threading.Thread(target=worker)
15      thread.start()
16
17  def task_completed():
18      button.config(state=tk.NORMAL, text="Start Task")
19      label.config(text="Task completed!")
20
21  root = tk.Tk()
22
23  label = tk.Label(root, text="Ready")
24  label.pack()
25
26  button = tk.Button(root, text="Start Task", command=long_task)
27  button.pack()
28
29  root.mainloop()
```

12.11 Progetto: Calculator

```
1   import tkinter as tk
2
3   class Calculator:
4       def __init__(self, root):
5           self.root = root
6           self.root.title("Calculator")
7
8           self.expression = ""
9
10          # Display
11          self.display = tk.Entry(root, font=("Arial", 20), justify=tk.
                RIGHT)
12          self.display.grid(row=0, column=0, columnspan=4, sticky="ew",
                padx=5, pady=5)
13
14          # Buttons
15          buttons = [
16              ('7', 1, 0), ('8', 1, 1), ('9', 1, 2), ('/', 1, 3),
17              ('4', 2, 0), ('5', 2, 1), ('6', 2, 2), ('*', 2, 3),
18              ('1', 3, 0), ('2', 3, 1), ('3', 3, 2), ('-', 3, 3),
19              ('0', 4, 0), ('.', 4, 1), ('+', 4, 2), ('=', 4, 3),
20          ]
21
22          for (text, row, col) in buttons:
23              button = tk.Button(
24                  root,
25                  text=text,
26                  font=("Arial", 16),
27                  command=lambda t=text: self.on_button_click(t)
28              )
29              button.grid(row=row, column=col, sticky="nsew", padx=2, pady
                    =2)
30
31          # Clear button
32          tk.Button(
33              root,
```

```
            text="C",
            font=("Arial", 16),
            command=self.clear
        ).grid(row=4, column=3, sticky="nsew", padx=2, pady=2)

        # Configure grid weights
        for i in range(5):
            root.grid_rowconfigure(i, weight=1)
        for i in range(4):
            root.grid_columnconfigure(i, weight=1)

    def on_button_click(self, char):
        if char == '=':
            try:
                result = eval(self.expression)
                self.display.delete(0, tk.END)
                self.display.insert(0, str(result))
                self.expression = str(result)
            except:
                self.display.delete(0, tk.END)
                self.display.insert(0, "Error")
                self.expression = ""
        else:
            self.expression += char
            self.display.delete(0, tk.END)
            self.display.insert(0, self.expression)

    def clear(self):
        self.expression = ""
        self.display.delete(0, tk.END)

root = tk.Tk()
app = Calculator(root)
root.mainloop()
```

Best Practices

- Separa logica business da UI (pattern MVC/MVP)
- Non bloccare main thread (usa threading per I/O)
- Usa ttk widgets per look nativo moderno
- Valida input prima di processare
- Gestisci eccezioni in callbacks
- Usa StringVar/IntVar per dati dinamici
- Organizza codice in classi per app complesse
- Usa grid per form, pack per semplici layout
- Non mescolare pack/grid nello stesso container
- Fai cleanup su close (close files, threads)

Errori Comuni

- Bloccare main thread con operazioni lunghe
- Mescolare pack e grid nello stesso parent
- Non gestire eccezioni in callbacks
- Usare eval() senza validazione (security!)
- Dimenticare mainloop()
- Creare troppe istanze Tk() (una basta)
- Non configurare grid weights (resize issues)
- Non usare ttk (look old-school)
- Update UI da thread secondari (crash!)
- Memory leak: non cleanup widgets rimossi

12.12 Esercizi

12.12.1 Livello Base

1. Crea form login (username, password, button)
2. Implementa converter temperatura (Celsius Fahrenheit)
3. Todo list (add, display, clear)

12.12.2 Livello Intermedio

1. Text editor con menu (New, Open, Save, Exit)
2. Calculator completa con history
3. Image viewer con file dialog

12.12.3 Livello Avanzato

1. Drawing app con Canvas (brush, colors, clear)
2. Database browser (SQLite) con CRUD operations
3. File manager con tree view e operations

12.13 Riferimenti

- Tkinter Documentation: https://docs.python.org/3/library/tk.html
- Tkinter Tutorial: https://realpython.com/python-gui-tkinter/
- ttk widgets: https://docs.python.org/3/library/tkinter.ttk.html

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 13

Web Development con Flask

13.1 Introduzione

Creare applicazioni web leggere con Flask: routing, template e richieste.

La descrizione introduce il ciclo richiesta-risposta, la definizione di rotte e l'uso dei template per generare HTML dinamico. Discutiamo buone pratiche di struttura (`create_app`, blueprint) e cenni a gestione errori e form.

Flask è adatto a API e piccoli siti, microservizi e prototipi veloci con una curva di apprendimento contenuta.

13.2 Obiettivi di Apprendimento

- Definire route e gestire metodi HTTP.
- Usare template Jinja2.
- Gestire form e parametri.

13.3 Concetti Fondamentali

Struttura di app e ciclo richiesta

- App factory (`create_app`) e blueprint per modularità. - Context di applicazione e richiesta, gestione `g`/`session`. - Routing e metodi HTTP, validazione input e sicurezza. - Template Jinja2: separazione presentazione/logica.

13.4 Esempi Pratici

13.4.1 App base

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def home():
    return "Hello Flask"

# avvia: flask --app app run
```

Spiegazione

- `Flask(__name__)` crea l'app; `@app.route` definisce una route. - Separare creazione app in factory (`create_app`) per testing/configurazioni. - Avviare con CLI `flask`; gestire segreti/config via variabili d'ambiente.

13.4.2 Parametri

```
from flask import request
@app.route("/echo")
def echo():
    msg = request.args.get("msg", "")
    return f"{msg}"
```

Spiegazione

- I parametri query si leggono da `request.args`; validare e sanificare input. - Usare metodi HTTP corretti (GET per lettura, POST per invio dati). - Gestire errori e codici di risposta appropriati (400/404/500).

13.4.3 Template

```
from flask import render_template
@app.route("/page")
def page():
    return render_template("page.html", title="Demo")
```

Spiegazione

- `render_template` usa Jinja2: separa presentazione dalla logica. - Passare variabili al template e evitare concatenazioni manuali di HTML. - Proteggere da XSS/CSRF; usare form e token sicuri.

Casi d'uso

- Micro API per servizi interni. - Piccoli siti con template dinamici. - Dashboard leggere con form e validazioni. - Prototipi rapidi di microservizi.

13.5 Esercizi

1. Crea una route `/hello/<name>`.
2. Gestisci un form POST e visualizza i dati.
3. Servi un template con una lista iterata.
4. Aggiungi gestione errori 404 custom.
5. Separa app in `create_app()` e blueprint.

13.6 Riepilogo

Hai definito una piccola app Flask con rotte e template.

13.7 Contesto e Applicazioni

Contesto e Applicazioni

- Microservizi e API interne. - Siti dinamici con template. - Dashboard leggere con form e validazioni. - Prototipi rapidi integrati con DB/servizi.

13.8 Approfondimenti

Spiegazioni dettagliate

- **App factory**: creare l'app con funzioni per configurazioni diverse (test, prod).
- **Blueprints**: modularizzare viste e API; isolare dipendenze.
- **Contesto di richiesta**: gestire `request`/`g`/`session` in modo sicuro; evitare globali.
- **Template Jinja2**: separare presentazione; validare input e proteggere da XSS/CSRF.
- **Testing**: usare il test client; verificare percorsi e stati; integrare CI.

Deployment Differenziare configurazione sviluppo/produzione, log, e gestione errori; preferire server WSGI adeguati e reverse proxy. Consulta: https://flask.palletsprojects.com/.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 14

Automazione e Web Scraping

Obiettivi del capitolo

Dopo questo capitolo saprai:

- Automatizzare task con subprocess e schedule
- Effettuare richieste HTTP con requests
- Parsare HTML con BeautifulSoup
- Consumare API REST
- Scrapare siti dinamici con Selenium
- Rispettare etica e robots.txt
- Implementare rate limiting e retry logic
- Salvare dati (CSV, JSON, database)
- Schedulare script automatici

14.1 Automazione con subprocess

14.1.1 Eseguire comandi sistema

```python
import subprocess

# Run command e cattura output
result = subprocess.run(
    ["ls", "-la"],
    capture_output=True,
    text=True
)

print(result.stdout)
print(f"Exit code: {result.returncode}")

# Con check (raise exception se fallisce)
try:
    subprocess.run(["git", "status"], check=True)
except subprocess.CalledProcessError as e:
    print(f"Command failed with exit code {e.returncode}")
```

```
18
19 # Con input
20 result = subprocess.run(
21     ["grep", "python"],
22     input="python is great\njava is good\n",
23     capture_output=True,
24     text=True
25 )
26 print(result.stdout)  # python is great
```

14.1.2 Automazione file system

```
1  import os
2  import shutil
3  from pathlib import Path
4
5  # Backup files
6  def backup_directory(source, dest):
7      """Copia directory con backup"""
8      if Path(dest).exists():
9          # Backup esistente
10         backup_path = f"{dest}.backup"
11         shutil.move(dest, backup_path)
12
13     shutil.copytree(source, dest)
14     print(f"Backed up {source} to {dest}")
15
16 # Cleanup old files
17 def cleanup_old_files(directory, days=30):
18     """Rimuovi file più vecchi di N giorni"""
19     import time
20
21     now = time.time()
22     cutoff = now - (days * 86400)
23
24     for file in Path(directory).glob("*"):
25         if file.is_file() and file.stat().st_mtime < cutoff:
26             print(f"Removing old file: {file}")
27             file.unlink()
28
29 # Batch rename
30 def batch_rename(directory, pattern, replacement):
31     """Rename files con pattern"""
32     for file in Path(directory).glob(pattern):
33         new_name = file.name.replace(pattern.replace("*", ""),
               replacement)
34         file.rename(file.parent / new_name)
35         print(f"Renamed: {file.name} -> {new_name}")
```

14.2 Web Scraping con requests

14.2.1 Installazione

```
1 pip install requests beautifulsoup4 lxml
```

14.2.2 GET requests

```python
import requests

# GET request semplice
response = requests.get("https://api.github.com/users/python")

# Check status
if response.status_code == 200:
    data = response.json()
    print(f"Name: {data['name']}")
    print(f"Repos: {data['public_repos']}")
else:
    print(f"Error: {response.status_code}")

# Con parameters
params = {"q": "python", "sort": "stars"}
response = requests.get("https://api.github.com/search/repositories",
    params=params)

# Con headers
headers = {"User-Agent": "Mozilla/5.0"}
response = requests.get("https://example.com", headers=headers)

# Con timeout
try:
    response = requests.get("https://example.com", timeout=5)
except requests.Timeout:
    print("Request timed out")
```

14.2.3 POST requests

```python
# POST con JSON
data = {"username": "alice", "email": "alice@example.com"}
response = requests.post(
    "https://api.example.com/users",
    json=data,
    headers={"Content-Type": "application/json"}
)

# POST con form data
form_data = {"username": "alice", "password": "secret"}
response = requests.post("https://example.com/login", data=form_data)

# POST con file upload
files = {"file": open("document.pdf", "rb")}
response = requests.post("https://example.com/upload", files=files)
```

14.2.4 Sessions e cookies

```python
# Session mantiene cookies tra requests
session = requests.Session()

# Login
session.post("https://example.com/login", data={"user": "alice", "pass":
    "secret"})
```

```
6
7  # Richieste successive usano cookies automaticamente
8  response = session.get("https://example.com/dashboard")
9
10 # Session con headers default
11 session.headers.update({"User-Agent": "My Bot 1.0"})
12
13 response = session.get("https://example.com/api")
```

14.3 BeautifulSoup: Parsing HTML

14.3.1 Basics

```
1  from bs4 import BeautifulSoup
2  import requests
3
4  url = "https://example.com"
5  response = requests.get(url)
6  soup = BeautifulSoup(response.content, "lxml")
7
8  # Find elementi
9  title = soup.find("title")
10 print(title.text)
11
12 # Find all
13 links = soup.find_all("a")
14 for link in links:
15     print(link.get("href"))
16
17 # CSS selectors
18 articles = soup.select(".article")
19 first_paragraph = soup.select_one("p")
20
21 # Navigazione
22 div = soup.find("div", class_="container")
23 h2 = div.find("h2")
24 paragraph = h2.find_next_sibling("p")
```

14.3.2 Esempio: Scraping news

```
1  import requests
2  from bs4 import BeautifulSoup
3
4  def scrape_news(url):
5      """Scrape news articles"""
6      response = requests.get(url, headers={"User-Agent": "Mozilla/5.0"})
7      soup = BeautifulSoup(response.content, "lxml")
8
9      articles = []
10
11     for article in soup.select(".article"):
12         title = article.select_one("h2").text.strip()
13         link = article.select_one("a")["href"]
14         summary = article.select_one(".summary").text.strip()
15
```

```
        articles.append({
            "title": title,
            "link": link,
            "summary": summary
        })

    return articles

# Uso
news = scrape_news("https://news.example.com")
for article in news:
    print(f"{article['title']}: {article['link']}")
```

14.4 API REST

14.4.1 Consumo API GitHub

```
import requests

class GitHubAPI:
    BASE_URL = "https://api.github.com"

    def __init__(self, token=None):
        self.session = requests.Session()
        if token:
            self.session.headers["Authorization"] = f"token {token}"

    def get_user(self, username):
        response = self.session.get(f"{self.BASE_URL}/users/{username}")
        response.raise_for_status()
        return response.json()

    def get_repos(self, username):
        response = self.session.get(
            f"{self.BASE_URL}/users/{username}/repos",
            params={"sort": "updated", "per_page": 100}
        )
        response.raise_for_status()
        return response.json()

# Uso
api = GitHubAPI()
user = api.get_user("python")
print(f"Python organization has {user['public_repos']} public repos")

repos = api.get_repos("python")
for repo in repos[:5]:
    print(f"- {repo['name']}: {repo['stargazers_count']} stars")
```

14.5 Selenium: Scraping dinamico

14.5.1 Setup

```
pip install selenium webdriver-manager

```

```
# Download ChromeDriver automaticamente con webdriver-manager
```

14.5.2 Basics

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from webdriver_manager.chrome import ChromeDriverManager

# Setup driver
driver = webdriver.Chrome(ChromeDriverManager().install())

# Naviga
driver.get("https://example.com")

# Trova elementi
title = driver.find_element(By.TAG_NAME, "h1")
print(title.text)

# Click
button = driver.find_element(By.ID, "submit-button")
button.click()

# Input text
input_field = driver.find_element(By.NAME, "username")
input_field.send_keys("alice")

# Wait per elemento
element = WebDriverWait(driver, 10).until(
    EC.presence_of_element_located((By.CLASS_NAME, "result"))
)

# Screenshot
driver.save_screenshot("screenshot.png")

# Close
driver.quit()
```

14.6 Etica e Best Practices

14.6.1 robots.txt

```
import requests
from urllib.robotparser import RobotFileParser

def can_fetch(url, user_agent="*"):
    """Check se posso scrapare URL"""
    robots_url = f"{url.rstrip('/')}/robots.txt"

    rp = RobotFileParser()
    rp.set_url(robots_url)
    rp.read()

    return rp.can_fetch(user_agent, url)
```

```

# Uso
if can_fetch("https://example.com/page"):
    # OK to scrape
    response = requests.get("https://example.com/page")
else:
    print("Not allowed by robots.txt")
```

14.6.2 Rate Limiting

```
import time
import requests

class RateLimitedScraper:
    def __init__(self, delay=1.0):
        self.delay = delay
        self.last_request = 0

    def get(self, url):
        # Wait se troppo veloce
        elapsed = time.time() - self.last_request
        if elapsed < self.delay:
            time.sleep(self.delay - elapsed)

        self.last_request = time.time()
        return requests.get(url)

# Uso: max 1 request/second
scraper = RateLimitedScraper(delay=1.0)

for url in urls:
    response = scraper.get(url)
    # Process...
```

14.6.3 Retry con backoff

```
import requests
from requests.adapters import HTTPAdapter
from requests.packages.urllib3.util.retry import Retry

def requests_retry_session(
    retries=3,
    backoff_factor=0.3,
    status_forcelist=(500, 502, 504),
):
    session = requests.Session()

    retry = Retry(
        total=retries,
        read=retries,
        connect=retries,
        backoff_factor=backoff_factor,
        status_forcelist=status_forcelist,
    )

    adapter = HTTPAdapter(max_retries=retry)
```

```
    session.mount("http://", adapter)
    session.mount("https://", adapter)

    return session

# Uso
session = requests_retry_session()
response = session.get("https://example.com")
# Automatically retries on connection errors
```

14.7 Salvataggio Dati

14.7.1 CSV

```
import csv

# Scrape e salva CSV
articles = scrape_news("https://news.example.com")

with open("news.csv", "w", newline="", encoding="utf-8") as f:
    writer = csv.DictWriter(f, fieldnames=["title", "link", "summary"])
    writer.writeheader()
    writer.writerows(articles)
```

14.7.2 JSON

```
import json

# Salva JSON
with open("news.json", "w", encoding="utf-8") as f:
    json.dump(articles, f, indent=2, ensure_ascii=False)

# Append JSON (JSONLines format)
with open("news.jsonl", "a", encoding="utf-8") as f:
    for article in articles:
        f.write(json.dumps(article, ensure_ascii=False) + "\n")
```

14.7.3 Database

```
import sqlite3

# Save to SQLite
conn = sqlite3.connect("news.db")
cursor = conn.cursor()

cursor.execute("""
    CREATE TABLE IF NOT EXISTS articles (
        id INTEGER PRIMARY KEY,
        title TEXT,
        link TEXT UNIQUE,
        summary TEXT,
        scraped_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
""")
```

```
16
17 for article in articles:
18     cursor.execute(
19         "INSERT OR IGNORE INTO articles (title, link, summary) VALUES
               (?, ?, ?)",
20         (article["title"], article["link"], article["summary"])
21     )
22
23 conn.commit()
24 conn.close()
```

14.8 Scheduling

14.8.1 Con schedule

```
1 pip install schedule
```

```
1  import schedule
2  import time
3
4  def job():
5      print("Running scraping job...")
6      articles = scrape_news("https://news.example.com")
7      save_to_database(articles)
8      print(f"Scraped {len(articles)} articles")
9
10 # Schedule
11 schedule.every().hour.do(job)                    # Ogni ora
12 schedule.every().day.at("10:30").do(job)        # Ogni giorno alle 10:30
13 schedule.every().monday.at("09:00").do(job)     # Ogni lunedì alle 9:00
14
15 # Run
16 while True:
17     schedule.run_pending()
18     time.sleep(60)  # Check every minute
```

14.8.2 Con cron (Linux/Mac)

```
1  # Modifica crontab
2  crontab -e
3
4  # Run script ogni ora
5  0 * * * * /usr/bin/python3 /path/to/scraper.py
6
7  # Run ogni giorno alle 2 AM
8  0 2 * * * /usr/bin/python3 /path/to/scraper.py
9
10 # Run ogni lunedì alle 9 AM
11 0 9 * * 1 /usr/bin/python3 /path/to/scraper.py
12
13 # Formato: minute hour day month weekday command
```

14.9 Progetto Completo: News Aggregator

```python
import requests
from bs4 import BeautifulSoup
import sqlite3
import schedule
import time
from datetime import datetime

class NewsAggregator:
    def __init__(self, db_path="news.db"):
        self.db_path = db_path
        self.setup_database()

    def setup_database(self):
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()
        cursor.execute("""
            CREATE TABLE IF NOT EXISTS articles (
                id INTEGER PRIMARY KEY,
                source TEXT,
                title TEXT,
                link TEXT UNIQUE,
                summary TEXT,
                scraped_at TIMESTAMP
            )
        """)
        conn.commit()
        conn.close()

    def scrape_source(self, name, url, selector):
        """Scrape single source"""
        try:
            response = requests.get(url, headers={"User-Agent": "Mozilla
                /5.0"}, timeout=10)
            soup = BeautifulSoup(response.content, "lxml")

            articles = []
            for article in soup.select(selector):
                title = article.select_one("h2").text.strip()
                link = article.select_one("a")["href"]
                summary = article.select_one(".summary").text.strip()

                articles.append({
                    "source": name,
                    "title": title,
                    "link": link,
                    "summary": summary,
                    "scraped_at": datetime.now()
                })

            return articles

        except Exception as e:
            print(f"Error scraping {name}: {e}")
            return []

    def save_articles(self, articles):
```

```
        """Save to database"""
        conn = sqlite3.connect(self.db_path)
        cursor = conn.cursor()

        saved = 0
        for article in articles:
            try:
                cursor.execute(
                    "INSERT INTO articles (source, title, link, summary,
                        scraped_at) VALUES (?, ?, ?, ?, ?)",
                    (article["source"], article["title"], article["link"
                       ], article["summary"], article["scraped_at"])
                )
                saved += 1
            except sqlite3.IntegrityError:
                pass  # Duplicate

        conn.commit()
        conn.close()
        return saved

    def run(self):
        """Run scraping job"""
        sources = [
            ("TechNews", "https://technews.example.com", ".article"),
            ("ScienceDaily", "https://sciencedaily.example.com", ".news-
               item"),
        ]

        all_articles = []
        for name, url, selector in sources:
            print(f"Scraping {name}...")
            articles = self.scrape_source(name, url, selector)
            all_articles.extend(articles)
            time.sleep(1)  # Rate limiting

        saved = self.save_articles(all_articles)
        print(f"Saved {saved} new articles out of {len(all_articles)}")

# Usage
aggregator = NewsAggregator()

# Run once
aggregator.run()

# Schedule
schedule.every().hour.do(aggregator.run)

while True:
    schedule.run_pending()
    time.sleep(60)
```

Best Practices

- Sempre rispetta robots.txt e Terms of Service
- Usa User-Agent header descrittivo (evita ban)

- Implementa rate limiting (1-2 requests/second max)
- Usa retry con exponential backoff
- Gestisci timeout (max 10-30 secondi)
- Salva dati incrementalmente (no perdi tutto se crash)
- Log errori per debugging
- Usa selectors CSS robusti (class, id, data attributes)
- Valida dati prima di salvare
- Non scrapare durante picchi di traffico

Errori Comuni

- Ignorare robots.txt (rischio ban)
- Troppi requests troppo velocemente (ban IP)
- Selectors fragili (class="btn-primary-2024-v3")
- Non gestire errori di rete
- Non usare timeout (script bloccati)
- Hardcodare selectors (cambiano spesso)
- Non validare dati estratti
- Non usare sessions (perdere cookies/auth)
- Scrapare dati personali senza consenso (GDPR!)
- Non gestire encoding (caratteri corrotti)

14.10 Esercizi

14.10.1 Livello Base

1. Scrivi script che scarica homepage di 5 siti e salva HTML
2. Scrape titoli articoli da news website e salva CSV
3. Automatizza backup directory con subprocess

14.10.2 Livello Intermedio

1. Implementa rate-limited scraper con retry logic
2. Consuma GitHub API e salva repo più starred in database
3. Schedule script che scrape ogni ora con schedule
4. Parse HTML table e converti in CSV

14.10.3 Livello Avanzato

1. Implementa news aggregator completo (3+ sources, DB, scheduling)
2. Scrape sito con login usando Selenium
3. Crea API wrapper per servizio esterno con caching
4. Implementa distributed scraping con multiprocessing

14.11 Riferimenti

- requests: https://requests.readthedocs.io/
- BeautifulSoup: https://www.crummy.com/software/BeautifulSoup/bs4/doc/
- Selenium: https://selenium-python.readthedocs.io/
- schedule: https://schedule.readthedocs.io/

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘:‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 15

Testing e Debugging

Obiettivi del capitolo

Dopo questo capitolo saprai:

- Comprendere l'importanza del testing e tipi di test
- Scrivere test con unittest e pytest
- Usare fixtures, parametrize e markers
- Implementare mocking con unittest.mock
- Misurare code coverage
- Applicare Test-Driven Development (TDD)
- Debuggare con pdb, breakpoint() e IDE
- Usare logging strategico
- Profilare codice per performance
- Integrare testing in CI/CD

15.1 Introduzione

15.1.1 Perché testare?

I test sono fondamentali per:

- **Qualità**: Catturare bug prima della produzione
- **Refactoring sicuro**: Modificare codice con confidenza
- **Documentazione**: Test come esempi d'uso
- **Design**: Test influenzano architettura (testabile = modulare)
- **Regressioni**: Prevenire re-introduzione di bug

15.1.2 Piramide dei test

```
    /\
   /E2E\        End-to-End (pochi, lenti, fragili)
```

```
     /------\
    /  Inte  \     Integration (moderati)
   /----------\
  /    Unit    \   Unit (molti, veloci, stabili)
 /--------------\
```

Unit Tests (70%):

- Testano singola funzione/classe isolatamente
- Veloci (ms), stabili, facili da debuggare
- Mock dependencies esterne

Integration Tests (20%):

- Testano interazione tra componenti
- Database, API, file system reali
- Più lenti, più setup richiesto

End-to-End Tests (10%):

- Testano applicazione completa (UI, API, DB)
- Molto lenti, fragili (flaky)
- Solo per flussi critici

15.2 unittest: Testing Standard Library

15.2.1 Struttura base

```
import unittest

def add(a, b):
    return a + b

def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")
    return a / b

class TestMathFunctions(unittest.TestCase):
    def test_add_positive_numbers(self):
        result = add(2, 3)
        self.assertEqual(result, 5)

    def test_add_negative_numbers(self):
        result = add(-2, -3)
        self.assertEqual(result, -5)

    def test_divide_normal(self):
        result = divide(10, 2)
        self.assertEqual(result, 5.0)

    def test_divide_by_zero_raises_error(self):
```

```
        with self.assertRaises(ValueError):
            divide(10, 0)

if __name__ == "__main__":
    unittest.main()
```

15.2.2 Assertions comuni

```
import unittest

class TestAssertions(unittest.TestCase):
    def test_equality(self):
        self.assertEqual(1 + 1, 2)
        self.assertNotEqual(1 + 1, 3)

    def test_boolean(self):
        self.assertTrue(True)
        self.assertFalse(False)

    def test_none(self):
        value = None
        self.assertIsNone(value)
        self.assertIsNotNone("hello")

    def test_membership(self):
        self.assertIn(3, [1, 2, 3])
        self.assertNotIn(4, [1, 2, 3])

    def test_types(self):
        self.assertIsInstance("hello", str)
        self.assertNotIsInstance("hello", int)

    def test_exceptions(self):
        with self.assertRaises(ZeroDivisionError):
            1 / 0

        with self.assertRaises(ValueError) as context:
            int("not a number")
        self.assertIn("invalid literal", str(context.exception))

    def test_approximate_equality(self):
        # Float comparison con tolleranza
        self.assertAlmostEqual(0.1 + 0.2, 0.3, places=7)
```

15.2.3 setUp e tearDown

```
import unittest

class DatabaseTest(unittest.TestCase):
    def setUp(self):
        """Eseguito PRIMA di ogni test"""
        print("Setup: Creating database connection")
        self.db_connection = connect_to_database()
        self.user = create_test_user()

    def tearDown(self):
```

```
        """Eseguito DOPO ogni test"""
        print("Teardown: Cleaning up")
        delete_test_user(self.user)
        self.db_connection.close()

    def test_user_creation(self):
        # self.db_connection e self.user disponibili qui
        self.assertIsNotNone(self.user.id)

    def test_user_update(self):
        # setUp viene eseguito di nuovo, fresh user
        self.user.name = "New Name"
        self.user.save()
        self.assertEqual(self.user.name, "New Name")

class TestWithClassSetup(unittest.TestCase):
    @classmethod
    def setUpClass(cls):
        """Eseguito UNA VOLTA all'inizio della classe"""
        print("Setup class: Heavy initialization")
        cls.expensive_resource = load_expensive_resource()

    @classmethod
    def tearDownClass(cls):
        """Eseguito UNA VOLTA alla fine della classe"""
        print("Teardown class: Release resources")
        cls.expensive_resource.cleanup()

    def test_one(self):
        self.assertIsNotNone(self.expensive_resource)

    def test_two(self):
        # Stessa expensive_resource condivisa
        self.assertIsNotNone(self.expensive_resource)
```

15.3 pytest: Testing Moderno

15.3.1 Installazione e primi test

```
pip install pytest

# Run tests
pytest                    # Tutti i test
pytest test_math.py       # File specifico
pytest -v                 # Verbose
pytest -k "test_add"      # Solo test con "add" nel nome
pytest -x                 # Stop alla prima failure
pytest --maxfail=2        # Stop dopo 2 failures
```

```
# test_math.py
def add(a, b):
    return a + b

def test_add_positive():
    assert add(2, 3) == 5

```

```python
def test_add_negative():
    assert add(-2, -3) == -5

def test_add_zero():
    assert add(5, 0) == 5

# pytest trova automaticamente tutti i file test_*.py
# e tutte le funzioni test_*()
```

15.3.2 Fixtures

```python
import pytest

@pytest.fixture
def sample_user():
    """Fixture: crea user per test"""
    user = {"id": 1, "name": "Alice", "email": "alice@example.com"}
    return user

@pytest.fixture
def database_connection():
    """Fixture con setup e teardown"""
    print("\nSetup: Connecting to database")
    conn = connect_to_database()

    yield conn  # Il test riceve conn

    print("\nTeardown: Closing database")
    conn.close()

def test_user_name(sample_user):
    # pytest inietta automaticamente sample_user
    assert sample_user["name"] == "Alice"

def test_database_query(database_connection):
    # database_connection automaticamente injected
    result = database_connection.execute("SELECT 1")
    assert result == 1

@pytest.fixture(scope="module")
def expensive_resource():
    """Fixture condivisa per tutto il modulo"""
    resource = load_expensive_resource()
    yield resource
    resource.cleanup()

# scope options: function (default), class, module, session
```

15.3.3 Parametrize: Test Data-Driven

```python
import pytest

@pytest.mark.parametrize("a,b,expected", [
    (2, 3, 5),
    (-2, -3, -5),
    (0, 0, 0),
```

```
    (100, 200, 300),
])
def test_add_parametrized(a, b, expected):
    assert add(a, b) == expected

@pytest.mark.parametrize("dividend,divisor,expected", [
    (10, 2, 5.0),
    (9, 3, 3.0),
    (7, 2, 3.5),
])
def test_divide_parametrized(dividend, divisor, expected):
    assert divide(dividend, divisor) == expected

@pytest.mark.parametrize("invalid_input", [
    "not a number",
    "",
    "12.34.56",
    None,
])
def test_int_conversion_fails(invalid_input):
    with pytest.raises(ValueError):
        int(invalid_input)
```

15.3.4 Markers: Organizzare test

```
import pytest

@pytest.mark.slow
def test_slow_operation():
    # Test che richiede molto tempo
    time.sleep(5)
    assert True

@pytest.mark.database
def test_database_query():
    # Test che richiede database
    result = db.query("SELECT 1")
    assert result == 1

@pytest.mark.skip(reason="Feature not implemented yet")
def test_future_feature():
    assert False

@pytest.mark.skipif(sys.platform == "win32", reason="Unix only")
def test_unix_specific():
    assert True

@pytest.mark.xfail(reason="Known bug #123")
def test_known_failing():
    assert 1 + 1 == 3  # Aspettato failure

# Run specific markers
# pytest -m slow              # Solo slow tests
# pytest -m "not slow"        # Escludi slow tests
# pytest -m "database and not slow"
```

15.4 Mocking

15.4.1 unittest.mock

```python
from unittest.mock import Mock, patch, MagicMock
import requests

def get_user_data(user_id):
    """Funzione che chiama API esterna"""
    response = requests.get(f"https://api.example.com/users/{user_id}")
    return response.json()

# Test SENZA mock (chiama API reale - MALE!)
# def test_get_user_data():
#     data = get_user_data(1)  # Chiama API reale!
#     assert data["name"] == "Alice"

# Test CON mock (simula API - BENE!)
@patch("requests.get")
def test_get_user_data_mocked(mock_get):
    # Configura mock response
    mock_response = Mock()
    mock_response.json.return_value = {"id": 1, "name": "Alice"}
    mock_get.return_value = mock_response

    # Chiama funzione (usa mock invece di requests.get reale)
    data = get_user_data(1)

    # Verifica
    assert data["name"] == "Alice"
    mock_get.assert_called_once_with("https://api.example.com/users/1")
```

15.4.2 Mock con side_effect

```python
from unittest.mock import Mock

def test_mock_side_effect():
    # Simula eccezione
    mock = Mock(side_effect=ValueError("Invalid input"))

    with pytest.raises(ValueError):
        mock()

    # Simula valori multipli
    mock = Mock(side_effect=[1, 2, 3])
    assert mock() == 1
    assert mock() == 2
    assert mock() == 3

    # Simula funzione custom
    def custom_function(x):
        return x * 2

    mock = Mock(side_effect=custom_function)
    assert mock(5) == 10
```

15.4.3 pytest-mock

```
pip install pytest-mock
```

```
# Con pytest-mock fixture
def test_get_user_data(mocker):
    # mocker.patch invece di @patch decorator
    mock_get = mocker.patch("requests.get")

    mock_response = mocker.Mock()
    mock_response.json.return_value = {"id": 1, "name": "Alice"}
    mock_get.return_value = mock_response

    data = get_user_data(1)

    assert data["name"] == "Alice"
    mock_get.assert_called_once()
```

15.5 Code Coverage

```
pip install coverage pytest-cov

# Con coverage tool
coverage run -m pytest
coverage report          # Report testuale
coverage html           # Report HTML in htmlcov/

# Con pytest-cov
pytest --cov=mymodule --cov-report=html
pytest --cov=mymodule --cov-report=term-missing  # Mostra righe non
    coperte
```

Coverage Best Practices

- **Non puntare a 100%**: Focus su codice critico, non su getters/setters
- **Coverage ≠ Quality**: 100% coverage non garantisce qualità test
- **Identifica gap**: Usa coverage per trovare codice non testato
- **Concentrati su logica business**: Testa algoritmi, edge cases
- **Target ragionevole**: 70-80% è spesso sufficiente

15.6 Test-Driven Development (TDD)

15.6.1 Red-Green-Refactor Cycle

```
1. RED:     Scrivi test che fallisce
2. GREEN:   Scrivi codice minimo per passare test
3. REFACTOR: Migliora codice mantenendo test verdi
```

15.6.2 Esempio TDD: Funzione is_prime()

```
# Step 1: RED - Scrivi test che fallisce
def test_is_prime_2():
    assert is_prime(2) == True

# Step 2: GREEN - Implementa minimo
def is_prime(n):
    return True  # Passa test ma è sbagliato!

# Step 1 (again): RED - Aggiungi test che fallisce
def test_is_prime_4():
    assert is_prime(4) == False

# Step 2 (again): GREEN - Fix implementation
def is_prime(n):
    if n < 2:
        return False
    for i in range(2, n):
        if n % i == 0:
            return False
    return True

# Step 3: REFACTOR - Ottimizza
def is_prime(n):
    if n < 2:
        return False
    if n == 2:
        return True
    if n % 2 == 0:
        return False
    for i in range(3, int(n**0.5) + 1, 2):
        if n % i == 0:
            return False
    return True

# Test ancora verdi! Refactoring sicuro
```

15.7 Debugging con pdb

15.7.1 Breakpoint manuale

```
def buggy_function(numbers):
    total = 0
    for num in numbers:
        import pdb; pdb.set_trace()  # Breakpoint qui
        total += num
    return total

result = buggy_function([1, 2, 3])

# Comandi pdb:
# n (next): esegui prossima riga
# s (step): entra in funzione
# c (continue): continua fino a prossimo breakpoint
# l (list): mostra codice circostante
```

```
# p variable: stampa variabile
# pp variable: pretty print
# w (where): mostra stack trace
# q (quit): esci
```

15.7.2 breakpoint() - Python 3.7+

```
def calculate(a, b):
    result = a + b
    breakpoint()  # Equivalente a pdb.set_trace()
    return result * 2

# Disable all breakpoints senza modificare codice:
# PYTHONBREAKPOINT=0 python script.py
```

15.7.3 Post-mortem debugging

```
import pdb

def divide(a, b):
    return a / b

try:
    result = divide(10, 0)
except ZeroDivisionError:
    pdb.post_mortem()  # Debug al momento del crash
```

15.8 Logging Strategico

15.8.1 Configurazione logging

```
import logging

# Setup logging
logging.basicConfig(
    level=logging.DEBUG,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler("app.log"),
        logging.StreamHandler()  # Anche su console
    ]
)

logger = logging.getLogger(__name__)

# Livelli: DEBUG < INFO < WARNING < ERROR < CRITICAL

def process_data(data):
    logger.debug(f"Processing data: {data}")

    if not data:
        logger.warning("Empty data received")
        return

```

```
    try:
        result = expensive_operation(data)
        logger.info(f"Operation successful: {result}")
        return result
    except Exception as e:
        logger.error(f"Operation failed: {e}", exc_info=True)
        raise
```

15.8.2 Logging in produzione

```
import logging
from logging.handlers import RotatingFileHandler

# Rotating log files (max 10MB, keep 5 backups)
handler = RotatingFileHandler(
    "app.log",
    maxBytes=10*1024*1024,
    backupCount=5
)

formatter = logging.Formatter(
    '%(asctime)s - %(name)s - %(levelname)s - %(message)s'
)
handler.setFormatter(formatter)

logger = logging.getLogger(__name__)
logger.setLevel(logging.INFO)  # Prod: INFO, Dev: DEBUG
logger.addHandler(handler)
```

15.9 Profiling Performance

15.9.1 timeit: Misura tempo

```
import timeit

# Misura tempo di esecuzione
def slow_function():
    return sum(range(1000000))

execution_time = timeit.timeit(slow_function, number=100)
print(f"Time: {execution_time:.4f}s for 100 executions")

# One-liner
time = timeit.timeit("sum(range(1000000))", number=100)

# Confronta approcci
list_comp = timeit.timeit("[x*2 for x in range(1000)]", number=10000)
map_func = timeit.timeit("list(map(lambda x: x*2, range(1000)))", number
    =10000)

print(f"List comprehension: {list_comp:.4f}s")
print(f"Map function: {map_func:.4f}s")
```

15.9.2 cProfile: Profiling dettagliato

```
# Da command line
python -m cProfile -s cumtime script.py

# In codice
import cProfile
import pstats

profiler = cProfile.Profile()
profiler.enable()

# Codice da profilare
slow_function()

profiler.disable()

# Stampa stats
stats = pstats.Stats(profiler)
stats.sort_stats("cumulative")
stats.print_stats(10)  # Top 10 funzioni più lente
```

Best Practices Testing

- Scrivi test PRIMA di fixare bug (test regression)
- Test devono essere veloci ($<$ 1s per unit test)
- Test devono essere deterministici (no random, no time.sleep)
- Test devono essere isolati (no dipendenze tra test)
- Usa nomi descrittivi: `test_user_creation_with_invalid_email_fails`
- Mock dipendenze esterne (DB, API, file system)
- Organizza test in directory `tests/`
- Usa fixtures per setup comune
- Non testare implementation details, testa behavior
- CI deve runare test su ogni commit

Errori Comuni

- Test che dipendono da ordine esecuzione
- Test che modificano stato globale
- Over-mocking (mock tutto, testa nulla)
- Under-testing edge cases
- Test troppo lenti (no sleep, no I/O reale)
- Assert senza messaggi descrittivi
- Non testare error paths (solo happy path)

- Non usare fixtures (duplicazione setup)
- Test fragili (hardcoded values, tight coupling)
- Ignorare test flaky invece di fixarli

15.10 CI/CD Integration

15.10.1 GitHub Actions esempio

```yaml
# .github/workflows/test.yml
name: Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python
      uses: actions/setup-python@v2
      with:
        python-version: 3.11

    - name: Install dependencies
      run: |
        pip install -r requirements.txt
        pip install pytest pytest-cov

    - name: Run tests with coverage
      run: |
        pytest --cov=myapp --cov-report=xml

    - name: Upload coverage
      uses: codecov/codecov-action@v2
```

15.11 Esercizi

15.11.1 Livello Base

1. Scrivi 5 unit test per funzione `fibonacci(n)`
2. Usa setUp/tearDown per creare/pulire file temporaneo
3. Parametrizza test con pytest per testare 10 input diversi

15.11.2 Livello Intermedio

1. Implementa TDD per funzione `sort_students_by_grade()`
2. Mock API call a `requests.get()` e testa error handling

3. Raggiungi 80% coverage su modulo esistente
4. Usa pdb per debuggare bug in algoritmo ricorsivo

15.11.3 Livello Avanzato

1. Implementa test suite completa per mini-blog (User, Post, Comment)
2. Setup CI/CD con GitHub Actions che run test su Python 3.9, 3.10, 3.11
3. Profile applicazione e ottimizza bottleneck (target: 50% faster)
4. Crea custom pytest fixture per database test con rollback automatico

15.12 Riferimenti

- unittest Documentation: https://docs.python.org/3/library/unittest.html
- pytest Documentation: https://docs.pytest.org/
- unittest.mock: https://docs.python.org/3/library/unittest.mock.html
- pdb Debugger: https://docs.python.org/3/library/pdb.html
- Python Logging: https://docs.python.org/3/library/logging.html

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘:‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 16

Database con SQLite e SQLAlchemy

Obiettivi del capitolo

Dopo questo capitolo saprai:

- Comprendere database relazionali e SQL
- Usare SQLite3 per query e operazioni CRUD
- Definire modelli con SQLAlchemy ORM
- Gestire relationships (One-to-Many, Many-to-Many)
- Eseguire query complesse con SQLAlchemy
- Implementare transazioni e gestione errori
- Usare Alembic per migrazioni schema
- Evitare problemi comuni (N+1, SQL injection)
- Applicare best practices per performance e sicurezza

16.1 Introduzione

I database sono fondamentali per applicazioni moderne. Permettono di:

- **Persistenza**: Salvare dati permanentemente
- **Query**: Cercare e filtrare dati efficientemente
- **Integrità**: Garantire consistenza con constraints
- **Concorrenza**: Gestire accessi multipli simultanei
- **Scalabilità**: Gestire milioni di record

16.1.1 SQLite vs altri database

SQLite:

- File-based (un file `.db`)
- Nessun server da configurare

- Ideale per: prototipi, app desktop, testing, embedded systems
- Limiti: concorrenza limitata, no clustering

PostgreSQL/MySQL:

- Client-server architecture
- Alta concorrenza e performance
- Ideale per: produzione, web apps, alta scalabilità
- Richiede: installazione server, configurazione

16.1.2 SQL vs ORM

Raw SQL (sqlite3):

- Pro: Controllo completo, massima performance
- Contro: Verboso, ripetitivo, SQL injection se non attento

ORM (SQLAlchemy):

- Pro: Pythonic, type safety, meno boilerplate
- Contro: Curva apprendimento, overhead performance minimo

16.2 SQLite3: Database di Base

16.2.1 Connessione e creazione database

```python
import sqlite3

# Database in memoria (temporaneo)
conn = sqlite3.connect(":memory:")

# Database su file (persistente)
conn = sqlite3.connect("mydatabase.db")

# Row factory per dict invece di tuple
conn.row_factory = sqlite3.Row

cursor = conn.cursor()

# SEMPRE chiudere connessione
try:
    # ... operazioni database ...
    pass
finally:
    conn.close()

# Oppure usa context manager (raccomandato)
with sqlite3.connect("mydatabase.db") as conn:
    cursor = conn.cursor()
    # Operazioni qui
    # Commit automatico se no eccezioni
```

16.2.2 CREATE TABLE

```
import sqlite3

conn = sqlite3.connect("blog.db")
cursor = conn.cursor()

# Crea tabella users
cursor.execute("""
    CREATE TABLE IF NOT EXISTS users (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        username TEXT UNIQUE NOT NULL,
        email TEXT UNIQUE NOT NULL,
        password_hash TEXT NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
    )
""")

# Crea tabella posts con foreign key
cursor.execute("""
    CREATE TABLE IF NOT EXISTS posts (
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        user_id INTEGER NOT NULL,
        title TEXT NOT NULL,
        content TEXT NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        FOREIGN KEY (user_id) REFERENCES users (id)
            ON DELETE CASCADE
    )
""")

# Crea indici per performance
cursor.execute("CREATE INDEX IF NOT EXISTS idx_username ON users(
    username)")
cursor.execute("CREATE INDEX IF NOT EXISTS idx_user_id ON posts(user_id)
    ")

conn.commit()
conn.close()
```

16.2.3 INSERT: Prepared Statements

```
import sqlite3

conn = sqlite3.connect("blog.db")
cursor = conn.cursor()

# INSICURO - SQL Injection vulnerability!
# username = input("Username: ")  # Se utente inserisce: ' OR '1'='1
# cursor.execute(f"INSERT INTO users (username) VALUES ('{username}')")

# SICURO - Prepared statements con ? placeholder
username = "alice"
email = "alice@example.com"
password_hash = "hashed_password"

cursor.execute(
```

```
    "INSERT INTO users (username, email, password_hash) VALUES (?, ?, ?)",
    (username, email, password_hash)
)

# Ottieni ID auto-incrementato
user_id = cursor.lastrowid
print(f"Inserted user with ID: {user_id}")

# INSERT multipli (più efficiente)
users = [
    ("bob", "bob@example.com", "hash1"),
    ("charlie", "charlie@example.com", "hash2"),
    ("diana", "diana@example.com", "hash3")
]

cursor.executemany(
    "INSERT INTO users (username, email, password_hash) VALUES (?, ?, ?)",
    users
)

conn.commit()
conn.close()
```

16.2.4 SELECT: Query dati

```
import sqlite3

conn = sqlite3.connect("blog.db")
conn.row_factory = sqlite3.Row  # Row invece di tuple
cursor = conn.cursor()

# SELECT singolo
cursor.execute("SELECT * FROM users WHERE username = ?", ("alice",))
user = cursor.fetchone()

if user:
    print(f"ID: {user['id']}, Username: {user['username']}, Email: {user['email']}")
else:
    print("User not found")

# SELECT multipli
cursor.execute("SELECT * FROM users ORDER BY created_at DESC")
users = cursor.fetchall()

for user in users:
    print(f"{user['username']} - {user['email']}")

# SELECT con JOIN
cursor.execute("""
    SELECT users.username, posts.title, posts.created_at
    FROM posts
    JOIN users ON posts.user_id = users.id
    WHERE users.username = ?
    ORDER BY posts.created_at DESC
```

```
""", ("alice",))

posts = cursor.fetchall()
for post in posts:
    print(f"{post['username']}: {post['title']}")

# SELECT con aggregazione
cursor.execute("""
    SELECT users.username, COUNT(posts.id) as post_count
    FROM users
    LEFT JOIN posts ON users.id = posts.user_id
    GROUP BY users.id
    HAVING post_count > 0
""")

stats = cursor.fetchall()
for stat in stats:
    print(f"{stat['username']}: {stat['post_count']} posts")

conn.close()
```

16.2.5 UPDATE e DELETE

```
import sqlite3

conn = sqlite3.connect("blog.db")
cursor = conn.cursor()

# UPDATE
cursor.execute(
    "UPDATE users SET email = ? WHERE username = ?",
    ("newemail@example.com", "alice")
)

affected_rows = cursor.rowcount
print(f"Updated {affected_rows} row(s)")

# DELETE
cursor.execute("DELETE FROM posts WHERE id = ?", (5,))

if cursor.rowcount > 0:
    print("Post deleted")
else:
    print("Post not found")

conn.commit()
conn.close()
```

16.2.6 Transazioni

```
import sqlite3

conn = sqlite3.connect("blog.db")
cursor = conn.cursor()

try:
```

```
    # BEGIN TRANSACTION (implicito)
    cursor.execute(
        "INSERT INTO users (username, email, password_hash) VALUES (?, ?, ?)",
        ("eve", "eve@example.com", "hash")
    )

    user_id = cursor.lastrowid

    cursor.execute(
        "INSERT INTO posts (user_id, title, content) VALUES (?, ?, ?)",
        (user_id, "First Post", "Hello World!")
    )

    # Simula errore
    # raise ValueError("Something went wrong")

    # COMMIT se tutto OK
    conn.commit()
    print("Transaction committed successfully")

except Exception as e:
    # ROLLBACK se errore
    conn.rollback()
    print(f"Transaction rolled back: {e}")

finally:
    conn.close()
```

16.3 SQLAlchemy: ORM Potente

16.3.1 Installazione e setup

```
pip install sqlalchemy
```

16.3.2 Definizione modelli

```
from sqlalchemy import Column, Integer, String, Text, DateTime, ForeignKey, create_engine
from sqlalchemy.orm import declarative_base, relationship, sessionmaker
from datetime import datetime

Base = declarative_base()

class User(Base):
    __tablename__ = "users"

    id = Column(Integer, primary_key=True)
    username = Column(String(50), unique=True, nullable=False)
    email = Column(String(100), unique=True, nullable=False)
    password_hash = Column(String(255), nullable=False)
    created_at = Column(DateTime, default=datetime.utcnow)

    # Relationship (One-to-Many)
```

```
    posts = relationship("Post", back_populates="author", cascade="all, delete-orphan")

    def __repr__(self):
        return f"<User(id={self.id}, username='{self.username}')>"

class Post(Base):
    __tablename__ = "posts"

    id = Column(Integer, primary_key=True)
    user_id = Column(Integer, ForeignKey("users.id"), nullable=False)
    title = Column(String(200), nullable=False)
    content = Column(Text, nullable=False)
    created_at = Column(DateTime, default=datetime.utcnow)

    # Relationship (Many-to-One)
    author = relationship("User", back_populates="posts")

    def __repr__(self):
        return f"<Post(id={self.id}, title='{self.title}')>"

# Crea engine e tabelle
engine = create_engine("sqlite:///blog.db", echo=True)  # echo=True mostra SQL
Base.metadata.create_all(engine)

# Crea Session factory
Session = sessionmaker(bind=engine)
```

16.3.3 CRUD Operations con SQLAlchemy

```
from sqlalchemy.orm import sessionmaker

Session = sessionmaker(bind=engine)
session = Session()

# CREATE
alice = User(
    username="alice",
    email="alice@example.com",
    password_hash="hashed_password"
)

session.add(alice)
session.commit()

print(f"Created user: {alice.id}")

# CREATE con relationship
post = Post(
    title="My First Post",
    content="Hello World!",
    author=alice  # Imposta relationship automaticamente
)

session.add(post)
session.commit()
```

```python

# READ - query singolo
user = session.query(User).filter_by(username="alice").first()
print(user)

# READ - query multipli
users = session.query(User).all()
for user in users:
    print(f"{user.username} - {user.email}")

# READ con WHERE
recent_posts = session.query(Post).filter(
    Post.created_at > datetime(2024, 1, 1)
).order_by(Post.created_at.desc()).all()

# READ con JOIN (automatico via relationship)
user = session.query(User).filter_by(username="alice").first()
for post in user.posts:  # Lazy loading
    print(f"Post: {post.title}")

# UPDATE
user = session.query(User).filter_by(username="alice").first()
user.email = "newemail@example.com"
session.commit()

# DELETE
post = session.query(Post).filter_by(id=1).first()
if post:
    session.delete(post)
    session.commit()

session.close()
```

16.3.4 Query avanzate

```python
from sqlalchemy import func, and_, or_

session = Session()

# COUNT
user_count = session.query(func.count(User.id)).scalar()
print(f"Total users: {user_count}")

# JOIN esplicito
results = session.query(User, Post).join(Post).filter(
    User.username == "alice"
).all()

for user, post in results:
    print(f"{user.username}: {post.title}")

# AND / OR
users = session.query(User).filter(
    and_(
        User.username.like("a%"),
        User.created_at > datetime(2024, 1, 1)
    )
```

```
).all()

# IN
usernames = ["alice", "bob", "charlie"]
users = session.query(User).filter(User.username.in_(usernames)).all()

# ORDER BY e LIMIT
top_posts = session.query(Post).order_by(
    Post.created_at.desc()
).limit(10).all()

# GROUP BY e HAVING
from sqlalchemy import func

stats = session.query(
    User.username,
    func.count(Post.id).label("post_count")
).join(Post).group_by(User.id).having(
    func.count(Post.id) > 5
).all()

for username, count in stats:
    print(f"{username}: {count} posts")

session.close()
```

16.3.5 Relationships: Many-to-Many

```
from sqlalchemy import Table

# Association table per Many-to-Many
post_tags = Table(
    "post_tags",
    Base.metadata,
    Column("post_id", Integer, ForeignKey("posts.id"), primary_key=True),
    Column("tag_id", Integer, ForeignKey("tags.id"), primary_key=True)
)

class Tag(Base):
    __tablename__ = "tags"

    id = Column(Integer, primary_key=True)
    name = Column(String(50), unique=True, nullable=False)

    # Many-to-Many relationship
    posts = relationship("Post", secondary=post_tags, back_populates="tags")

    def __repr__(self):
        return f"<Tag(name='{self.name}')>"

# Modifica Post model per aggiungere tags
class Post(Base):
    # ... (campi esistenti) ...

```

```
    tags = relationship("Tag", secondary=post_tags, back_populates="
        posts")

# Uso
session = Session()

# Crea tags
python_tag = Tag(name="python")
database_tag = Tag(name="database")

session.add_all([python_tag, database_tag])

# Crea post con tags
post = Post(
    title="SQLAlchemy Tutorial",
    content="Learn SQLAlchemy...",
    author=alice,
    tags=[python_tag, database_tag]  # Many-to-Many
)

session.add(post)
session.commit()

# Query posts by tag
posts_with_python = session.query(Post).join(Post.tags).filter(
    Tag.name == "python"
).all()

session.close()
```

16.3.6 Evitare N+1 Query Problem

```
from sqlalchemy.orm import joinedload, selectinload

session = Session()

# PROBLEMA: N+1 queries (inefficiente!)
users = session.query(User).all()
for user in users:  # 1 query
    for post in user.posts:  # N queries (una per user!)
        print(post.title)

# SOLUZIONE 1: joinedload (INNER JOIN, 1 query)
users = session.query(User).options(
    joinedload(User.posts)
).all()

for user in users:
    for post in user.posts:  # Nessuna query aggiuntiva!
        print(post.title)

# SOLUZIONE 2: selectinload (2 queries totali)
users = session.query(User).options(
    selectinload(User.posts)
).all()

# Con nested relationships
```

```
users = session.query(User).options(
    joinedload(User.posts).joinedload(Post.tags)
).all()

session.close()
```

16.3.7 Transazioni e gestione errori

```
from sqlalchemy.exc import IntegrityError

session = Session()

try:
    # Transazione complessa
    user = User(username="frank", email="frank@example.com",
       password_hash="hash")
    session.add(user)
    session.flush()  # Esegue INSERT ma non commit

    post = Post(title="Test", content="Content", author=user)
    session.add(post)

    # Simula errore
    # raise ValueError("Something went wrong")

    session.commit()  # Commit tutto
    print("Transaction successful")

except IntegrityError as e:
    session.rollback()
    print(f"Integrity error (es. UNIQUE constraint): {e}")

except Exception as e:
    session.rollback()
    print(f"Transaction failed: {e}")

finally:
    session.close()

# Context manager (raccomandato)
with Session() as session:
    with session.begin():
        user = User(username="grace", email="grace@example.com",
           password_hash="hash")
        session.add(user)
        # Auto-commit se no eccezioni, auto-rollback se errore
```

16.4 Migrazioni con Alembic

16.4.1 Setup Alembic

```
pip install alembic

# Inizializza Alembic
alembic init alembic
```

```

# Modifica alembic.ini: imposta sqlalchemy.url
# sqlalchemy.url = sqlite:///blog.db

# Modifica alembic/env.py: importa Base
# from myapp.models import Base
# target_metadata = Base.metadata
```

16.4.2 Creazione migration

```
# Crea migration automatica (da differenza modelli)
alembic revision --autogenerate -m "Add users and posts tables"

# Applica migration
alembic upgrade head

# Rollback migration
alembic downgrade -1

# Storia migrazioni
alembic history
```

16.4.3 Migration manuale

```
# File: alembic/versions/xxxx_add_bio_to_users.py

from alembic import op
import sqlalchemy as sa

def upgrade():
    # Aggiungi colonna
    op.add_column("users", sa.Column("bio", sa.Text, nullable=True))

    # Crea indice
    op.create_index("idx_email", "users", ["email"])

def downgrade():
    # Rimuovi indice
    op.drop_index("idx_email", table_name="users")

    # Rimuovi colonna
    op.drop_column("users", "bio")
```

Best Practices

- Usa prepared statements (sempre!) per evitare SQL injection
- Chiudi sempre connessioni/sessioni (usa context manager)
- Usa transazioni per operazioni multi-step
- Evita N+1 queries con joinedload/selectinload
- Indicizza colonne usate in WHERE, JOIN, ORDER BY
- Usa Alembic per migrazioni (no ALTER TABLE manuale)

- Gestisci eccezioni (IntegrityError, OperationalError)
- Testa query complesse con EXPLAIN QUERY PLAN
- Usa connection pooling in produzione
- Backup database regolarmente

Errori Comuni

- SQL injection con f-strings invece di ? placeholders
- Dimenticare commit() dopo INSERT/UPDATE/DELETE
- Non chiudere sessioni (resource leak)
- N+1 queries con lazy loading
- Non gestire IntegrityError (UNIQUE, FK violations)
- Usare `session.query(User).all()` per grandi dataset (usa limit/offset)
- Non usare indici (query lente)
- Modificare schema senza migration (inconsistenze)
- Non usare transazioni per operazioni critiche
- Esporre oggetti SQLAlchemy direttamente in API (usa DTO/serializer)

16.5 Pattern e Best Practices

16.5.1 Repository Pattern

```
from typing import Optional, List
from sqlalchemy.orm import Session

class UserRepository:
    def __init__(self, session: Session):
        self.session = session

    def get_by_id(self, user_id: int) -> Optional[User]:
        return self.session.query(User).filter_by(id=user_id).first()

    def get_by_username(self, username: str) -> Optional[User]:
        return self.session.query(User).filter_by(username=username).
            first()

    def get_all(self, limit: int = 100, offset: int = 0) -> List[User]:
        return self.session.query(User).limit(limit).offset(offset).all
            ()

    def create(self, username: str, email: str, password_hash: str) ->
        User:
        user = User(username=username, email=email, password_hash=
            password_hash)
        self.session.add(user)
```

```
        self.session.commit()
        return user

    def update(self, user: User) -> User:
        self.session.commit()
        return user

    def delete(self, user: User) -> None:
        self.session.delete(user)
        self.session.commit()

# Uso
with Session() as session:
    repo = UserRepository(session)

    user = repo.create("henry", "henry@example.com", "hash")
    print(f"Created: {user.id}")

    found = repo.get_by_username("henry")
    print(f"Found: {found.email}")
```

16.5.2 Database seeding

```
def seed_database():
    """Popola database con dati iniziali"""
    with Session() as session:
        # Verifica se già popolato
        if session.query(User).count() > 0:
            print("Database already seeded")
            return

        # Crea utenti
        users = [
            User(username="admin", email="admin@blog.com", password_hash
                ="hash1"),
            User(username="alice", email="alice@blog.com", password_hash
                ="hash2"),
            User(username="bob", email="bob@blog.com", password_hash="
                hash3"),
        ]

        session.add_all(users)
        session.commit()

        # Crea tags
        tags = [
            Tag(name="python"),
            Tag(name="database"),
            Tag(name="web"),
        ]

        session.add_all(tags)
        session.commit()

        # Crea posts
        posts = [
```

```
            Post(title="Python Tutorial", content="...", author=users[0], tags=[tags[0]]),
            Post(title="Database Guide", content="...", author=users[1], tags=[tags[1]]),
            Post(title="Web Development", content="...", author=users[2], tags=[tags[2]]),
        ]

        session.add_all(posts)
        session.commit()

        print("Database seeded successfully")

if __name__ == "__main__":
    seed_database()
```

16.6 Troubleshooting

Problema: IntegrityError: UNIQUE constraint failed

```
# Gestisci duplicati
try:
    user = User(username="alice", email="alice@example.com", password_hash="hash")
    session.add(user)
    session.commit()
except IntegrityError:
    session.rollback()
    print("User already exists")
```

Problema: DetachedInstanceError (oggetto non in sessione)

```
# Riattacca oggetto
session.add(user)  # Merge oggetto nella sessione

# Oppure usa expire_on_commit=False
Session = sessionmaker(bind=engine, expire_on_commit=False)
```

Problema: Query lente

```
# Analizza query con EXPLAIN
# In SQLite
EXPLAIN QUERY PLAN SELECT * FROM posts WHERE user_id = 5;

# Aggiungi indici
CREATE INDEX idx_user_id ON posts(user_id);
```

16.7 Esercizi

16.7.1 Livello Base

1. Crea database SQLite con tabella `books` (id, title, author, year)
2. Inserisci 10 libri e query tutti con `SELECT *`
3. Trova libri pubblicati dopo 2000 con prepared statement

16.7.2 Livello Intermedio

1. Definisci modelli SQLAlchemy per `Author` e `Book` (One-to-Many)
2. Implementa CRUD completo con Repository pattern
3. Aggiungi relationship Many-to-Many per `Book` e `Category`
4. Query libri con `joinedload` per evitare N+1

16.7.3 Livello Avanzato

1. Implementa sistema blog completo (User, Post, Comment, Tag)
2. Usa Alembic per creare migrations
3. Implementa pagination per lista posts
4. Aggiungi full-text search con FTS5 (SQLite)
5. Crea seed script per popolare database di test

16.8 Riferimenti

- SQLite Documentation: https://docs.python.org/3/library/sqlite3.html
- SQLAlchemy Documentation: https://docs.sqlalchemy.org/
- Alembic Tutorial: https://alembic.sqlalchemy.org/
- SQL Tutorial: https://www.w3schools.com/sql/

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘:‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 17

Introduzione a NAO (V6) e NAOqi

17.1 Contesto e Applicazioni

Contesto e Applicazioni

- Didattica STEM e laboratori universitari. - Social robotics, interazione uomo-robot (HRI). - Demo e prototipi per mostre ed eventi. - Sperimentazione di visione, linguaggio e movimento.

17.2 Approfondimenti

NAO (V6) utilizza NAOqi 2.x: un framework di servizi (*AL*) accessibili via rete. La programmazione in Python avviene tipicamente creando una sessione (`qi`) e recuperando i servizi, come `ALTextToSpeech`, `ALMotion`, `ALRobotPosture`, `ALVideoDevice`.

Prime connessioni

```
# Connessione ai servizi NAOqi (NAO V6)
import qi

session = qi.Session()
session.connect("tcp://nao.local:9559")  # sostituisci host/IP

tts = session.service("ALTextToSpeech")
tts.say("Ciao! Sono NAO")

motion = session.service("ALMotion")
posture = session.service("ALRobotPosture")
# Passa alla postura base prima di muovere
posture.goToPosture("StandInit", 0.5)
```

Note pratiche Assicurare connettività (Wi-Fi/Ethernet) e versione corretta del SDK. Per stabilità, gestire eccezioni di rete e latenza; loggare operazioni critiche.

Guida rapida per programmatori non-Python

Python è tipizzato dinamicamente e l'indentazione fa parte della sintassi (blocchi). Le eccezioni sono il meccanismo standard di errore; non si usano codici di ritorno come in C. Le liste (array dinamici) e i dizionari (mappe chiave->valore) sono strutture fondamentali.

- Import: `import qi` carica il modulo del SDK NAOqi. - Sessione: `qi.Session()` crea un oggetto che gestisce la connessione TCP verso il robot. - Servizi: `session.service("ALMotion")` risolve il servizio remoto e ritorna un proxy su cui invocare metodi.
Gestione errori: usare `try/except` per catturare `RuntimeError` quando la connessione fallisce. **Stringhe**: sono Unicode; passare testo direttamente a `tts.say`. **Convenzione**: nomi come `goToPosture` sono in stile CamelCase perché ereditati dalle API NAOqi.

Connessione commentata passo-passo

```python
import qi

def connect(host="nao.local", port=9559):
    """Stabilisce una sessione TCP con NAO.
    - host: hostname o IP del robot
    - port: porta NAOqi (default 9559)
    Ritorna: qi.Session connessa.
    """
    session = qi.Session()
    try:
        session.connect(f"tcp://{host}:{port}")
    except RuntimeError as e:
        # In Python le eccezioni si catturano per tipo; qui gestiamo problemi di rete
        print("Connessione fallita:", e)
        raise
    return session

session = connect()
tts = session.service("ALTextToSpeech")  # Proxy al servizio TTS
motion = session.service("ALMotion")      # Proxy al servizio di movimento
posture = session.service("ALRobotPosture")

# Prima di muovere: posizionare NAO in postura stabile
posture.goToPosture("StandInit", 0.5)  # 0.5 = fattore di velocità (0..1)
tts.say("Pronto a muovermi!")
```

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 18

Setup Ambiente e API NAOqi

18.1 Contesto e Applicazioni

Contesto e Applicazioni

- Preparazione ambiente di sviluppo e dipendenze. - Connessione sicura al robot e gestione rete. - Struttura di progetto per script e demo.

18.2 Approfondimenti

Per NAOqi 2.x in Python si usa tipicamente il pacchetto `qi` per la sessione e i servizi. Organizzare un piccolo progetto con virtualenv e file di configurazione (host/IP, porte, credenziali se presenti).

Sessione e servizi

```python
import qi

def connect(address="tcp://nao.local:9559"):
    session = qi.Session()
    try:
        session.connect(address)
    except RuntimeError as e:
        print("Connessione fallita:", e)
        raise
    return session

session = connect()
tts = session.service("ALTextToSpeech")
posture = session.service("ALRobotPosture")
motion = session.service("ALMotion")
```

Networking Preferire IP statico o hostname risolvibile; verificare porte aperte e firewall. Per contesti multi-robot, definire discovery e registri di indirizzi.

Spiegazione per sviluppatori non-Python

Virtualenv: in Python si crea un ambiente isolato per dipendenze con `python -m venv .venv` e si attiva prima di installare pacchetti. **Configurazione**: usare un file `config.json` o variabili d'ambiente per host/porta, quindi leggerli con il modulo `json`/`os`.
Pattern di connessione:

```python
import os, qi

HOST = os.getenv("NAO_HOST", "nao.local")
PORT = int(os.getenv("NAO_PORT", "9559"))

def get_session():
    s = qi.Session()
    s.connect(f"tcp://{HOST}:{PORT}")
    return s

session = get_session()
tts = session.service("ALTextToSpeech")
tts.say("Configurazione caricata da ENV")
```

Eccezioni e retry: implementare un semplice *retry* con backoff se la connessione fallisce; usare `time.sleep` e un numero massimo di tentativi.

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘:‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Capitolo 19

Movimento e Visione

19.1 Contesto e Applicazioni

Contesto e Applicazioni

- Gesture e posture per interazioni HRI. - Acquisizione immagini per analisi e riconoscimento. - Demo di navigazione e percezione di base.

19.2 Approfondimenti

Il servizio `ALMotion` controlla giunti e camminata, mentre `ALRobotPosture` gestisce posture predefinite. `ALVideoDevice` fornisce flussi dalle camere.

Esempi di movimento

```
import qi, time

session = qi.Session(); session.connect("tcp://nao.local:9559")
motion = session.service("ALMotion")
posture = session.service("ALRobotPosture")

posture.goToPosture("StandInit", 0.5)
# Muove il braccio destro (pitch/roll)
names = ["RShoulderPitch", "RShoulderRoll"]
angles = [0.5, -0.2]   # radianti
speed = 0.2
motion.setAngles(names, angles, speed)
time.sleep(1)
posture.goToPosture("Stand", 0.5)
```

Acquisizione immagini

```
import qi
session = qi.Session(); session.connect("tcp://nao.local:9559")
vid = session.service("ALVideoDevice")

# Iscrizione con API esplicita per camera e spazio colore
# cameraIndex: 0 = frontale, 1 = bottom, 2 = 3D (se disponibile)
# resolution: 8=QVGA(320x240), 11=VGA(640x480), ecc.
# colorSpace: 13=BGR, 11=YUV, ecc. (vedi documentazione NAOqi)
```

```
name = vid.subscribeCamera("python_client", 0, 11, 13, 10)
frame = vid.getImageRemote(name)
vid.unsubscribe(name)

# frame è una lista: [width, height, layers, colorSpace, timestamp,
    data]
width, height = frame[0], frame[1]
data = frame[6]  # buffer raw (bytes)

# Scrittura grezza su file (per ispezione)
with open("frame.raw", "wb") as f:
    f.write(bytearray(data))

# Per salvare come PNG/JPEG, usare PIL (Pillow) e convertire BGR->
    RGB
# from PIL import Image; Image.frombytes(...)
```

Sicurezza Impostare posture stabili prima di muovere; evitare collisioni e mantenere velocità moderate. Analizzare eccezioni e latenza di rete.

Spiegazioni per non-Python

Liste vs tuple: il risultato di `getImageRemote` è una lista; si accede per indice. In Python le tuple sono immutabili, le liste no. **Bytes**: la variabile `data` è una sequenza di byte; si può convertire con `bytearray` o `bytes` per scrittura su file. **Context manager**: l'istruzione `with open(...)` gestisce automaticamente apertura/chiusura file (equivalente a `try/finally`).

Guida per programmatori non-Python

Questa guida riassume come leggere e capire codice Python se provieni da altri linguaggi. - Sintassi a indentazione: blocchi definiti da indentazione (4 spazi), niente ‘‘. - Tipi dinamici: il tipo appartiene all’oggetto; le variabili sono riferimenti. - Truthiness: oggetti vuoti e zero sono ‘False‘; altrimenti ‘True‘. - Strutture chiave: ‘list‘ (array dinamico), ‘dict‘ (mappa), ‘set‘ (insieme), ‘tuple‘ (immutabile). - Slice e comprensioni: ‘seq[a:b:c]‘ e ‘[f(x) for x in xs if cond]‘. - Funzioni: prima classe, closure e default arguments; attenzione ai default mutabili. - Errori: eccezioni con ‘try/except‘; EAFP (fallo e gestisci) spesso preferito a LBYL. - Moduli e package: import espliciti, nomi in `snake_case`, `__name__ == "__main__"` per entrypoint. - Stile: PEP 8 per nomi, lunghezza righe, spazi; scrivere docstring.
Esempio di lettura codice:

```
def process(items):
    """Filtra e trasforma i dati; nessun I/O qui."""
    # Comprensione: compatta, dichiarativa
    return [x * 2 for x in items if x > 0]

def main():
    data = [1, -1, 3]
    result = process(data)
    print(result)  # I/O separato dalla logica

if __name__ == "__main__":
    main()
```

Suggerimenti pratici: - Separare logica pura (facile da testare) e I/O. - Usare ambienti virtuali per dipendenze; evitare globali. - Preferire nomi descrittivi e funzioni piccole; aggiungere docstring.

Appendice A

Soluzioni agli Esercizi

Modulo 00 — Fondamenti Python

Esercizio 1

```python
nome = "Ada"
eta = 27
print(f"Mi chiamo {nome} e ho {eta} anni")
```

Esercizio 2

```python
import math
r = float(input("Raggio: "))
area = math.pi * (r ** 2)
print(f"Area: {area}")
```

Esercizio 3

```python
a = float(input("Primo numero: "))
b = float(input("Secondo numero: "))
print("Somma:", a + b)
print("Differenza:", a - b)
print("Prodotto:", a * b)
print("Divisione:", a / b if b != 0 else "Impossibile")
```

Esercizio 4

```python
c = float(input("Celsius: "))
f = c * 9 / 5 + 32
print(f"Fahrenheit: {f}")
```

Esercizio 5

```python
items = ["uno", "due", "tre"]
for it in items:
    print(it)
```

Appendice B

Appendice: Soluzioni e Approfondimenti

Appendice C

Soluzioni agli Esercizi

Modulo 00 — Fondamenti Python

Esercizio 1

```python
nome = "Ada"
eta = 27
print(f"Mi chiamo {nome} e ho {eta} anni")
```

Esercizio 2

```python
import math
r = float(input("Raggio: "))
area = math.pi * (r ** 2)
print(f"Area: {area}")
```

Esercizio 3

```python
a = float(input("Primo numero: "))
b = float(input("Secondo numero: "))
print("Somma:", a + b)
print("Differenza:", a - b)
print("Prodotto:", a * b)
print("Divisione:", a / b if b != 0 else "Impossibile")
```

Esercizio 4

```python
c = float(input("Celsius: "))
f = c * 9 / 5 + 32
print(f"Fahrenheit: {f}")
```

Esercizio 5

```python
items = ["uno", "due", "tre"]
for it in items:
    print(it)
```