Arcadia J. Darell

Guida pratica a Python

Dalle basi alla programmazione avanzata

Arcadia J. Darell

Guida pratica a Python

Dalle basi alla programmazione avanzata

ARCADIA J. DARELL

Guida pratica a Python

Dalle basi alla programmazione avanzata

UUID: de135aab-98a7-4b7a-92d7-9d5a76408fc8

Questo libro è stato realizzato con StreetLib Write
https://writeapp.io

Indice dei contenuti

1. Copyright

2. Indice

3. Ringraziamenti

4. Capitolo 1: Introduzione a Python

5. 1.2 Perché scegliere Python?

6. 1.3 Installazione e configurazione dell' ambiente di sviluppo

7. Capitolo 2: Sintassi di base di Python | 2.1 Variabili e tipi di dati

8. 2.2 Operatori

9. 2.3 Strutture di controllo (if, else, elif)

10. Capitolo 3: Strutture dati in Python

11. 3.2 Tuple

12. 3.3 Dizionari

13. 3.4 Set

14. Capitolo 4: Funzioni e Moduli

15. 4.2 Parametri e argomenti

16. 4.3 Moduli e pacchetti

17. Capitolo 5: Programmazione Orientata agli Oggetti (OOP)

18. 5.2 Ereditarietà

19. 5.3 Polimorfismo e Incapsulamento

20. Approfondimento sull'OOP

21. Capitolo 6: Gestione delle eccezioni

22. 6.2 Blocco try-except

23. 6.3 Creazione di eccezioni personalizzate

24. Approfondimento sulla gestione delle eccezioni

25. Capitolo 7: Input/Output e File

26. 7.2 Gestione dei file CSV

27. 7.3 Operazioni con i file JSON

28. Capitolo 8: Python e Database

29. 8.2 Connessione a Database SQLite

30. 8.3 Operazioni CRUD (Create, Read, Update, Delete)

31. Capitolo 9: Sviluppo web con Python

32. 9.2 Creazione di una semplice applicazione web

33. 9.3 Introduzione a Django

34. Capitolo 10: Data Science con Python

35. 10.2 Analisi dei dati

36. 10.3 Visualizzazione dei dati con Matplotlib e Seaborn

37. Capitolo 11: Machine Learning con Python

38. 11.2 Librerie principali: Scikit-learn

39. 11.3 Creazione di modelli di machine learning

40. Capitolo 12: Automazione e scripting

41. 12.2 Parsing di file e automazione web

42. 12.3 Scripting di sistema

43. Capitolo 13: Testing e debugging

44. 13.2 Unittest e Pytest

45. 13.3 Tecniche di Debugging

46. Capitolo 14: Python avanzato

47. 14.2 Generatori

48. 14.3 Context manager

49. Capitolo 15: Progetti pratici

50. 15.2 Sviluppo di un 'Applicazione Web Completa

51. 15.3 Implementazione di un progetto di data science

52. Conclusione

COPYRIGHT

Prima edizione: Agosto, 2024

INDICE

Capitolo 6: Gestione delle eccezioni
6.1 Tipi di eccezioni
6.2 Blocco try-except
6.3 Creazione di eccezioni personalizzate
• Approfondimento sulla gestione delle eccezioni
Capitolo 7: Input/Output e file
7.1 Lettura e scrittura di file
7.2 Gestione dei file CSV
7.3 Operazioni con i file JSON
Capitolo 8: Python e database
8.1 Introduzione ai database
8.2 Connessione a database SQLite
8.3 Operazioni CRUD (Create, Read, Update, Delete)
Capitolo 9: Sviluppo web con Python
9.1 Introduzione a Flask
9.2 Creazione di una semplice applicazione web
9.3 Introduzione a Django
Capitolo 10: Data Science con Python
10.1 Librerie principali: NumPy e Pandas
10.2 Analisi dei dati
10.3 Visualizzazione dei dati con Matplotlib e Seaborn
Capitolo 11: Machine Learning con Python
11.1 Introduzione al machine learning
11.2 Librerie principali: Scikit-learn
11.3 Creazione di modelli di machine learning
Capitolo 12: Automazione e scripting
12.1 Automazione delle attività con Python
12.2 Parsing di file e automazione web
12.3 Scripting di sistema
Capitolo 13: Testing e debugging
13.1 Introduzione al testing
13.2 Unittest e pytest
13.3 Tecniche di debugging
Capitolo 14: Python avanzato
14.1 Decoratori
14.2 Generatori

14.3 Context manager
Capitolo 15: Progetti pratici
15.1 Creazione di un'applicazione CLI
15.2 Sviluppo di un'applicazione web completa
15.3 Implementazione di un progetto di data science

RINGRAZIAMENTI

Alla mia famiglia, a Domenico, Nicola, Maria Carmela,
Nora e Valentina.

CAPITOLO 1: INTRODUZIONE A PYTHON

1.1 Storia e background di Python

- Python è un linguaggio di programmazione di alto livello, noto per la sua leggibilità e la sintassi semplice. Creato da Guido van Rossum nei tardi anni '80, Python ha una storia ricca e interessante che ne ha plasmato le caratteristiche distintive. In questo sotto-capitolo, esploreremo le origini, gli obiettivi di design, l'evoluzione e l'impatto della comunità sullo sviluppo di Python.

Le origini di Python

Python nasce dal desiderio di Guido van Rossum di creare un linguaggio di programmazione che combinasse la potenza e la flessibilità dei linguaggi più complessi con la semplicità e la facilità d'uso dei linguaggi di scripting. Van Rossum lavorava presso il Centrum Wiskunde & Informatica (CWI) nei Paesi Bassi quando iniziò a sviluppare Python come successore del linguaggio ABC. ABC era un linguaggio progettato per essere semplice da usare, ma mancava di alcune funzionalità chiave che i programmatori richiedevano per progetti più complessi.

Durante il Natale del 1989, van Rossum iniziò a lavorare su Python. Il nome "Python" non deriva dal

serpente, come molti potrebbero pensare, ma dal gruppo comico britannico Monty Python, che van Rossum ammirava. Il primo rilascio di Python, la versione 0.9.0, avvenne nel febbraio 1991. Questa versione iniziale includeva già molte delle funzionalità che ancora oggi caratterizzano Python, come l'uso delle indentazioni per delimitare i blocchi di codice e la gestione delle eccezioni.

Gli obiettivi di design di Python

Van Rossum ha concepito Python con alcuni obiettivi di design chiave, che ancora oggi ne guidano lo sviluppo. Tra questi:

1. **Leggibilità del codice**:
    - Uno degli obiettivi principali di Python è quello di rendere il codice facile da leggere. Questo è ottenuto attraverso una sintassi chiara e concisa che utilizza l'indentazione per definire i blocchi di codice, evitando l'uso di parentesi graffe o parole chiave come "begin" e "end". La leggibilità del codice facilita la manutenzione e la collaborazione tra sviluppatori, riducendo la probabilità di errori e migliorando la qualità del software.
2. **Produttività dello sviluppatore**:
    - Python mira a minimizzare la complessità del codice, permettendo ai programmatori di scrivere codice efficiente in meno tempo. La sintassi semplice e intuitiva di Python permette di esprimere concetti complessi in poche linee di codice. Questo aumenta la produttività dello sviluppatore, consentendo di concentrarsi sulla

logica dell'applicazione piuttosto che sulla gestione dei dettagli del linguaggio.

3. **Portabilità**:
   - Python è progettato per essere un linguaggio multipiattaforma. Questo significa che il codice Python può essere eseguito su una vasta gamma di sistemi operativi senza modifiche. Python è disponibile su numerose piattaforme, dal desktop ai server ai dispositivi mobili. La portabilità di Python lo rende una scelta ideale per sviluppare applicazioni che devono funzionare su diversi ambienti.
4. **Estendibilità**:
   - Python è altamente estendibile, il che significa che può essere facilmente integrato con altri linguaggi e sistemi, come C, C++, e Java. Questa flessibilità rende Python ideale per una vasta gamma di applicazioni, dal web development alla data science. Le estensioni permettono di ottimizzare parti critiche del codice per migliorare le prestazioni o per sfruttare librerie esistenti scritte in altri linguaggi.

Evoluzione di Python

Nel corso degli anni, Python ha subito diverse evoluzioni significative, con il rilascio di nuove versioni che hanno introdotto miglioramenti e nuove funzionalità. La comunità di sviluppatori e utenti di Python ha giocato un ruolo cruciale in questo processo, contribuendo con feedback, bug report e nuove idee.

Python 2.x

Rilasciato nel 2000, Python 2.0 ha introdotto nuove funzionalità come la comprensione delle liste e il garbage collection. La comprensione delle liste ha reso possibile creare nuove liste applicando un'espressione a ogni elemento di una lista esistente. Il garbage collection ha migliorato la gestione della memoria automatizzando il processo di recupero della memoria non più utilizzata.

Python 2.x è stato ampiamente utilizzato per molti anni, diventando una scelta popolare per lo sviluppo di applicazioni web, scripting e automazione. Tuttavia, la necessità di mantenere la compatibilità con il codice esistente ha limitato l'introduzione di alcune migliorie e ha portato alla decisione di sviluppare una nuova versione del linguaggio che avrebbe rotto la compatibilità con Python 2.x.

Python 2.x è stato dichiarato obsoleto il 1° gennaio 2020, e il supporto ufficiale è stato interrotto. Questo ha segnato la fine di un'era e l'inizio della transizione verso Python 3.x.

Python 3.x

Rilasciato nel 2008, Python 3.0 ha rappresentato un'importante svolta, con numerosi cambiamenti che non erano retrocompatibili con Python 2.x. Python 3 è stato progettato per correggere i difetti del design di Python 2.x e per introdurre nuove funzionalità che avrebbero reso il linguaggio più coerente e facile da usare.

Le modifiche principali includevano una miglior gestione delle stringhe Unicode, una sintassi più chiara per la divisione dei numeri interi, e l'eliminazione di funzionalità obsolete. La gestione delle stringhe Unicode ha reso Python 3 più adatto per le applicazioni che devono gestire testi internazionali, mentre la nuova

sintassi per la divisione dei numeri interi ha ridotto le possibilità di errori di programmazione.

La migrazione da Python 2 a Python 3 è stata graduale, con molti progetti che hanno continuato a utilizzare Python 2.x per diversi anni. Tuttavia, la comunità Python ha lavorato attivamente per incoraggiare la migrazione, fornendo strumenti e guide per aiutare gli sviluppatori a portare il loro codice a Python 3.x. Oggi, Python 3.x è la versione standard utilizzata dalla comunità, e la maggior parte dei nuovi progetti viene sviluppata direttamente in Python 3.

La comunità Python

Un fattore chiave nel successo di Python è la sua vivace e attiva comunità. La comunità Python contribuisce in molti modi, rendendo Python non solo un linguaggio di programmazione, ma anche una piattaforma collaborativa in continua evoluzione.

1. **Sviluppo del linguaggio**:
    - Python è sviluppato come un progetto open source, con contributi provenienti da tutto il mondo. Il processo di sviluppo è gestito dalla Python Software Foundation (PSF), un'organizzazione no-profit che promuove e supporta l'uso di Python. La PSF organizza anche la conferenza annuale PyCon, che riunisce sviluppatori di Python da tutto il mondo per condividere idee e collaborare.
2. **Documentazione e tutorial**:
    - La comunità fornisce una vasta gamma di risorse educative, incluse documentazione ufficiale, tutorial, e corsi online. La documentazione ufficiale di Python è considerata uno dei migliori

esempi di documentazione di un linguaggio di programmazione, fornendo guide dettagliate, esempi di codice e riferimenti completi alle librerie standard.

3. **Framework e librerie**:
    - Numerosi framework e librerie sono stati sviluppati per Python, estendendone l'utilizzo in settori come lo sviluppo web (Django, Flask), la data science (NumPy, Pandas, Matplotlib), il machine learning (TensorFlow, Scikit-learn) e molti altri. Questi strumenti hanno reso Python una scelta popolare per una vasta gamma di applicazioni, dal web development alla ricerca scientifica.

Python oggi

Oggi, Python è utilizzato in una vasta gamma di applicazioni e industrie. Alcuni esempi includono:

1. **Sviluppo web**:
    - Python è utilizzato per costruire applicazioni web scalabili e sicure. Framework come Django e Flask offrono potenti strumenti per lo sviluppo web, rendendo Python una scelta popolare per sviluppatori di tutte le esperienze.
2. **Data science e machine learning**:
    - Le librerie Python sono ampiamente utilizzate per l'analisi dei dati, la visualizzazione e il machine learning. Strumenti come Pandas, NumPy, Matplotlib e Scikit-learn hanno reso Python una scelta preferita per data scientist e ricercatori.
3. **Automazione**:

- Python è utilizzato per automatizzare compiti ripetitivi e gestire configurazioni di sistema. Librerie come Selenium e BeautifulSoup permettono di automatizzare il web scraping, mentre strumenti come Ansible utilizzano Python per l'automazione della configurazione.

4. **Sviluppo software**:
   - Python è utilizzato per sviluppare una varietà di software, dai giochi alle applicazioni desktop. Framework come Pygame permettono di sviluppare giochi in Python, mentre strumenti come PyQt e Tkinter offrono librerie per lo sviluppo di interfacce grafiche.
5. **IoT e sviluppo embedded**:
   - Python sta guadagnando terreno anche nel campo dell'Internet delle cose (IoT) e dello sviluppo embedded. Framework come MicroPython permettono di eseguire Python su dispositivi con risorse limitate, mentre librerie come GPIO Zero facilitano la programmazione di dispositivi hardware come i Raspberry Pi.

Il futuro di Python

Il futuro di Python sembra promettente, con un continuo aumento della sua popolarità e una comunità sempre più ampia che continua a contribuire alla sua crescita e evoluzione. La continua innovazione nella comunità Python, combinata con la sua semplicità e flessibilità, garantisce che Python rimarrà un linguaggio di programmazione fondamentale per molti anni a venire.

1. **Adattabilità alle nuove tecnologie**:

    - Python continua ad adattarsi alle nuove tecnologie emergenti. Ad esempio, con l'aumento dell'importanza dell'intelligenza artificiale e del machine learning, Python ha visto un'espansione delle librerie e degli strumenti dedicati a questi campi, come TensorFlow, Keras e PyTorch.
2. **Espansione nell'educazione**:
    - Python è sempre più utilizzato come linguaggio di insegnamento nelle scuole e nelle università, grazie alla sua semplicità e alla sua leggibilità. Questo trend continuerà a formare nuove generazioni di programmatori che iniziano il loro percorso con Python.
3. **Ecosistema in crescita**:
    - L'ecosistema di Python continua a crescere con nuovi pacchetti e librerie che vengono sviluppati e mantenuti dalla comunità. Questa espansione rende Python sempre più versatile e adatto a una vasta gamma di applicazioni.
4. **Supporto aziendale**:
    - Python è adottato da molte grandi aziende per progetti mission-critical. Questo supporto aziendale garantisce un flusso continuo di investimenti nello sviluppo e nel miglioramento del linguaggio.
5. **Collaborazione globale**:
    - La natura open source di Python e la collaborazione globale della comunità assicurano che Python continui ad evolversi per soddisfare le esigenze degli sviluppatori in tutto il mondo. La partecipazione attiva di una comunità diversificata porta a un linguaggio più robusto e adattabile.

Python è nato dal desiderio di creare un linguaggio di programmazione che fosse potente e flessibile, ma anche

semplice e intuitivo. Guidato dagli obiettivi di design di leggibilità, produttività, portabilità ed estendibilità, Python ha attraversato decenni di evoluzione per diventare uno dei linguaggi di programmazione più popolari e influenti al mondo. La sua comunità attiva e la sua capacità di adattarsi alle nuove tecnologie assicurano che Python rimanga rilevante e fondamentale per lo sviluppo del software nel futuro. Con la sua ampia adozione in diversi settori e applicazioni, Python continua a essere uno strumento indispensabile per sviluppatori e aziende di tutto il mondo.

1.2 PERCHÉ SCEGLIERE PYTHON?

Python è uno dei linguaggi di programmazione più amati e utilizzati al mondo, e ci sono molte ragioni per cui sviluppatori, aziende e istituti educativi scelgono di adottarlo. In questo sotto-capitolo, esploreremo i principali vantaggi di Python e perché è diventato una scelta così popolare in una vasta gamma di campi.

Facilità di apprendimento e leggibilità

Uno dei principali punti di forza di Python è la sua facilità di apprendimento. La sintassi semplice e leggibile di Python permette ai principianti di imparare rapidamente i concetti di base della programmazione. A differenza di molti altri linguaggi, Python utilizza un'indentazione chiara per delimitare i blocchi di codice, rendendo il codice più leggibile e meno suscettibile a errori.

- **Sintassi semplice**: Python elimina molte delle complessità sintattiche presenti in altri linguaggi di programmazione. Ad esempio, non richiede la dichiarazione esplicita dei tipi di variabili, permettendo di scrivere codice in modo più intuitivo. La mancanza di simboli complessi e la riduzione del boilerplate rendono il codice Python

più simile a pseudocodice, facilitando la comprensione anche per i non programmatori.
- **Leggibilità del codice**: Python è progettato per essere leggibile come il linguaggio naturale. Le istruzioni in Python sono spesso più brevi e comprensibili rispetto a quelle in altri linguaggi, facilitando la collaborazione e la manutenzione del codice. La leggibilità del codice è ulteriormente migliorata dall'uso dell'indentazione per definire i blocchi di codice, riducendo la necessità di parentesi graffe e migliorando la struttura visiva del programma.
- **Supporto della comunità**: La vasta comunità di Python offre una grande quantità di risorse educative, dai tutorial online ai corsi strutturati, facilitando ulteriormente il processo di apprendimento per i nuovi arrivati. Siti come Python.org, Stack Overflow, e molte altre piattaforme forniscono una vasta gamma di risorse che coprono ogni aspetto del linguaggio, dalle basi agli argomenti avanzati.

Versatilità e ampiezza di applicazioni

Python è un linguaggio estremamente versatile, utilizzato in una vasta gamma di applicazioni. La sua flessibilità permette agli sviluppatori di utilizzarlo per tutto, dalla creazione di script semplici alla costruzione di applicazioni web complesse, dall'analisi dei dati al machine learning.

- **Sviluppo web**: Python è ampiamente utilizzato per lo sviluppo web grazie a framework potenti come Django e Flask. Questi strumenti permettono di costruire applicazioni web robuste e scalabili in modo rapido ed efficiente. Django, in particolare, fornisce una soluzione completa con un'ampia

gamma di funzionalità integrate, mentre Flask offre una maggiore flessibilità per chi preferisce un approccio più modulare.

- **Data science**: Python è diventato lo standard de facto per la data science, grazie a librerie come Pandas, NumPy, Matplotlib e Scikit-learn. Questi strumenti offrono funzionalità avanzate per la manipolazione, l'analisi e la visualizzazione dei dati. Pandas, ad esempio, permette di lavorare facilmente con grandi set di dati, mentre Matplotlib offre potenti strumenti di visualizzazione per creare grafici e plot complessi.
- **Machine learning e AI**: Python è il linguaggio preferito per lo sviluppo di applicazioni di machine learning e intelligenza artificiale. Librerie come TensorFlow, Keras e PyTorch forniscono potenti strumenti per la creazione di modelli di machine learning. Questi strumenti consentono agli sviluppatori di costruire, addestrare e valutare modelli complessi con poche righe di codice, accelerando il processo di sviluppo.
- **Automazione**: Python è spesso utilizzato per automatizzare compiti ripetitivi, come il web scraping e la gestione delle configurazioni di sistema. Librerie come Selenium e BeautifulSoup permettono di automatizzare il web scraping, mentre strumenti come Ansible utilizzano Python per l'automazione della configurazione. Questo rende Python una scelta eccellente per amministratori di sistema e sviluppatori DevOps.
- **Sviluppo di giochi e multimedia**: Python può essere utilizzato anche per lo sviluppo di giochi e applicazioni multimediali. Framework come Pygame permettono di sviluppare giochi 2D, fornendo strumenti per la gestione di grafica, suoni e interazioni utente. Questo rende Python una scelta

popolare anche tra i game developer e gli sviluppatori di applicazioni interattive.

Forte supporto per l'integrazione e la scalabilità

Python è noto per la sua capacità di integrarsi facilmente con altri linguaggi e tecnologie, rendendolo una scelta eccellente per progetti che richiedono interoperabilità e scalabilità.

- **Integrazione con altri linguaggi**: Python può essere facilmente integrato con linguaggi come C, C++ e Java, permettendo agli sviluppatori di sfruttare i punti di forza di diversi linguaggi in un unico progetto. Questo è particolarmente utile per applicazioni che richiedono prestazioni elevate in specifiche parti del codice. Ad esempio, è possibile scrivere le parti più critiche in C o C++ per ottimizzare le prestazioni e utilizzare Python per la logica di alto livello.
- **Scalabilità**: Python è utilizzato in numerose applicazioni su larga scala, da start-up a grandi aziende tecnologiche. Ad esempio, aziende come Google, Facebook e Instagram utilizzano Python per costruire e mantenere le loro infrastrutture scalabili. La capacità di Python di gestire grandi volumi di traffico e di scalare orizzontalmente lo rende una scelta ideale per applicazioni che devono crescere con l'aumento degli utenti.
- **Ambiente di sviluppo**: Python offre un ricco ecosistema di strumenti e ambienti di sviluppo che supportano la produttività degli sviluppatori. IDE come PyCharm, Visual Studio Code e Jupyter Notebook forniscono funzionalità avanzate per la scrittura, il debugging e l'esecuzione del codice Python. Jupyter Notebook, in particolare, è

ampiamente utilizzato nel campo della data science per la sua capacità di combinare codice eseguibile, visualizzazioni e note in un unico documento interattivo.

Comunità attiva e risorse abbondanti

La comunità Python è una delle più grandi e attive nel mondo della programmazione. Questo offre numerosi vantaggi, tra cui un ampio supporto, risorse educative abbondanti e continui miglioramenti al linguaggio.

- **Supporto della comunità**: La comunità Python è nota per essere accogliente e collaborativa. Esistono numerosi forum, gruppi di discussione e canali di chat dove gli sviluppatori possono ottenere aiuto e condividere conoscenze. Piattaforme come Stack Overflow e Reddit offrono un supporto rapido e competente per risolvere problemi e discutere best practice.
- **Documentazione e tutorial**: Python offre una documentazione eccellente e una vasta gamma di tutorial online, corsi e libri che coprono ogni aspetto del linguaggio. Questo rende più facile per i nuovi sviluppatori imparare Python e per gli esperti approfondire argomenti avanzati. La documentazione ufficiale di Python è considerata un modello per la sua completezza e chiarezza, fornendo esempi pratici e dettagliati per ogni funzione e libreria standard.
- **Conferenze e eventi**: La comunità Python organizza numerose conferenze ed eventi in tutto il mondo, come PyCon, che offrono opportunità di networking e apprendimento per gli sviluppatori di tutti i livelli. Questi eventi sono un'ottima occasione per incontrare altri sviluppatori, scoprire le ultime novità

nel mondo Python e partecipare a workshop e sessioni di formazione.

Sostenibilità e futuro promettente

Python ha dimostrato di essere un linguaggio sostenibile e con un futuro promettente. La sua continua crescita e adozione in vari settori garantisce che Python rimarrà una scelta rilevante per molti anni a venire.

- **Crescita continua**: La popolarità di Python continua a crescere, con un numero sempre maggiore di sviluppatori e aziende che scelgono di adottarlo. Questo assicura che Python rimarrà un linguaggio vitale e in evoluzione. Secondo vari sondaggi e rapporti di settore, Python è costantemente classificato tra i linguaggi di programmazione più popolari al mondo.
- **Innovazione costante**: La comunità Python e la Python Software Foundation lavorano costantemente per migliorare il linguaggio e introdurre nuove funzionalità. Questo impegno verso l'innovazione assicura che Python rimanga all'avanguardia nel campo della programmazione. Le nuove versioni di Python continuano a introdurre miglioramenti significativi in termini di prestazioni, sicurezza e funzionalità.
- **Adozione in nuovi settori**: Python continua a trovare nuovi ambiti di applicazione, dall'Internet delle cose (IoT) alla robotica, garantendo che il linguaggio continui a evolversi e adattarsi alle nuove esigenze tecnologiche. La sua flessibilità e la vasta gamma di librerie disponibili lo rendono adatto a una varietà di progetti, dai più semplici ai più complessi.

Python nel mondo dell'istruzione

Python è diventato il linguaggio di programmazione preferito per l'istruzione in molte scuole e università. La sua sintassi semplice e la facilità di apprendimento lo rendono ideale per insegnare i concetti di base della programmazione.

- **Introdurre i concetti di programmazione**: Python è spesso utilizzato come primo linguaggio di programmazione nei corsi introduttivi di informatica. La sua sintassi chiara e l'approccio intuitivo aiutano gli studenti a concentrarsi sui concetti fondamentali senza essere distratti da dettagli sintattici complessi.
- **Strumenti educativi**: Esistono numerosi strumenti e piattaforme educative basate su Python che aiutano gli studenti a imparare a programmare. Ad esempio, piattaforme come Codecademy, edX e Coursera offrono corsi di Python interattivi e guidati che coprono una vasta gamma di argomenti, dalla programmazione di base alla data science avanzata.
- **Progetti pratici**: Python è ideale per progetti pratici e applicazioni reali, permettendo agli studenti di applicare ciò che hanno imparato in modo concreto. Ad esempio, gli studenti possono creare piccoli giochi, sviluppare applicazioni web o analizzare dati reali utilizzando Python e le sue librerie.

Python nell'industria

Python è ampiamente utilizzato in molte industrie, grazie alla sua versatilità e alla vasta gamma di librerie disponibili.

- **Tecnologia e software**: Grandi aziende tecnologiche come Google, Facebook, Amazon e Netflix utilizzano Python per una varietà di applicazioni, dalle

infrastrutture backend ai sistemi di machine learning. Python è utilizzato per lo sviluppo di servizi web, l'automazione di processi e l'analisi dei dati.
- **Finanza**: Python è utilizzato nel settore finanziario per lo sviluppo di algoritmi di trading, l'analisi dei dati finanziari e la gestione dei portafogli. La sua capacità di manipolare grandi volumi di dati e la disponibilità di librerie specializzate lo rendono una scelta ideale per applicazioni finanziarie.
- **Sanità**: Python è utilizzato nel settore sanitario per l'analisi dei dati medici, lo sviluppo di applicazioni di telemedicina e la ricerca bioinformatica. Le sue potenti librerie di data science permettono di analizzare grandi dataset medici e di sviluppare modelli predittivi per la diagnosi e il trattamento delle malattie.
- **Media e intrattenimento**: Python è utilizzato per lo sviluppo di applicazioni multimediali, giochi e contenuti digitali. Ad esempio, molte case di produzione cinematografica utilizzano Python per la gestione delle pipeline di produzione e per lo sviluppo di strumenti di effetti speciali.

Python per l'innovazione e la ricerca

Python è uno strumento potente per l'innovazione e la ricerca, grazie alla sua flessibilità e alla vasta gamma di librerie scientifiche.

- **Ricerca accademica**: Python è ampiamente utilizzato nella ricerca accademica per l'analisi dei dati, la modellazione e la simulazione. La disponibilità di librerie come SciPy, NumPy e Matplotlib permette ai ricercatori di eseguire calcoli complessi e di visualizzare i risultati in modo efficace.

- **Intelligenza artificiale**: Python è il linguaggio preferito per la ricerca nel campo dell'intelligenza artificiale, grazie a librerie come TensorFlow, Keras e PyTorch. Questi strumenti permettono ai ricercatori di sviluppare e testare nuovi algoritmi di machine learning in modo rapido ed efficiente.
- **Scienze dei materiali**: Python è utilizzato nella scienza dei materiali per la modellazione e la simulazione di nuovi materiali. Le sue librerie scientifiche permettono di eseguire calcoli complessi e di analizzare i dati sperimentali per sviluppare nuovi materiali con proprietà desiderate.

Python è diventato uno dei linguaggi di programmazione più popolari e versatili grazie alla sua semplicità, leggibilità e vasto ecosistema di librerie e strumenti. La sua facilità di apprendimento lo rende ideale per i principianti, mentre la sua potenza e flessibilità lo rendono una scelta eccellente per sviluppatori esperti. La comunità Python continua a crescere e a innovare, garantendo che Python rimanga un linguaggio di programmazione rilevante e influente per molti anni a venire. Che tu sia uno sviluppatore, un data scientist, un ricercatore o un educatore, Python offre gli strumenti e le risorse necessarie per raggiungere i tuoi obiettivi.

1.3 INSTALLAZIONE E CONFIGURAZIONE DELL' AMBIENTE DI SVILUPPO

L'installazione e la configurazione di Python e dell'ambiente di sviluppo sono i primi passi fondamentali per iniziare a programmare con questo linguaggio. In questo sotto-capitolo, esploreremo come installare Python su diversi sistemi operativi, configurare un ambiente di sviluppo integrato (IDE) e impostare un ambiente virtuale per la gestione delle dipendenze.

Installazione di Python

Python è disponibile per una vasta gamma di sistemi operativi, tra cui Windows, macOS e Linux. Di seguito sono riportate le istruzioni dettagliate per l'installazione di Python su ciascuno di questi sistemi.

Windows

1. **Scaricare il programma di installazione**:
   - Visitate il sito ufficiale di Python python.org e scaricate l'ultima versione stabile del programma di installazione per Windows.
2. **Eseguire il programma di installazione**:
   - Aprite il file scaricato e seguite le istruzioni visualizzate. Assicuratevi di selezionare l'opzione

"Add Python to PATH" per rendere Python accessibile da qualsiasi terminale. Questa opzione aggiunge il percorso di Python alle variabili d'ambiente del sistema, permettendovi di eseguire Python direttamente dal prompt dei comandi.

3. **Verificare l'installazione**:
   - Aprite il prompt dei comandi e digitate `python --version`. Dovrebbe comparire la versione di Python installata. Questo conferma che Python è stato installato correttamente e che il percorso è stato aggiunto alle variabili d'ambiente.

macOS

1. **Scaricare il programma di installazione**:
   - Visitate il sito ufficiale di Python e scaricate l'ultima versione stabile del programma di installazione per macOS.
2. **Eseguire il programma di installazione**:
   - Aprite il file scaricato e seguite le istruzioni visualizzate per completare l'installazione. Questo installerà Python 3 insieme al gestore di pacchetti `pip`.
3. **Verificare l'installazione**:
   - Aprite il terminale e digitate `python3 --version`. Dovrebbe comparire la versione di Python installata. Su macOS, `python` di solito si riferisce alla versione preinstallata di Python 2, quindi è importante utilizzare `python3`.

Linux

1. **Usare il gestore di pacchetti**:

- La maggior parte delle distribuzioni Linux include Python nel loro gestore di pacchetti. Per installare Python, aprite il terminale e utilizzate il comando appropriato per la vostra distribuzione:
  - Debian/Ubuntu: `sudo apt-get update && sudo apt-get install python3`
  - Fedora: `sudo dnf install python3`
  - Arch Linux: `sudo pacman -S python`

2. **Verificare l'installazione**:
   - Dopo l'installazione, verificate la versione di Python digitando `python3 --version` nel terminale. Questo confermerà che Python è stato installato correttamente.

Configurazione dell'ambiente di sviluppo

Una volta installato Python, il passo successivo è configurare un ambiente di sviluppo integrato (IDE) che faciliti la scrittura, il debugging e l'esecuzione del codice. Esistono numerosi IDE tra cui scegliere, ognuno con caratteristiche uniche. Di seguito sono riportate alcune delle opzioni più popolari.

PyCharm

1. **Installazione**:
   - Visitate il sito web di JetBrains jetbrains.com/pycharm e scaricate la versione gratuita di PyCharm (Community Edition) o la versione a pagamento (Professional Edition) se desiderate funzionalità aggiuntive.
2. **Configurazione**:
   - Dopo l'installazione, aprite PyCharm e configurate il vostro primo progetto Python. PyCharm rileverà automaticamente

l'installazione di Python e configurerà l'interprete per il progetto.
3. **Caratteristiche**:
    - PyCharm offre una vasta gamma di funzionalità, tra cui il completamento automatico del codice, il refactoring, il debugging integrato, il supporto per i test e molto altro. Supporta anche l'integrazione con sistemi di controllo versione come Git, facilitando la gestione del codice sorgente.

Visual Studio Code

1. **Installazione**:
    - Visitate il sito web di Visual Studio Code code.visualstudio.com e scaricate l'ultima versione per il vostro sistema operativo.
2. **Configurazione**:
    - Dopo l'installazione, aprite Visual Studio Code e installate l'estensione Python di Microsoft. Questa estensione fornisce supporto per il completamento del codice, il debugging, il linting e altre funzionalità.
3. **Caratteristiche**:
    - Visual Studio Code è noto per la sua leggerezza e la sua flessibilità. Supporta una vasta gamma di estensioni che permettono di personalizzare l'ambiente di sviluppo secondo le proprie esigenze. È possibile configurare l'ambiente per adattarlo a vari flussi di lavoro e linguaggi di programmazione.

Jupyter Notebook

1. **Installazione**:

    - Jupyter Notebook è uno strumento molto utilizzato nella data science e nel machine learning. Potete installarlo utilizzando il gestore di pacchetti `pip`. Aprite il terminale e digitate `pip install notebook`.
2. **Configurazione**:
    - Dopo l'installazione, avviate Jupyter Notebook digitando `jupyter notebook` nel terminale. Questo aprirà un'interfaccia web dove potrete creare e gestire i vostri notebook.
3. **Caratteristiche**:
    - Jupyter Notebook permette di scrivere ed eseguire codice Python in celle interattive. È particolarmente utile per l'analisi dei dati, la visualizzazione e la creazione di documentazione interattiva. È possibile combinare codice, testo, immagini e grafici in un unico documento, facilitando la condivisione dei risultati.

Impostazione di un ambiente virtuale

Un ambiente virtuale è un'installazione isolata di Python che permette di gestire le dipendenze del progetto senza influenzare il sistema globale. Utilizzare ambienti virtuali è una pratica consigliata per evitare conflitti tra le versioni delle librerie e mantenere i progetti puliti e organizzati.

Creazione di un ambiente virtuale

1. **Installare virtualenv**:
    - Se non lo avete già, potete installare `virtualenv` utilizzando `pip`. Aprite il terminale e digitate `pip install virtualenv`.
2. **Creare un nuovo ambiente virtuale**:

    - Navigate alla directory del vostro progetto e create un nuovo ambiente virtuale utilizzando il comando `virtualenv venv`. Questo creerà una directory chiamata `venv` che conterrà l'ambiente virtuale.
3. **Attivare l'ambiente virtuale**:
    - Windows: `.venvScriptsactivate`
    - macOS/Linux: `source venv/bin/activate`
4. **Installare le dipendenze**:
    - Con l'ambiente virtuale attivato, potete installare le dipendenze del progetto utilizzando `pip`. Ad esempio, per installare la libreria `requests`, digitate `pip install requests`.

Gestione delle dipendenze

1. **requirements.txt**:
    - È una buona pratica creare un file `requirements.txt` che elenchi tutte le dipendenze del progetto. Potete generare questo file utilizzando il comando `pip freeze > requirements.txt`. Per installare le dipendenze da un file `requirements.txt`, utilizzate `pip install -r requirements.txt`.
2. **Aggiornamento delle dipendenze**:
    - È importante mantenere aggiornate le dipendenze del progetto. Potete verificare se ci sono aggiornamenti disponibili utilizzando strumenti come `pip list --outdated` e aggiornare le librerie con `pip install --upgrade <nome_libreria>`.

Utilizzo di Conda per la gestione degli ambienti

Oltre a `virtualenv`, un altro strumento popolare per la gestione degli ambienti virtuali è `Conda`. Conda è un

gestore di pacchetti e ambienti che permette di gestire librerie e dipendenze in modo efficiente.

Installazione di Conda

1. **Scaricare Anaconda o Miniconda**:
   - Anaconda include una vasta gamma di librerie scientifiche preinstallate, mentre Miniconda è una versione più leggera che include solo Conda e Python. Potete scaricare Anaconda o Miniconda dal sito ufficiale anaconda.com.
2. **Installazione**:
   - Seguite le istruzioni fornite per il vostro sistema operativo. Dopo l'installazione, Conda sarà disponibile dal terminale.

Creazione e gestione degli ambienti con Conda

1. **Creare un nuovo ambiente:**
   - Per creare un nuovo ambiente, utilizzate il comando `conda create --name nome_ambiente`. Potete specificare le versioni delle librerie da installare utilizzando l'opzione `--name`.
2. **Attivare l'ambiente**:
   - Per attivare l'ambiente, utilizzate il comando `conda activate nome_ambiente`.
3. **Installare le dipendenze**:
   - Per installare librerie specifiche, utilizzate il comando `conda install nome_libreria`.
4. **Gestione delle dipendenze**:
   - Conda permette di esportare e importare le dipendenze degli ambienti utilizzando i file `environment.yml`. Potete creare un file `environment.yml` con il comando `conda env export > environment.yml` e ricreare l'ambiente con `conda env create -f environment.yml`.

Utilizzo di Docker per ambienti isolati

Docker è una piattaforma per la creazione, la distribuzione e l'esecuzione di applicazioni in container. Utilizzare Docker per creare ambienti di sviluppo isolati può essere molto utile per garantire la consistenza tra diversi ambienti.

Installazione di Docker

1. **Scaricare e installare Docker**:
   - Visitate il sito ufficiale di Docker docker.com e scaricate Docker Desktop per il vostro sistema operativo. Seguite le istruzioni per completare l'installazione.

Creazione di un ambiente Docker

1. **Creare un Dockerfile**:
   - Un Dockerfile è un file di testo che contiene tutte le istruzioni per creare un'immagine Docker. Ecco un esempio di Dockerfile per un'applicazione Python:

```
# Utilizza un'immagine base con Python 3


FROM python:3.9
```

```
# Imposta la directory di lavoro

WORKDIR /app

# Copia i file di dipendenze e installa le dipendenze

COPY requirements.txt .

RUN pip install --no-cache-dir -r requirements.txt

# Copia il codice sorgente

COPY . .
```

```
# Comando per eseguire l'applicazione


CMD ["python", "main.py"]
```

2. **Costruire l'immagine Docker**:
   - Utilizzate il comando `docker build -t nome_immagine .` per costruire l'immagine Docker.
3. **Eseguire il container**:
   - Utilizzate il comando `docker run -d -p 8000:8000 nome_immagine` per eseguire il container. Questo esporrà l'applicazione sulla porta 8000 del vostro host.

Installare e configurare Python, insieme all'impostazione di un ambiente di sviluppo adeguato, sono passaggi fondamentali per iniziare a programmare in modo efficiente. Indipendentemente dal sistema operativo che utilizzate, ci sono strumenti e pratiche che possono facilitare l'installazione, la configurazione e la gestione delle dipendenze. Che si tratti di creare ambienti virtuali con `virtualenv` o Conda, o di utilizzare Docker per garantire la consistenza degli ambienti di sviluppo, Python offre una vasta gamma di opzioni per

adattarsi alle vostre esigenze. Con un ambiente di sviluppo ben configurato, sarete pronti per iniziare a scrivere, testare e distribuire applicazioni Python di alta qualità.

CAPITOLO 2: SINTASSI DI BASE DI PYTHON | 2.1 VARIABILI E TIPI DI DATI

2.1 Variabili e tipi di dati

In Python, le variabili sono utilizzate per memorizzare valori. Non è necessario dichiarare il tipo di variabile in anticipo, poiché Python è un linguaggio dinamicamente tipizzato. Questo significa che il tipo della variabile viene determinato automaticamente in base al valore assegnato.

Tipi di Dati Comuni

In Python, ci sono diversi tipi di dati comuni che puoi utilizzare per memorizzare vari valori. Ecco alcuni dei tipi di dati più comuni:

Interi (int): Numeri interi senza parte decimale.

```
x = 10
```

Float: Numeri decimali con una parte frazionaria.

```
y = 3.14
```

Stringhe (str): Sequenze di caratteri utilizzate per rappresentare testo.

```
name = "Python"
```

Booleani (bool): Valori di verità che possono essere `True` o `False`.

```
is_valid = True
```

Dichiarazione delle Variabili

Dichiarare variabili in Python è semplice. Basta utilizzare il nome della variabile seguito da un segno di uguale (=) e il valore che si desidera assegnare. Ecco alcuni esempi di variabili dichiarate con diversi tipi di dati:

```
# Esempi di variabili

numero_intero = 10

numero_decimale = 3.14
```

```
testo = "Ciao, Python!"

valore_booleano = True
```

In questi esempi:

- `numero_intero` è una variabile di tipo intero (`int`).
- `numero_decimale` è una variabile di tipo float.
- `testo` è una variabile di tipo stringa (`str`).
- `valore_booleano` è una variabile di tipo booleano (`bool`).

Operazioni con le Variabili

Python permette di eseguire diverse operazioni sulle variabili. Ecco alcune operazioni comuni che puoi eseguire:

Operazioni Aritmetiche

Puoi eseguire operazioni aritmetiche sui numeri, come addizione, sottrazione, moltiplicazione e divisione.

```
a = 5

b = 2


# Addizione

somma = a + b # Risultato: 7


# Sottrazione

differenza = a - b # Risultato: 3


# Moltiplicazione

prodotto = a * b # Risultato: 10
```

```
# Divisione

quoziente = a / b # Risultato: 2.5


# Divisione intera

quoziente_intero = a // b # Risultato: 2


# Modulo (resto della divisione)

resto = a % b # Risultato: 1


# Esponenziazione

potenza = a ** b # Risultato: 25
```

Operazioni con le Stringhe

Le stringhe possono essere concatenate e ripetute utilizzando gli operatori `+` e `*`.

```
str1 = "Ciao"

str2 = "Python"


# Concatenazione

saluto = str1 + ", " + str2 + "!" # Risultato: "Ciao,
Python!"


# Ripetizione

eco = str1 * 3 # Risultato: "CiaoCiaoCiao"
```

Operazioni con i Booleani

Le operazioni logiche possono essere eseguite sui valori booleani utilizzando gli operatori `and`, `or`, e `not`.

```
vero = True

falso = False


# AND logico

e_logico = vero and falso # Risultato: False


# OR logico

o_logico = vero or falso # Risultato: True


# NOT logico

not_logico = not vero # Risultato: False
```

Conversione dei Tipi di Dati

In Python, puoi convertire una variabile da un tipo di dato a un altro utilizzando le funzioni di conversione integrate.

Conversione di Interi e Float

Puoi convertire tra interi e float utilizzando `int()` e `float()`.

```
intero = 10

decimale = 3.14

# Conversione da intero a float

intero_a_float = float(intero) # Risultato: 10.0

# Conversione da float a intero

float_a_intero = int(decimale) # Risultato: 3
```

Conversione di Numeri e Stringhe

Puoi convertire numeri in stringhe e viceversa utilizzando `str()` e `int()`/ `float()`.

```
numero = 123

testo = "456"

# Conversione da numero a stringa

numero_a_stringa = str(numero) # Risultato: "123"

# Conversione da stringa a numero

stringa_a_intero = int(testo) # Risultato: 456

stringa_a_float = float(testo) # Risultato: 456.0
```

Comprendere le variabili e i tipi di dati è fondamentale per scrivere programmi efficaci in Python. La capacità di manipolare e convertire questi tipi di dati ti permette di creare script e applicazioni flessibili e potenti. Con queste basi, sei pronto per esplorare ulteriori funzionalità di Python e sviluppare competenze più avanzate.

2.2 OPERATORI

Gli operatori sono simboli che dicono all'interprete di Python di eseguire calcoli specifici. Sono divisi in diverse categorie e vengono utilizzati per eseguire operazioni su variabili e valori.

Operatori Aritmetici

Gli operatori aritmetici sono utilizzati per eseguire operazioni matematiche comuni.

- **Addizione (`+`)**: Somma due valori.
  ```
  a = 5 b = 3 risultato = a + b # Risultato: 8
  ```
- **Sottrazione (`-`)**: Sottrae un valore dall'altro.
  ```
  risultato = a - b # Risultato: 2
  ```
- **Moltiplicazione (`*`)**: Moltiplica due valori.
  ```
  risultato = a * b # Risultato: 15
  ```
- **Divisione (`/`)**: Divide un valore per l'altro.
  ```
  risultato = a / b # Risultato: 1.666...
  ```
- **Modulo (`%`)**: Restituisce il resto della divisione.
  ```
  risultato = a % b # Risultato: 2
  ```

Operatori di Assegnazione

Gli operatori di assegnazione sono utilizzati per assegnare valori alle variabili.

- **Uguale (`=`)**: Assegna un valore a una variabile.
  ```
  a = 5
  ```
- **Incremento (`+=`)**: Incrementa una variabile di un valore specificato.
  ```
  a += 2 # Equivalente a: a = a + 2; Risultato: a = 7
  ```
- **Decremento (`-=`)**: Decrementa una variabile di un valore specificato.
  ```
  a -= 2 # Equivalente a: a = a - 2; Risultato: a = 3
  ```

Operatori di Confronto

Gli operatori di confronto sono utilizzati per confrontare due valori. Restituiscono un valore booleano (`True` O `False`).

- **Uguale (`==`)**: Verifica se due valori sono uguali.
  ```
  risultato = (a == b) # Risultato: False
  ```
- **Diverso (`!=`)**: Verifica se due valori sono diversi.
  ```
  risultato = (a != b) # Risultato: True
  ```
- **Maggiore (`>`)**: Verifica se il primo valore è maggiore del secondo.
  ```
  risultato = (a > b) # Risultato: True
  ```
- **Minore (`<`)**: Verifica se il primo valore è minore del secondo.
  ```
  risultato = (a < b) # Risultato: False
  ```
- **Maggiore o uguale (`>=`)**: Verifica se il primo valore è maggiore o uguale al secondo.
  ```
  risultato = (a >= b) # Risultato: True
  ```
- **Minore o uguale (`<=`)**: Verifica se il primo valore è minore o uguale al secondo.
  ```
  risultato = (a <= b) # Risultato: False
  ```

Operatori Logici

Gli operatori logici sono utilizzati per combinare più condizioni logiche.

- **AND (`and`)**: Restituisce `True` se entrambe le condizioni sono vere.
  `a = True b = False risultato = a and b # Risultato: False`
- **OR (`or`)**: Restituisce `True` se almeno una delle condizioni è vera.
  `risultato = a or b # Risultato: True`
- **NOT (`not`)**: Inverte il valore di una condizione.
  `risultato = not a # Risultato: False`

Esempi di Utilizzo

Di seguito alcuni esempi che illustrano l'utilizzo combinato di diversi operatori:

Esempio 1: Calcolo Aritmetico e Assegnazione

```
a = 10

b = 5


# Uso degli operatori aritmetici

somma = a + b # Risultato: 15

differenza = a - b # Risultato: 5

prodotto = a * b # Risultato: 50

quoziente = a / b # Risultato: 2.0

modulo = a % b # Risultato: 0
```

```
# Uso degli operatori di assegnazione

a += 2 # a diventa 12

b *= 3 # b diventa 15
```

Esempio 2: Confronti e Logica

```
x = 8

y = 3


# Uso degli operatori di confronto

uguali = (x == y) # Risultato: False

diversi = (x != y) # Risultato: True

x_maggiore = (x > y) # Risultato: True

x_minore_uguale = (x <= y) # Risultato: False


# Uso degli operatori logici

condizione = (x > 5) and (y < 10) # Risultato: True

altra_condizione = (x < 5) or (y < 10) # Risultato:
True

not_condizione = not (x == y) # Risultato: True
```

Gli operatori sono fondamentali per eseguire calcoli e prendere decisioni nei tuoi programmi Python. Comprendere come e quando utilizzare i vari operatori ti

aiuterà a scrivere codice più efficace e robusto. Con questi concetti di base, puoi ora passare ad esplorare strutture di controllo e funzioni per arricchire ulteriormente le tue competenze di programmazione in Python.

2.3 STRUTTURE DI CONTROLLO (IF, ELSE, ELIF)

Le strutture di controllo sono essenziali per la gestione del flusso di esecuzione di un programma. In Python, le strutture di controllo `if`, `else` e `elif` vengono utilizzate per prendere decisioni basate su condizioni logiche.

Condizione `if`

La dichiarazione `if` viene utilizzata per eseguire un blocco di codice solo se una condizione è vera.

```
x = 10 if x > 5: print( "x è maggiore di 5")
```

In questo esempio, il messaggio "x è maggiore di 5" verrà stampato perché la condizione `x > 5` è vera.

Condizione `else`

La dichiarazione `else` viene utilizzata per eseguire un blocco di codice se tutte le condizioni `if` precedenti sono false.

```
x = 3
```

```
if x > 5:

 print("x è maggiore di 5")

else:

 print("x non è maggiore di 5")
```

In questo esempio, il messaggio "x non è maggiore di 5" verrà stampato perché la condizione `x > 5` è falsa.

Condizione `elif`

La dichiarazione `elif` (abbreviazione di "else if") viene utilizzata per controllare più condizioni diverse. Se la condizione `if` originale è falsa, Python verifica la condizione `elif`. Puoi avere più dichiarazioni `elif` in una struttura di controllo.

```
x = 5


if x > 5:

 print("x è maggiore di 5")

elif x == 5:

 print("x è uguale a 5")

else:

 print("x è minore di 5")
```

In questo esempio, il messaggio "x è uguale a 5" verrà stampato perché la condizione `x == 5` è vera.

Esempi di Utilizzo Esempio 1: Controllo dei Voti

Un programma che determina il voto di uno studente in base al punteggio:

```
punteggio = 85


if punteggio >= 90:
 voto = "A"
elif punteggio >= 80:
 voto = "B"
elif punteggio >= 70:
 voto = "C"
elif punteggio >= 60:
 voto = "D"
else:
 voto = "F"


print(f"Il voto dello studente è: {voto}")
```

Esempio 2: Verifica Pari o Dispari

Un programma che verifica se un numero è pari o dispari:

```
numero = 7
```

```
if numero % 2 == 0:
 print(f"{numero} è un numero pari")
else:
 print(f"{numero} è un numero dispari")
```

Nidificazione delle Condizioni

È possibile nidificare le strutture di controllo, cioè mettere una dichiarazione `if` all'interno di un'altra. Tuttavia, è importante mantenere il codice leggibile.

```
x = 10
y = 20

if x > 5:
 if y > 15:
 print("x è maggiore di 5 e y è maggiore di 15")
 else:
 print("x è maggiore di 5 ma y non è maggiore di 15")
else:
 print("x non è maggiore di 5")
```

In questo esempio, il messaggio "x è maggiore di 5 e y è maggiore di 15" verrà stampato.

Utilizzo di `if` in una Singola Linea

In Python, puoi scrivere una dichiarazione `if` su una singola linea per espressioni semplici. Questo è noto come condizione ternaria.

```
  x = 10 messaggio = "x è maggiore di 5" if x > 5 else "x non è maggiore di 5"
  print(messaggio)
```

In questo esempio, il messaggio "x è maggiore di 5" verrà stampato.

Le strutture di controllo `if`, `else` e `elif` sono fondamentali per prendere decisioni nei programmi Python. Comprendere come utilizzarle efficacemente ti permetterà di scrivere codice più complesso e condizionale, migliorando la logica e la funzionalità dei tuoi programmi. Con queste basi, sei pronto a esplorare altre strutture di controllo come i cicli e le funzioni per arricchire ulteriormente le tue competenze di programmazione in Python.

CAPITOLO 3: STRUTTURE DATI IN PYTHON

3.1 Liste

Le liste sono una delle strutture dati più versatili e utilizzate in Python. Una lista è una raccolta ordinata e mutabile di elementi, che possono essere di tipi diversi.

Creazione di una lista:

Puoi creare una lista usando parentesi quadre `[]` e separando gli elementi con virgole.

```
# Creazione di una lista vuota

lista_vuota = []

# Lista di numeri interi

numeri = [1, 2, 3, 4, 5]

# Lista di stringhe

frutti = ["mela", "banana", "ciliegia"]

# Lista mista
```

```
mista = [1, "ciao", 3.14, True]
```

Accesso agli elementi della lista: Gli elementi di una lista possono essere accessi usando gli indici, che iniziano da 0.

```
# Accesso al primo elemento

primo_elemento = frutti[0] # "mela"


# Accesso all'ultimo elemento

ultimo_elemento = frutti[-1] # "ciliegia"
```

Modifica degli elementi della lista: Puoi modificare un elemento di una lista assegnando un nuovo valore all'indice corrispondente.

```
frutti[1] = "arancia"

print(frutti) # ["mela", "arancia", "ciliegia"]
```

Aggiungere e rimuovere elementi: Python fornisce diversi metodi per aggiungere e rimuovere elementi da una lista.

- **append():** Aggiunge un elemento alla fine della lista.

```
frutti.append("kiwi")

print(frutti) # ["mela", "arancia", "ciliegia", "kiwi"]
```

- **insert():** Inserisce un elemento in una posizione specifica.

```
frutti.insert(1, "mango")

print(frutti) # ["mela", "mango", "arancia", "ciliegia",
"kiwi"]
```

- **remove():** Rimuove il primo elemento uguale al valore specificato.

```
frutti.remove("arancia")

print(frutti) # ["mela", "mango", "ciliegia", "kiwi"]
```

- **pop():** Rimuove e restituisce l'elemento in una posizione specifica. Se non viene specificato alcun

indice, rimuove l'ultimo elemento.

```
ultimo = frutti.pop()

print(ultimo) # "kiwi"

print(frutti) # ["mela", "mango", "ciliegia"]
```

Metodi comuni delle liste: Ecco alcuni metodi comunemente usati con le liste:

- **len():** Restituisce la lunghezza della lista.

```
lunghezza = len(frutti) # 3
```

- **sort():** Ordina gli elementi della lista in ordine crescente.

```
numeri = [3, 1, 4, 1, 5, 9]

numeri.sort()

print(numeri) # [1, 1, 3, 4, 5, 9]
```

- **reverse():** Inverte l'ordine degli elementi nella lista.

```
numeri.reverse()

print(numeri) # [9, 5, 4, 3, 1, 1]
```

- **index():** Restituisce l'indice del primo elemento uguale al valore specificato.

```
indice = frutti.index("mango") # 1
```

- **count():** Restituisce il numero di volte che un elemento appare nella lista.

```
conteggio = numeri.count(1) # 2
```

- **clear():** Rimuove tutti gli elementi dalla lista.

```
frutti.clear()

print(frutti) # []
```

Le liste in Python sono strumenti potenti e flessibili che ti permettono di gestire e manipolare raccolte di dati in modo efficiente. Utilizzando i metodi e le tecniche discussi sopra, puoi eseguire una vasta gamma di operazioni sulle liste.

3.2 TUPLE

Le tuple in Python sono simili alle liste, ma con una differenza fondamentale: le tuple sono immutabili, il che significa che una volta create, non possono essere modificate. Questa caratteristica rende le tuple utili per memorizzare dati che non devono essere cambiati nel tempo.

Caratteristiche Principali delle Tuple

1. **Immutabilità**: Una volta creata una tupla, non è possibile modificarne gli elementi.
2. **Ordinamento**: Gli elementi all'interno di una tupla mantengono il loro ordine.
3. **Indicizzazione**: Gli elementi di una tupla possono essere accessibili tramite l'indice, partendo da zero.
4. **Possibilità di Contenere Tipi Diversi**: Una tupla può contenere elementi di diversi tipi di dati.

Creazione di una Tupla

Le tuple possono essere create racchiudendo gli elementi tra parentesi tonde `()`:

```
tupla = ( 1, 2, 3)
tupla_mista = ( 1, "Python", 3.14)
```

Accesso agli Elementi

È possibile accedere agli elementi di una tupla utilizzando l'indice:

```
prima_voce =tupla[ 0] # Restituisce 1
```

Operazioni con le Tuple

Nonostante l'immutabilità, è possibile eseguire alcune operazioni con le tuple:

Concatenazione

È possibile creare una nuova tupla concatenando due o più tuple:

```
tupla1 = ( 1, 2, 3)
tupla2 = ( 4, 5, 6)
tupla_concatenata = tupla1 + tupla2 # Restituisce (1, 2, 3, 4, 5, 6)
```

Ripetizione

È possibile creare una nuova tupla ripetendo gli elementi di una tupla esistente:

```
tupla_ripetuta = tupla1 * 3 # Restituisce (1, 2, 3, 1, 2, 3, 1, 2, 3)
```

Slicing

È possibile ottenere una porzione di una tupla utilizzando l'operatore di slicing:

```
sottotupla =tupla1[ 1: 3] # Restituisce (2, 3)
```

Appartenenza

È possibile verificare se un elemento è presente in una tupla utilizzando l'operatore `in`:

```
esiste = 2 in tupla1 # Restituisce True
```

Metodi delle Tuple

Le tuple hanno solo due metodi integrati:

- **count()**: Restituisce il numero di volte in cui un elemento appare nella tupla.
- **index()**: Restituisce l'indice della prima occorrenza di un elemento.

```
tupla = ( 1, 2, 3, 2, 4, 2)
# Utilizzo di count() c
onteggio =tupla.count( 2) # Restituisce 3
# Utilizzo di index()
indice =tupla.index( 2) # Restituisce 1 (la prima occorrenza di 2)
```

Vantaggi delle Tuple

- **Immutabilità**: Essendo immutabili, le tuple sono più sicure da usare in contesti in cui i dati non devono cambiare.
- **Performance**: Le tuple possono essere più efficienti in termini di memoria e tempo di accesso rispetto alle liste, grazie alla loro immutabilità.
- **Chiavi nei Dizionari**: Le tuple possono essere utilizzate come chiavi nei dizionari, mentre le liste no.

Le tuple sono un tipo di dato fondamentale in Python, utile per memorizzare sequenze di valori immutabili. La loro immutabilità le rende adatte per scenari in cui i dati devono rimanere costanti, migliorando così la sicurezza e

la performance del programma. Comprendere come creare, accedere e utilizzare le tuple ti permetterà di sfruttare al meglio questo tipo di dato nella programmazione in Python.

3.3 DIZIONARI

I dizionari sono strutture dati che permettono di memorizzare valori associati a chiavi. Ogni elemento in un dizionario è una coppia chiave-valore, dove la chiave deve essere unica e immutabile (come una stringa, un numero o una tupla), mentre i valori possono essere di qualsiasi tipo.

Caratteristiche principali dei dizionari:

- **Associazioni chiave-valore**: Ogni elemento è una coppia formata da una chiave e un valore.
- **Non ordinati**: I dizionari non mantengono l'ordine degli elementi (fino a Python 3.7; da Python 3.7 in poi, l'ordine di inserimento è mantenuto ma non garantito come parte dell'interfaccia pubblica).
- **Mutabili**: I dizionari possono essere modificati dopo la loro creazione.

Creazione di un dizionario

I dizionari possono essere creati utilizzando le parentesi graffe `{}` e separando le chiavi dai valori con i due punti :

dizionario = {"nome": "Alice", "età": 25, "città": "Roma"}

Accesso ai valori È possibile accedere ai valori di un dizionario utilizzando le chiavi corrispondenti:

nome = dizionario["nome"] # Restituisce "Alice"

Operazioni con i dizionari

- **Aggiungere o modificare un elemento:** Basta assegnare un valore a una chiave.

```
dizionario["età"] = 26 # Modifica il valore associato alla chiave
"età"

dizionario["email"] = "alice@example.com" # Aggiunge una nuova
coppia chiave-valore
```

- **Rimuovere un elemento**: Utilizzando il metodo `del` o `pop`.

```
del dizionario["città"]

email = dizionario.pop("email")
```

- **Iterare su un dizionario**: È possibile iterare sia sulle chiavi che sui valori.

```
dizionario = {"name": "Alice", "age": 25, "city": "Rome"}

# Aumenta l'età di ogni persona di 1
for key in dizionario:
    if key == "age":
        dizionario[key] += 1
```

```
print(dizionario)
```

3.4 SET

I set sono collezioni non ordinate di elementi unici. Questo significa che un set non può contenere elementi duplicati e non mantiene l'ordine di inserimento degli elementi.

Caratteristiche principali dei set:

- **Unicità**: Ogni elemento può apparire una sola volta.
- **Non ordinati**: Gli elementi non hanno un ordine definito.
- **Mutabili**: È possibile aggiungere o rimuovere elementi dopo la creazione del set.

Creazione di un set I set possono essere creati utilizzando le parentesi graffe `{}` o la funzione `set()`

```
  set_di_numeri = {1, 2, 3, 4}
set_vuoto = set()
```

Operazioni con i set

- **Aggiungere un elemento**: Utilizzando il metodo `add`

```
set_di_numeri.add(5)
```

- **Rimuovere un elemento**: Utilizzando il metodo `remove` o `discard`

```
  set_di_numeri.remove(1)
set_di_numeri.discard(2)
```

- **Operazioni insiemistiche**: I set supportano varie operazioni matematiche come unione, intersezione e differenza.

```
set_a = {1, 2, 3}

set_b = {3, 4, 5}



unione = set_a | set_b # Restituisce {1, 2, 3, 4, 5}

intersezione = set_a & set_b # Restituisce {3}

differenza = set_a - set_b # Restituisce {1, 2}
```

Queste sono le principali strutture dati in Python: liste, tuple, dizionari e set. Ogni struttura ha le sue peculiarità e il suo utilizzo ideale a seconda del tipo di dati e delle operazioni che si desidera effettuare. Conoscere e capire queste strutture è fondamentale per scrivere codice efficiente e mantenibile in Python.

CAPITOLO 4: FUNZIONI E MODULI

4.1 Definizione e utilizzo delle funzioni

Le funzioni sono uno degli elementi fondamentali della programmazione in Python. Una funzione è un blocco di codice riutilizzabile che esegue una specifica operazione. Le funzioni aiutano a strutturare il codice in modo modulare, rendendolo più leggibile, manutenibile e riutilizzabile.

Definizione di una funzione In Python, una funzione si definisce utilizzando la parola chiave `def` seguita dal nome della funzione e da parentesi tonde che possono contenere parametri. Il corpo della funzione è indentato.

```
def saluta():

 print("Ciao, mondo!")
```

Questa funzione, chiamata `saluta`, quando invocata, stamperà "Ciao, mondo!".

Utilizzo di una funzione Per utilizzare una funzione, basta chiamarla con il suo nome seguito da parentesi tonde.

```
saluta() # Output: Ciao, mondo!
```

Funzioni con parametri Le funzioni possono accettare parametri, che permettono di passare informazioni alla funzione quando viene chiamata.

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


saluta("Alice") # Output: Ciao, Alice!
saluta("Bob") # Output: Ciao, Bob!
```

In questo esempio, la funzione `saluta` accetta un parametro `nome` che viene utilizzato per personalizzare il messaggio di saluto.

Funzioni con valore di ritorno Le funzioni possono anche restituire un valore utilizzando la parola chiave `return`.

```
def somma(a, b):
 return a + b


risultato = somma(3, 4)
print(risultato) # Output: 7
```

Qui, la funzione `somma` accetta due parametri, `a` e `b`, e restituisce la loro somma.

4.2 PARAMETRI E ARGOMENTI

I parametri e gli argomenti sono un aspetto cruciale delle funzioni in Python. I parametri sono le variabili elencate tra le parentesi nella definizione della funzione. Gli argomenti sono i valori effettivi passati alla funzione quando viene chiamata.

Tipi di parametri

- **Parametri Posizionali:** Sono i più comuni e vengono associati agli argomenti in base alla loro posizione.

```
def moltiplica(a, b):

 return a * b


risultato = moltiplica(5, 3) # a=5, b=3
```

- **Parametri con valore predefinito**: Permettono di specificare un valore di default nel caso in cui non venga fornito un argomento.

```
def saluta(nome="ospite"):
```

```
 print(f"Ciao, {nome}!")


saluta()        # Output: Ciao, ospite!
saluta("Maria") # Output: Ciao, Maria!
```

- **Parametri con nome (keyword arguments)**: Permettono di passare argomenti utilizzando il nome del parametro, indipendentemente dall'ordine.

```
def sottrai(a, b):
 return a - b


risultato = sottrai(b=10, a=5) # a=5, b=10
```

- **Parametri variabili**: Permettono di passare un numero variabile di argomenti utilizzando `*args` per parametri posizionali e `**kwargs` per parametri con nome.

```
def somma(*args):
 return sum(args)
```

```
risultato = somma(1, 2, 3, 4) # Output: 10


def stampa_parametri(**kwargs):

 for chiave, valore in kwargs.items():

 print(f"{chiave}: {valore}")


stampa_parametri(nome="Alice", età=30) # Output: nome: Alice,
età:
30
```

4.3 MODULI E PACCHETTI

Python supporta la modularità attraverso l'uso di moduli e pacchetti. Un modulo è un singolo file contenente definizioni e istruzioni Python, mentre un pacchetto è una raccolta di moduli organizzati in una directory con un file speciale `__init__.py`.

Importare moduli I moduli possono essere importati nel proprio codice utilizzando la parola chiave `import`.

```
import math

print(math.sqrt(16)) # Output: 4.0
```

È possibile importare solo specifiche parti di un modulo utilizzando la sintassi `from ... import ...`

```
from math import pi

print(pi) # Output: 3.141592653589793
```

Creazione di un modulo Per creare un modulo, basta salvare il proprio codice Python in un file con estensione

`.py`. Ad esempio, se si crea un file `mio_modulo.py` con il seguente contenuto:

```
def saluta():
  print("Ciao dal modulo!")
```

Si può poi importare e utilizzare questo modulo in un altro file:

```
import mio_modulo

mio_modulo.saluta() # Output: Ciao dal modulo!
```

Creazione di un pacchetto Un pacchetto è una directory che contiene uno o più moduli. La presenza di un file `__init__.py` all'interno della directory indica a Python che questa è una directory di pacchetti.

Supponiamo di avere la seguente struttura di directory:

```
mio_pacchetto/
 __init__.py
 modulo1.py
```

```
 modulo2.py
```

- `__init__.py` potrebbe essere vuoto o contenere codice di inizializzazione per il pacchetto.
- `modulo1.py` e `modulo2.py` contengono i moduli effettivi.

Per utilizzare i moduli in un pacchetto, si usa la sintassi `import nome_pacchetto.nome_modulo`.

```
from mio_pacchetto import modulo1, modulo2


modulo1.funzione1()

modulo2.funzione2()
```

Importazioni relative All'interno di un pacchetto, si possono fare importazioni relative utilizzando i punti. Un singolo punto `.` si riferisce alla directory corrente, due punti `..` si riferiscono alla directory padre, e così via.

```
# Dentro modulo1.py

from . import modulo2
```

Utilizzo di pacchetti esterni Python ha una vasta gamma di pacchetti esterni disponibili attraverso il Python Package Index (PyPI). Il gestore di pacchetti standard per Python è `pip`, che permette di installare e gestire pacchetti esterni.

Installazione di un pacchetto Per installare un pacchetto utilizzando `pip`, si utilizza il comando:

```
pip install nome_pacchetto
```

Ad esempio, per installare il pacchetto `requests` per fare richieste HTTP:

```
pip install requests
```

Utilizzo di un pacchetto esterno Dopo aver installato un pacchetto, è possibile importarlo nel proprio codice come qualsiasi altro modulo.

```
import requests


response = requests.get("https://api.example.com/data")

print(response.json())
```

Con una comprensione approfondita delle funzioni e dei moduli, si è in grado di scrivere codice Python più

modulare, riutilizzabile ed efficiente. La capacità di definire funzioni per operazioni specifiche e di organizzare il proprio codice in moduli e pacchetti permette di gestire progetti di qualsiasi dimensione con maggiore facilità e mantenibilità.

CAPITOLO 5: PROGRAMMAZIONE ORIENTATA AGLI OGGETTI (OOP)

5.1 Classi e Oggetti

La Programmazione Orientata agli Oggetti (OOP) è un paradigma di programmazione che utilizza "oggetti" e "classi" per creare modelli di dati complessi. Python supporta pienamente l'OOP, permettendo di creare e gestire oggetti e di definire classi per strutturare il codice in modo più naturale e intuitivo.

Classi Una classe è un modello per creare oggetti. Definisce attributi (proprietà) e metodi (funzioni) che gli oggetti creati da essa avranno.

```
class Persona:

 def __init__(self, nome, età):

 self.nome = nome

 self.età = età



 def saluta(self):
```

```
 print(f"Ciao, mi chiamo {self.nome} e ho {self.età} anni.")
```

In questo esempio, `Persona` è una classe con un metodo speciale `__init__` (il costruttore) che inizializza gli attributi `nome` ed `età`. La classe ha anche un metodo `saluta` che stampa un saluto.

Oggetti Un oggetto è un'istanza di una classe. Creare un oggetto significa istanziare una classe.

```
persona1 = Persona("Alice", 30)

persona2 = Persona("Bob", 25)


persona1.saluta() # Output: Ciao, mi chiamo Alice e ho 30 anni.

persona2.saluta() # Output: Ciao, mi chiamo Bob e ho 25 anni.
```

Ogni oggetto ha i propri valori per gli attributi definiti nella classe.

5.2 EREDITARIETÀ

L'ereditarietà è un meccanismo fondamentale dell'OOP che permette di creare una nuova classe basata su una classe esistente. La classe derivata (sottoclasse) eredita attributi e metodi dalla classe base (superclasse), permettendo il riutilizzo del codice.

```
class Studente(Persona):

 def __init__(self, nome, età, matricola):

 super().__init__(nome, età)

 self.matricola = matricola



 def studia(self):

 print(f"{self.nome} sta studiando.")



studente1 = Studente("Clara", 22, "S12345")

studente1.saluta() # Output: Ciao, mi chiamo Clara e ho 22
anni.

studente1.studia() # Output: Clara sta studiando.
```

In questo esempio, `Studente` è una sottoclasse di `Persona`. Utilizza il metodo `super()` per chiamare il costruttore della superclasse e aggiunge un nuovo attributo `matricola` e un nuovo metodo `studia`.

5.3 POLIMORFISMO E INCAPSULAMENTO

Il polimorfismo permette di utilizzare un'interfaccia comune per oggetti di classi diverse. In Python, questo si realizza spesso definendo metodi con lo stesso nome in classi diverse.

```
class Cane:
 def suono(self):
 return "Bau"

class Gatto:
 def suono(self):
 return "Miao"

def fai_suono(animale):
 print(animale.suono())

cane = Cane()
gatto = Gatto()
```

```
fai_suono(cane) # Output: Bau

fai_suono(gatto) # Output: Miao
```

La funzione `fai_suono` può accettare qualsiasi oggetto purché abbia un metodo `suono`. Questo è il polimorfismo in azione.

Incapsulamento L'incapsulamento è il concetto di nascondere i dettagli interni di un oggetto e di permettere l'accesso e la modifica dei dati solo attraverso metodi definiti. Questo migliora la sicurezza e l'integrità dei dati.

```
class ContoBancario:

 def __init__(self, saldo_iniziale):

 self.__saldo = saldo_iniziale


def deposita(self, importo):

 if importo > 0:

 self.__saldo += importo


def preleva(self, importo):

 if 0 < importo <= self.__saldo:

 self.__saldo -= importo
```

```
def mostra_saldo(self):

 return self.__saldo


conto = ContoBancario(1000)

conto.deposita(500)

conto.preleva(200)

print(conto.mostra_saldo()) # Output: 1300
```

In questo esempio, l'attributo `__saldo` è privato (indicata dai doppi underscore __). Questo significa che non può essere accessibile direttamente dall'esterno della classe. L'accesso e la modifica del saldo sono consentiti solo tramite i metodi `deposita`, `preleva` e `mostra_saldo`, proteggendo l'integrità dei dati.

APPROFONDIMENTO SULL'OOP

La Programmazione Orientata agli Oggetti (OOP) in Python offre molte caratteristiche avanzate che permettono di creare programmi più modulari, estensibili e manutenibili. In questo approfondimento, esploreremo alcuni di questi concetti avanzati: metodi speciali, classi astratte e interfacce, e l'MRO (Method Resolution Order).

Metodi Speciali

I metodi speciali in Python, noti anche come "magic methods" o "dunder methods" (da "double underscore"), permettono di personalizzare il comportamento degli oggetti nelle operazioni standard. Alcuni dei metodi speciali più comuni includono:

- `__str__`: Definisce la rappresentazione stringa dell'oggetto per la funzione `str()` e per la funzione `print()`.
- `__repr__`: Definisce la rappresentazione "ufficiale" dell'oggetto, utile per il debugging.
- `__eq__`: Definisce il comportamento dell'operatore di uguaglianza `==`.
- `__lt__`, `__le__`, `__gt__`, `__ge__`: Definiscono il comportamento degli operatori di confronto (`<`, `<=`, `>`,

>=).

Esempio di Metodi Speciali

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

def __str__(self):
     return f"Persona(nome={self.nome}, eta={self.eta})"

def __repr__(self):
     return f"Persona('{self.nome}', {self.eta})"

def __eq__(self, other):
    if isinstance(other, Persona):
        return self.nome == other.nome and self.eta ==
other.eta
    return False

# Creazione di oggetti Persona
p1 = Persona("Alice", 30)
p2 = Persona("Bob", 25)
```

```
p3 = Persona("Alice", 30)


# Utilizzo di __str__ e __repr__

print(str(p1)) # Output: Persona(nome=Alice, eta=30)

print(repr(p1)) # Output: Persona('Alice', 30)


# Utilizzo di __eq__

print(p1 == p2) # Output: False

print(p1 == p3) # Output: True
```

Classi Astratte e Interfacce

Le classi astratte e le interfacce permettono di definire classi che non possono essere istanziate ma servono come base per altre classi. In Python, si possono creare classi astratte utilizzando il modulo `abc` (Abstract Base Classes).

Esempio di Classe Astratta

```
from abc import ABC, abstractmethod


class Forma(ABC):

    @abstractmethod
```

```
    def area(self):
          pass

    @abstractmethod
    def perimetro(self):
         pass

class Cerchio(Forma):
    def __init__(self, raggio):
       self.raggio = raggio

 def area(self):
     return 3.14 * self.raggio ** 2

 def perimetro(self):
     return 2 * 3.14 * self.raggio

class Rettangolo(Forma):
  def __init__(self, larghezza, altezza):
      self.larghezza = larghezza
      self.altezza = altezza
```

```
 def area(self):
      return self.larghezza * self.altezza

 def perimetro(self):
      return 2 * (self.larghezza + self.altezza)

# Le seguenti righe solleveranno un'eccezione perché Forma non può essere istanziata
# f = Forma()

# Creazione di oggetti Cerchio e Rettangolo
c = Cerchio(5)
r = Rettangolo(4, 6)

print(f"Area del cerchio: {c.area()}")
print(f"Perimetro del cerchio: {c.perimetro()}")
print(f"Area del rettangolo: {r.area()}")
print(f"Perimetro del rettangolo: {r.perimetro()}")
```

MRO (Method Resolution Order)

In Python, l'ordine di ricerca dei metodi in una gerarchia di classi è definito dall'MRO (Method

Resolution Order), che segue l'algoritmo C3 linearization. L'MRO determina l'ordine in cui le classi vengono ispezionate quando un metodo viene chiamato. È particolarmente utile in caso di eredità multipla.

Esempio di MRO

```
class A:
    def metodo(self):
        print("Metodo di A")


class B(A):
    def metodo(self):
        print("Metodo di B")


class C(A):
     def metodo(self):
        print("Metodo di C")

class D(B, C):
      pass

d = D()
d.metodo() # Output: Metodo di B
```

```
# Visualizza l'MRO

print(D.mro())

# Output: [<class '__main__.D'>, <class
'__main__.B'>, <class '__main__.C'>, <class
'__main__.A'>, <class 'object'>]
```

L'OOP in Python offre potenti strumenti per la gestione del codice complesso e per la creazione di strutture flessibili e riutilizzabili. Comprendere e padroneggiare i metodi speciali, le classi astratte, le interfacce e l'MRO permette di scrivere programmi Python più modulari e manutenibili. Con queste conoscenze avanzate, sei in grado di sfruttare appieno le capacità dell'OOP in Python.

CAPITOLO 6: GESTIONE DELLE ECCEZIONI

6.1 Tipi di eccezioni

La gestione delle eccezioni è un aspetto essenziale della programmazione, in quanto consente di gestire gli errori in modo efficiente e prevenire il crash dei programmi. Python offre un sistema robusto per la gestione delle eccezioni, che permette di identificare, catturare e gestire vari tipi di errori durante l'esecuzione del programma.

In Python, le eccezioni sono eventi anomali che interrompono il normale flusso di esecuzione di un programma quando si verifica un errore. Conoscere i tipi di eccezioni più comuni è fondamentale per scrivere codice robusto e gestire gli errori in modo efficace. Di seguito, esploreremo le eccezioni predefinite più comuni in Python.

1. Exception

`Exception` è la classe base per tutte le eccezioni in Python. Sebbene si possa catturare qualsiasi eccezione generica usando `Exception`, è buona pratica gestire le eccezioni specifiche per fornire messaggi di errore più dettagliati e precisi.

Esempio:

```
try:
 # Codice che potrebbe generare un'eccezione
 pass
except Exception as e:
 print(f"Si è verificata un'eccezione generica: {e}")
```

2. ValueError

`ValueError` viene sollevata quando una funzione riceve un argomento di tipo corretto, ma con un valore inappropriato. Questo tipo di eccezione è comune durante la conversione di stringhe in numeri.

Esempio:

```
try:
 numero = int("abc")
except ValueError as e:
 print(f"Errore di valore: {e}")
```

3. TypeError

`TypeError` viene sollevata quando un'operazione o una funzione viene applicata a un oggetto di tipo

inappropriato. Questo errore si verifica spesso quando si eseguono operazioni tra tipi non compatibili.

Esempio:

```
try:
 risultato = 5 + "stringa"
except TypeError as e:
 print(f"Errore di tipo: {e}")
```

4. IndexError

`IndexError` viene sollevata quando si tenta di accedere a un indice che non esiste in una sequenza, come una lista o una stringa.

Esempio:

```
try:
 lista = [1, 2, 3]
 elemento = lista[5]
except IndexError as e:
 print(f"Errore di indice: {e}")
```

5. KeyError

`KeyError` viene sollevata quando si tenta di accedere a un dizionario utilizzando una chiave che non esiste.

Esempio:

```
try:
 dizionario = {"a": 1, "b": 2}
 valore = dizionario["c"]
except KeyError as e:
 print(f"Errore di chiave: {e}")
```

6. AttributeError

`AttributeError` viene sollevata quando si tenta di accedere a un attributo di un oggetto che non esiste.

Esempio:

```
try:
 class Oggetto:
 pass

oggetto = Oggetto()
 valore = oggetto.attributo_inesistente
except AttributeError as e:
 print(f"Errore di attributo: {e}")
```

7. ZeroDivisionError

`ZeroDivisionError` viene sollevata quando si tenta di dividere un numero per zero.

Esempio:

```
try:
 risultato = 10 / 0
except ZeroDivisionError as e:
 print(f"Errore di divisione per zero: {e}")
```

8. FileNotFoundError

`FileNotFoundError` viene sollevata quando si tenta di aprire un file che non esiste.

Esempio:

```
try:
 with open("file_non_esistente.txt", "r") as file:
 contenuto = file.read()
except FileNotFoundError as e:
 print(f"Errore di file non trovato: {e}")
```

Eccezioni Integrate Addizionali

Python include molte altre eccezioni integrate che coprono diversi scenari:

- **`ImportError`** : Sollevata quando un'istruzione `import` fallisce.
- **`StopIteration`** : Sollevata per indicare la fine di un iteratore.

- **`NameError`** : Sollevata quando un nome locale o globale non è trovato.
- **`IndentationError`** : Sollevata quando c'è un problema con l'indentazione del codice.

Eccezioni Personalizzate

Oltre alle eccezioni predefinite, gli sviluppatori possono creare eccezioni personalizzate per gestire scenari specifici. Questo si ottiene creando una nuova classe che eredita da `Exception` o da una delle sue sottoclassi. Le eccezioni personalizzate aiutano a comunicare chiaramente il tipo di errore che si è verificato e a gestire situazioni particolari nel codice.

Esempio di Eccezione Personalizzata:

```
class ErrorePersonalizzato(Exception):

 pass


def funzione_rischiosa():

 raise ErrorePersonalizzato("Si è verificato un errore
personalizzato")


try:

 funzione_rischiosa()

except ErrorePersonalizzato as e:

 print(f"Errore personalizzato catturato: {e}")
```

La conoscenza dei tipi di eccezioni è essenziale per scrivere codice robusto e resiliente. Gestendo correttamente le eccezioni, gli sviluppatori possono

garantire che il loro programma continui a funzionare in modo prevedibile, anche quando si verificano errori. Python fornisce una vasta gamma di eccezioni predefinite per affrontare diversi scenari e offre la flessibilità di creare eccezioni personalizzate per gestire condizioni specifiche. Un uso efficace della gestione delle eccezioni contribuisce a migliorare la manutenibilità e la leggibilità del codice, riducendo al contempo il rischio di errori non gestiti.

6.2 BLOCCO TRY-EXCEPT

Il blocco `try-except` è il meccanismo principale in Python per gestire le eccezioni. Esso permette di isolare il codice che potrebbe generare un'eccezione e di definire le azioni da intraprendere qualora l'eccezione si verifichi. Utilizzare correttamente `try-except` migliora la robustezza del codice e assicura che il programma possa gestire gli errori in modo elegante e continuare l'esecuzione senza interrompersi inaspettatamente.

Struttura di Base del Blocco `try-except`

La struttura di un blocco `try-except` è composta da diverse parti:

- **`try` block**: Contiene il codice che potrebbe generare un'eccezione.
- **`except` block**: Definisce come gestire specifiche eccezioni che potrebbero essere sollevate nel blocco `try`.
- **`else` block (opzionale)**: Contiene il codice che viene eseguito solo se non si verifica alcuna eccezione nel blocco `try`.
- **`finally` block (opzionale)**: Viene eseguito indipendentemente dal verificarsi o meno di un'eccezione, utile per il rilascio di risorse.

Esempio di Utilizzo di `try-except`

Ecco un esempio di base che illustra come utilizzare il blocco `try-except`:

```
try:
 # Codice che potrebbe generare un'eccezione
 numero = int(input("Inserisci un numero: "))
 risultato = 10 / numero
except ValueError:
 # Gestisce un'eccezione di tipo ValueError
 print("Errore: Non hai inserito un numero valido.")
except ZeroDivisionError:
 # Gestisce un'eccezione di tipo ZeroDivisionError
 print("Errore: Divisione per zero non permessa.")
else:
 # Eseguito solo se non si verifica un'eccezione
 print(f"Il risultato è: {risultato}")
finally:
 # Eseguito indipendentemente dal verificarsi di un'eccezione
 print("Fine dell'operazione.")
```

Gestione di Eccezioni Multiple

Python permette di gestire più tipi di eccezioni all'interno di un singolo blocco `try-except`. È possibile specificare più blocchi `except` per catturare e gestire diversi tipi di eccezioni.

Esempio:

```
try:
 # Potenziale codice che genera eccezioni
 elemento = lista[indice]
except IndexError:
 print("Errore: Indice fuori dal range.")
except KeyError:
 print("Errore: Chiave non trovata nel dizionario.")
except (TypeError, ValueError) as e:
 print(f"Errore di tipo o di valore: {e}")
```

Uso del Blocco `else`

Il blocco `else` viene eseguito solo se il blocco `try` non genera alcuna eccezione. È utile per codice che deve essere eseguito solo quando non ci sono errori.

Esempio:

```
try:
```

```
 numero = int(input("Inserisci un numero: "))
 risultato = 10 / numero
except (ValueError, ZeroDivisionError) as e:
 print(f"Errore: {e}")
else:
 print(f"Risultato della divisione: {risultato}")
```

Uso del Blocco `finally`

Il blocco `finally` viene sempre eseguito, indipendentemente dal verificarsi di un'eccezione, e viene utilizzato per liberare risorse o eseguire pulizie.

Esempio:

```
try:
 file = open("dati.txt", "r")
 # Operazioni sul file
except FileNotFoundError:
 print("Errore: Il file non è stato trovato.")
finally:
 file.close()
```

```
 print("Il file è stato chiuso.")
```

Rilanciare le Eccezioni

In alcuni casi, potrebbe essere necessario rilanciare un'eccezione dopo averla catturata, ad esempio quando si desidera loggare un'eccezione ma lasciarla comunque gestire a livello superiore.

Esempio:

```
try:
 # Codice che potrebbe generare un'eccezione
 risultato = funzione_critica()
except Exception as e:
 print(f"Errore registrato: {e}")
 raise
```

Creazione di Eccezioni Specifiche

Quando si gestiscono eccezioni, è buona pratica catturare solo quelle specifiche che ci si aspetta possano verificarsi. Evitare di usare un `except` generico (`except Exception`) a meno che non si abbia una buona ragione per farlo, poiché può mascherare errori inaspettati.

Il blocco `try-except` è uno strumento potente per la gestione delle eccezioni in Python, che consente agli sviluppatori di scrivere codice robusto e gestire gli errori in modo ordinato. Utilizzando correttamente `try-except`, `else`, e `finally`, è possibile controllare il flusso del programma in presenza di errori, migliorare la leggibilità del codice e garantire che le risorse vengano gestite correttamente. Assicurarsi di catturare solo le eccezioni previste aiuta a mantenere il codice pulito e a prevenire comportamenti indesiderati.

6.3 CREAZIONE DI ECCEZIONI PERSONALIZZATE

Oltre alle eccezioni predefinite fornite da Python, gli sviluppatori hanno la possibilità di creare eccezioni personalizzate. Queste eccezioni consentono di gestire situazioni specifiche che non sono coperte dalle eccezioni predefinite e di fornire messaggi di errore più chiari e significativi. La creazione di eccezioni personalizzate è particolarmente utile in progetti di grandi dimensioni o in librerie, dove è importante comunicare in modo esplicito il tipo di errore che si è verificato.

Creazione di una Classe di Eccezione Personalizzata

Per creare un'eccezione personalizzata in Python, è necessario definire una nuova classe che erediti dalla classe base `Exception`. È possibile estendere ulteriormente questa classe aggiungendo metodi personalizzati o attributi per fornire informazioni aggiuntive sull'errore.

Esempio di Creazione di un'eccezione personalizzata:

```
class ErrorePersonalizzato(Exception):
```

```
 """Classe di eccezione personalizzata per errori
specifici."""



def __init__(self, messaggio, codice):

 super().__init__(messaggio)

 self.codice = codice



# Utilizzo dell'eccezione personalizzata

def funzione_critica(valore):

 if valore < 0:

 raise ErrorePersonalizzato("Il valore non può essere
negativo.", 1001)



try:

 funzione_critica(-1)

except ErrorePersonalizzato as e:

 print(f"Errore catturato: {e} con codice {e.codice}")
```

Vantaggi delle Eccezioni Personalizzate

Le eccezioni personalizzate offrono diversi vantaggi rispetto all'uso delle eccezioni predefinite:

- **Chiarezza**: Forniscono messaggi di errore più chiari e specifici, rendendo più semplice la comprensione del

problema.

- **Manutenzione**: Facilitano la gestione del codice in progetti complessi, dove è utile avere eccezioni specifiche per diversi moduli o componenti.
- **Informazioni Aggiuntive**: Consentono di allegare informazioni aggiuntive all'eccezione, come codici di errore o dati di contesto, che possono essere utili per il debugging o il logging.

Ereditarietà delle Eccezioni

Le eccezioni personalizzate possono anche essere strutturate in una gerarchia, creando sottoclassi di eccezioni per gestire diverse tipologie di errori correlati. Ciò consente di catturare in modo selettivo le eccezioni in base al livello di specificità desiderato.

Esempio di Ereditarietà delle eccezioni:

```
class ErroreGenerale(Exception):

 """Classe base per errori generali."""


class ErroreConnessione(ErroreGenerale):

 """Eccezione per errori di connessione."""


class ErroreTimeout(ErroreConnessione):

 """Eccezione per errori di timeout di connessione."""


# Funzione che simula un errore di connessione

def connetti_al_server(timeout):
```

```
 if timeout:

 raise ErroreTimeout("Timeout durante la connessione al
server.")



try:

 connetti_al_server(timeout=True)

except ErroreTimeout as e:

 print(f"Errore di timeout: {e}")

except ErroreConnessione as e:

 print(f"Errore di connessione: {e}")

except ErroreGenerale as e:

 print(f"Errore generale: {e}")
```

Buone Pratiche per la Creazione di Eccezioni Personalizzate

Quando si creano eccezioni personalizzate, è importante seguire alcune buone pratiche:

1. **Nomi Significativi**: Utilizzare nomi di classe che descrivano chiaramente l'errore rappresentato dall'eccezione.
2. **Documentazione**: Fornire una documentazione chiara nella classe dell'eccezione per spiegare quando e come dovrebbe essere utilizzata.
3. **Ereditarietà Corretta**: Ereditare direttamente da `Exception` o da una sottoclasse appropriata e non da

`BaseException`, che è destinata a situazioni di errore più basse nel sistema (ad esempio, interruzioni del sistema).
4. **Messaggi di Errore Utili**: Includere messaggi di errore dettagliati che aiutano a identificare e risolvere rapidamente il problema.
5. **Gestione Consapevole**: Assicurarsi che le eccezioni personalizzate siano utilizzate per casi specifici in cui le eccezioni standard non sono sufficienti o adeguate.

Le eccezioni personalizzate sono uno strumento potente per migliorare la gestione degli errori in Python. Consentono di definire errori specifici per la propria applicazione o libreria, offrendo una comunicazione più chiara e una gestione più precisa dei problemi. Seguendo le buone pratiche nella creazione e gestione delle eccezioni personalizzate, gli sviluppatori possono aumentare la leggibilità e la manutenibilità del codice, facilitando il debugging e migliorando l'esperienza utente complessiva.

APPROFONDIMENTO SULLA GESTIONE DELLE ECCEZIONI

La gestione delle eccezioni è essenziale per scrivere codice robusto e affidabile. Alcuni punti importanti da considerare includono:

- **Eccezioni specifiche**: È buona pratica gestire eccezioni specifiche piuttosto che usare un blocco `except` generico. Questo rende il codice più chiaro e più facile da debugare.
- **Logging delle eccezioni**: In applicazioni complesse, è utile loggare le eccezioni per diagnosticare problemi. Il modulo `logging` di Python fornisce strumenti potenti per il logging.
- **Ripristino da eccezioni**: In alcuni casi, potrebbe essere possibile ripristinare da un'eccezione eseguendo azioni correttive e continuando l'esecuzione del programma.

```
import logging


logging.basicConfig(level=logging.ERROR)
```

```
def dividi(a, b):

 try:

 return a / b

 except ZeroDivisionError as e:

 logging.error("Tentativo di divisione per zero.",
exc_info=True)

 return None


risultato = dividi(10, 0)
```

In questo esempio, l'eccezione `ZeroDivisionError` viene loggata con un messaggio di errore e il traceback completo, facilitando l'analisi del problema.

Con una gestione efficace delle eccezioni, è possibile creare applicazioni Python più resilienti, capaci di gestire errori imprevisti senza interrompere il normale flusso di esecuzione.

CAPITOLO 7: INPUT/OUTPUT E FILE

7.1 Lettura e scrittura di file

Lavorare con i file è una delle operazioni fondamentali in qualsiasi linguaggio di programmazione. Python offre un'ampia gamma di funzioni e metodi per gestire file in modo efficiente.

Apertura di un file

Per aprire un file in Python, si utilizza la funzione `open()`, che restituisce un oggetto file. È possibile specificare la modalità di apertura del file con il secondo argomento.

- `"r"`: Lettura (modalità predefinita)
- `"w"`: Scrittura (crea un nuovo file o sovrascrive il file esistente)
- `"a"`: Append (aggiunge dati alla fine del file senza sovrascriverlo)
- `"b"`: Modalità binaria (da combinare con `"r"`, `"w"` o `"a"`)

```
file = open("esempio.txt", "r")
```

Lettura di un file

Ci sono diversi metodi per leggere i contenuti di un file.

- **`read()`** : Legge l'intero contenuto del file.

```
contenuto = file.read()

print(contenuto)
```

- **`readline()`** : Legge una singola riga del file.

```
prima_linea = file.readline()

print(prima_linea)
```

Scrittura in un file

Per scrivere in un file, si utilizza il metodo `write()`.

```
file = open("esempio.txt", "w")

file.write("Ciao, mondo!\n")

file.write("Questa è una seconda riga.")
```

Chiusura di un file

È importante chiudere un file dopo aver terminato le operazioni su di esso per liberare le risorse.

```
file.close()
```

Utilizzo del contesto `with`

Python consente di aprire e chiudere automaticamente i file utilizzando il contesto `with`. Questo approccio è preferibile poiché gestisce automaticamente la chiusura del file anche in caso di eccezioni.

7.2 GESTIONE DEI FILE CSV

I file CSV (Comma Separated Values) sono un formato comune per memorizzare dati tabulari. Python fornisce il modulo `csv` per leggere e scrivere file CSV.

Lettura di file CSV

Per leggere un file CSV, si utilizza `csv.reader`.

```
import csv


with open("esempio.csv", "r") as file:
 reader = csv.reader(file)
 for row in reader:
 print(row)
```

Scrittura di file CSV

Per scrivere in un file CSV, si utilizza `csv.writer`.

```
import csv


dati = [
 ["Nome", "Età", "Città"],
 ["Alice", 30, "Roma"],
 ["Bob", 25, "Milano"]
]


with open("esempio.csv", "w", newline="") as file:
 writer = csv.writer(file)
 writer.writerows(dati)
```

Lettura e scrittura con `DictReader` e `DictWriter`

`csv.DictReader` e `csv.DictWriter` permettono di lavorare con file CSV come se fossero liste di dizionari.

```
import csv


with open("esempio.csv", "r") as file:
 reader = csv.DictReader(file)
 for row in reader:
 print(row)


with open("esempio.csv", "w", newline="") as file:
```

```
 fieldnames = ["Nome", "Età", "Città"]

 writer = csv.DictWriter(file, fieldnames=fieldnames)

 writer.writeheader()

 writer.writerow({"Nome": "Alice", "Età": 30, "Città":
"Roma"})

 writer.writerow({"Nome": "Bob", "Età": 25, "Città":
"Milano"})
```

7.3 OPERAZIONI CON I FILE JSON

JSON (JavaScript Object Notation) è un formato leggero per lo scambio di dati. Python fornisce il modulo `json` per leggere e scrivere dati in formato JSON.

Lettura di file JSON

Per leggere un file JSON, si utilizza `json.load`.

```
import json


with open("esempio.json", "r") as file:
 dati = json.load(file)
 print(dati)
```

Scrittura di file JSON

Per scrivere dati in un file JSON, si utilizza `json.dump`.

```
import json
```

```
dati = {
 "nome": "Alice",
 "età": 30,
 "città": "Roma"
}

with open("esempio.json", "w") as file:
 json.dump(dati, file, indent=4)
```

Convertire tra JSON e stringhe

È possibile convertire un oggetto Python in una stringa JSON utilizzando `json.dumps` e convertire una stringa JSON in un oggetto Python utilizzando `json.loads`.

```
import json

# Convertire un oggetto Python in una stringa JSON
dati = {
 "nome": "Alice",
 "età": 30,
 "città": "Roma"
```

```
}

json_string = json.dumps(dati, indent=4)

print(json_string)


# Convertire una stringa JSON in un oggetto Python

dati_caricati = json.loads(json_string)

print(dati_caricati)
```

La gestione dei file in Python è una competenza fondamentale che consente di leggere, scrivere e manipolare dati in vari formati. Con la conoscenza delle tecniche di lettura e scrittura di file, gestione di file CSV e operazioni con file JSON, è possibile gestire efficacemente dati strutturati in Python.

CAPITOLO 8: PYTHON E DATABASE

8.1 Introduzione ai Database

I database sono un componente essenziale per molte applicazioni, permettendo la memorizzazione, l'organizzazione e la gestione dei dati. Ci sono vari tipi di database, ma possono essere generalmente classificati in due categorie principali: database relazionali e non relazionali.

Database Relazionali

I database relazionali organizzano i dati in tabelle, che sono costituite da righe e colonne. Ogni tabella rappresenta un'entità e le colonne rappresentano gli attributi di quell'entità. Le righe, chiamate anche record, contengono i dati effettivi. Un database relazionale utilizza il linguaggio SQL (Structured Query Language) per la gestione dei dati.

Caratteristiche principali:

- **Schema fisso**: I database relazionali richiedono una struttura definita (schema) per le tabelle.
- **Integrità dei dati**: Vincoli e chiavi possono essere definiti per mantenere l'integrità dei dati.

- **Relazioni**: Le tabelle possono essere collegate tramite chiavi esterne, permettendo di stabilire relazioni tra i dati.
- **Transazioni**: Le operazioni sui dati possono essere eseguite in transazioni, garantendo la coerenza dei dati anche in caso di errori o guasti.

Esempi di database relazionali:

- MySQL
- PostgreSQL
- SQLite
- Oracle Database
- Microsoft SQL Server

Database Non Relazionali (NoSQL)

I database non relazionali, comunemente noti come NoSQL, non utilizzano un modello tabellare fisso. Sono progettati per gestire grandi volumi di dati, strutture di dati dinamiche e accessi rapidi ai dati. I database NoSQL sono particolarmente adatti per applicazioni che richiedono scalabilità e flessibilità.

Tipi comuni di database NoSQL:

- **Database a documenti**: Memorizzano i dati in documenti JSON o BSON. Esempi: MongoDB, CouchDB.
- **Database a colonne**: Memorizzano i dati in colonne piuttosto che in righe. Esempi: Apache Cassandra, HBase.
- **Database a chiave-valore**: Memorizzano i dati come coppie chiave-valore. Esempi: Redis, DynamoDB.
- **Database a grafo**: Memorizzano i dati come nodi e relazioni tra nodi. Esempi: Neo4j, Amazon Neptune.

Caratteristiche principali:

- **Schema flessibile**: Non richiedono uno schema fisso per i dati.
- **Scalabilità orizzontale**: Progettati per scalare aggiungendo più macchine piuttosto che potenziando le macchine esistenti.
- **Elevate prestazioni**: Ottimizzati per accessi rapidi ai dati e per gestire grandi volumi di dati.

8.2 CONNESSIONE A DATABASE SQLITE

SQLite è un database relazionale leggero e integrato, perfetto per applicazioni desktop, mobile e piccoli progetti. È serverless, senza configurazione e trasparente per gli sviluppatori, che possono includerlo direttamente nelle loro applicazioni.

Installazione del modulo sqlite3

SQLite è integrato in Python tramite il modulo `sqlite3`, quindi non è necessario installare pacchetti aggiuntivi.

Connessione a un database SQLite

Per connettersi a un database SQLite, si utilizza il modulo `sqlite3`. Se il database non esiste, SQLite ne creerà uno nuovo.

```
import sqlite3


# Connessione a un database (crea il file se non esiste)
conn = sqlite3.connect('esempio.db')
```

```
# Creazione di un cursore

cur = conn.cursor()
```

Creazione di una tabella

Una volta connessi al database, è possibile creare tabelle utilizzando le istruzioni SQL.

```
cur.execute('''

CREATE TABLE IF NOT EXISTS utenti (

 id INTEGER PRIMARY KEY AUTOINCREMENT,

 nome TEXT NOT NULL,

 età INTEGER NOT NULL

)

''')

# Confermare le modifiche

conn.commit()
```

Inserimento di dati

Per inserire dati in una tabella, si utilizza l'istruzione `INSERT`.

```
cur.execute('''
INSERT INTO utenti (nome, età) VALUES (?, ?)
''', ('Alice', 30))

# Confermare le modifiche
conn.commit()
```

Lettura dei dati

Per leggere i dati da una tabella, si utilizza l'istruzione `SELECT`.

```
cur.execute('SELECT * FROM utenti')
rows = cur.fetchall()

for row in rows:
 print(row)
```

Aggiornamento dei dati

Per aggiornare i dati in una tabella, si utilizza l'istruzione `UPDATE`.

```
cur.execute('''
UPDATE utenti SET età = ? WHERE nome = ?
''', (31, 'Alice'))

# Confermare le modifiche
conn.commit()
```

Eliminazione dei dati

Per eliminare i dati da una tabella, si utilizza l'istruzione `DELETE`.

```
cur.execute('''
DELETE FROM utenti WHERE nome = ?
''', ('Alice',))

# Confermare le modifiche
conn.commit()
```

Chiusura della connessione

È importante chiudere la connessione al database quando non è più necessaria.ù

```
conn.close()
```

8.3 OPERAZIONI CRUD (CREATE, READ, UPDATE, DELETE)

Le operazioni CRUD sono fondamentali per la gestione dei dati in un database. Vediamo come eseguire queste operazioni utilizzando SQLite e il modulo `sqlite3`.

Create (Creazione)

La creazione di dati implica l'inserimento di nuovi record in una tabella. Utilizzando l'istruzione `INSERT`, possiamo aggiungere nuovi utenti alla tabella `utenti`.

```
import sqlite3


# Connessione al database

conn = sqlite3.connect('esempio.db')

cur = conn.cursor()


# Inserimento di nuovi utenti

cur.execute('''

INSERT INTO utenti (nome, età) VALUES (?, ?)
```

```
''', ('Bob', 25))

cur.execute('''

INSERT INTO utenti (nome, età) VALUES (?, ?)

''', ('Clara', 28))

# Confermare le modifiche

conn.commit()


# Chiusura della connessione

conn.close()
```

Read (Lettura)

La lettura dei dati implica la selezione e il recupero dei record dalla tabella. Utilizzando l'istruzione `SELECT`, possiamo leggere tutti gli utenti dalla tabella `utenti`.

```
import sqlite3


# Connessione al database

conn = sqlite3.connect('esempio.db')

cur = conn.cursor()


# Lettura di tutti gli utenti
```

```
cur.execute('SELECT * FROM utenti')

rows = cur.fetchall()


for row in rows:

 print(row)


# Chiusura della connessione

conn.close()
```

Update (Aggiornamento)

L'aggiornamento dei dati implica la modifica dei record esistenti. Utilizzando l'istruzione `UPDATE`, possiamo aggiornare l'età di un utente specifico.

```
import sqlite3


# Connessione al database

conn = sqlite3.connect('esempio.db')

cur = conn.cursor()


# Aggiornamento dell'età di Bob

cur.execute('''

UPDATE utenti SET età = ? WHERE nome = ?
```

```
''', (26, 'Bob'))

# Confermare le modifiche
conn.commit()

# Chiusura della connessione
conn.close()
```

Delete (Eliminazione)

L'eliminazione dei dati implica la rimozione dei record dalla tabella. Utilizzando l'istruzione `DELETE`, possiamo eliminare un utente specifico dalla tabella `utenti`.

```
import sqlite3

# Connessione al database
conn = sqlite3.connect('esempio.db')
cur = conn.cursor()

# Eliminazione di Clara
cur.execute('''
DELETE FROM utenti WHERE nome = ?
```

```
''', ('Clara',))

# Confermare le modifiche
conn.commit()

# Chiusura della connessione
conn.close()
```

La gestione dei database è una competenza fondamentale per qualsiasi sviluppatore. In questo capitolo, abbiamo esplorato i concetti di base dei database relazionali e non relazionali, la connessione a un database SQLite utilizzando il modulo `sqlite3`, e le operazioni CRUD fondamentali. Queste competenze permettono di gestire e manipolare i dati in modo efficiente, consentendo di creare applicazioni robuste e scalabili.

CAPITOLO 9: SVILUPPO WEB CON PYTHON

9.1 Introduzione a Flask

Flask è un micro-framework per lo sviluppo di applicazioni web in Python. È stato creato da Armin Ronacher come parte del progetto Pocoo e rilasciato per la prima volta nel 2010. Flask è progettato per essere leggero e flessibile, consentendo agli sviluppatori di creare applicazioni web rapidamente e facilmente. Nonostante la sua semplicità, Flask è molto potente e può essere utilizzato per creare applicazioni web di produzione.

Caratteristiche principali di Flask

1. **Leggerezza**:
    - Flask è un framework minimo, senza molte funzionalità predefinite. A differenza di altri framework web più completi come Django, Flask non impone molteplici strati di astrazione o convenzioni. Invece, fornisce solo gli strumenti essenziali per costruire un'applicazione web, permettendo agli sviluppatori di aggiungere solo ciò di cui hanno bisogno. Questa leggerezza consente di avere un controllo maggiore sul flusso e sulla logica dell'applicazione.

2. **Flessibilità**:
    - Flask consente di strutturare l'applicazione come si preferisce, senza imporre un'organizzazione rigida. Gli sviluppatori possono organizzare il loro progetto come meglio credono, sia che si tratti di un piccolo script o di una grande applicazione con molteplici moduli. Questo approccio rende Flask ideale per progetti di qualsiasi dimensione.
3. **Estendibilità**:
    - Flask può essere esteso con numerosi plugin e estensioni che aggiungono funzionalità come l'autenticazione, il supporto per i database, la gestione delle sessioni, il controllo degli accessi e molto altro. Alcune estensioni popolari includono Flask-SQLAlchemy (per l'integrazione con i database), Flask-Migrate (per la gestione delle migrazioni del database), Flask-WTF (per la gestione dei moduli), e Flask-Login (per l'autenticazione degli utenti).
4. **Semplicità**:
    - Flask è facile da apprendere e da usare, con una curva di apprendimento bassa. Anche chi ha poca esperienza con lo sviluppo web può rapidamente creare applicazioni funzionanti. La sua documentazione chiara e completa, insieme a una vasta comunità di sviluppatori, rende l'apprendimento e l'uso di Flask un'esperienza piacevole.

Filosofia di Flask

La filosofia di Flask si basa su alcuni principi fondamentali:

- **Keep It Simple and Lightweight**:
  - Flask cerca di mantenere il nucleo del framework il più piccolo possibile, aggiungendo solo funzionalità essenziali e lasciando agli sviluppatori la libertà di aggiungere altre funzionalità tramite estensioni o librerie esterne.
- **Easy to Get Started**:
  - Flask è progettato per essere facile da iniziare. Con poche righe di codice, è possibile avere una semplice applicazione web funzionante. Questo rende Flask ideale per i prototipi e per le applicazioni che necessitano di essere sviluppate rapidamente.
- **Flexible and Extensible**:
  - Flask non impone molte restrizioni su come deve essere organizzata un'applicazione, permettendo agli sviluppatori di prendere decisioni architettoniche che meglio si adattano al loro progetto. La possibilità di estendere Flask con plugin e librerie esterne aggiunge un ulteriore livello di flessibilità.

Installazione di Flask

Per installare Flask, si utilizza il gestore di pacchetti `pip`. È consigliabile creare un ambiente virtuale per isolare le dipendenze del progetto.

```
# Creazione di un ambiente virtuale

python -m venv env

# Attivazione dell'ambiente virtuale
```

```
# Su Windows

env\Scripts\activate

# Su macOS/Linux

source env/bin/activate


# Installazione di Flask

pip install Flask
```

Creazione di una semplice applicazione Flask

Vediamo come creare una semplice applicazione web con Flask. Iniziamo creando un file chiamato `app.py`.

app.py:

from flask import Flask


```
# Creazione dell'istanza dell'applicazione Flask

app = Flask(__name__)


# Definizione di una route per la homepage

@app.route('/')

def home():

 return "Ciao, mondo!"
```

```
# Esecuzione dell'applicazione

if __name__ == '__main__':

 app.run(debug=True)
```

Salvare il file e eseguirlo con il comando:

```
 python app.py
```

L'applicazione sarà disponibile all'indirizzo `http://127.0.0.1:5000/`. Visitando questo URL nel browser, si vedrà il messaggio "Ciao, mondo!".

Gestione delle route

Le route in Flask sono definite utilizzando il decoratore `@app.route`. Questo decoratore mappa una URL specifica a una funzione che gestisce le richieste a quell'URL.

app.py:

```
from flask import Flask


app = Flask(__name__)


@app.route('/')
```

```
def home():
 return "Ciao, mondo!"


@app.route('/about')
def about():
 return "Questa è la pagina about."


if __name__ == '__main__':
 app.run(debug=True)
```

Visitando `http://127.0.0.1:5000/about`, si vedrà il messaggio "Questa è la pagina about.".

Gestione dei template

Flask utilizza il motore di template Jinja2 per generare l'output HTML dinamico. I template sono file HTML che possono includere segnaposto e logica per generare contenuti dinamici.

templates/index.html:

```
<!doctype html>
<html lang="it">
<head>
```

```
 <meta charset="utf-8">
 <title>Home Page</title>
</head>
<body>
 <h1>Ciao, mondo!</h1>
 <p>Questa è una semplice applicazione Flask.</p>
</body>
</html>
```

Modifichiamo l'applicazione per utilizzare il template.
app.py:

```
from flask import Flask, render_template

app = Flask(__name__)


@app.route('/')
def home():
 return render_template('index.html')


if __name__ == '__main__':
 app.run(debug=True)
```

Ora, visitando `http://127.0.0.1:5000/`, vedremo il contenuto del file `index.html`.

Gestione delle richieste HTTP

Flask permette di gestire diversi tipi di richieste HTTP (GET, POST, ecc.). Vediamo un esempio di gestione delle richieste POST.

templates/form.html:

```
<!doctype html>

<html lang="it">

<head>

 <meta charset="utf-8">

 <title>Modulo</title>

</head>

<body>

 <h1>Invia il tuo nome</h1>

 <form method="POST">

 <input type="text"  placeholder="Inserisci il tuo
nome">

 <button type="submit">Invia</button>

 </form>

</body>
```

```
</html>
```

app.py:

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/')
def home():
 return render_template('index.html')

@app.route('/form', methods=['GET', 'POST'])
def form():
 if request.method == 'POST':
 nome = request.form['nome']
 return f"Ciao, {nome}!"
 return render_template('form.html')

if __name__ == '__main__':
 app.run(debug=True)
```

Flask permette di gestire diversi tipi di richieste HTTP (GET, POST, ecc.). Vediamo un esempio di gestione delle richieste POST.Flask è un micro-framework potente e flessibile per lo sviluppo di applicazioni web in Python. La sua leggerezza, flessibilità ed estendibilità lo rendono una scelta eccellente per progetti di qualsiasi dimensione, dai prototipi rapidi alle applicazioni di produzione complete. Con Flask, è possibile creare applicazioni web moderne e dinamiche, sfruttando la semplicità e la potenza di Python.

9.2 CREAZIONE DI UNA SEMPLICE APPLICAZIONE WEB

In questa sezione, svilupperemo una semplice applicazione web utilizzando Flask, coprendo i vari passaggi necessari per configurare, sviluppare e eseguire l'applicazione. Questa guida fornirà una panoramica dettagliata di come creare una struttura di base per una semplice applicazione web, includendo routing, gestione delle richieste, template rendering e interazione con un database.

Passo 1: Installazione di Flask

Prima di iniziare, è necessario installare Flask. Si consiglia di utilizzare un ambiente virtuale per isolare le dipendenze del progetto.

```
# Creazione di un ambiente virtuale

python -m venv env



# Attivazione dell'ambiente virtuale

# Su Windows
```

```
env\Scripts\activate

# Su macOS/Linux

source env/bin/activate


# Installazione di Flask

pip install Flask
```

Passo 2: Configurazione dell'applicazione

Creiamo la struttura di base per l'applicazione. Iniziamo creando una directory per il progetto e un file chiamato `app.py`.

Struttura del progetto:

```
my_flask_app/

 ├── app.py

 ├── templates/

 │ └── index.html

 └── static/

 └── style.css
```

app.py:

```
from flask import Flask, render_template, request, redirect,
url_for
```

```
app = Flask(__name__)

# Route per la homepage
@app.route('/')
def home():
 return render_template('index.html')

# Route per il modulo
@app.route('/form', methods=['GET', 'POST'])
def form():
 if request.method == 'POST':
 nome = request.form['nome']
 return redirect(url_for('saluto', nome=nome))
 return render_template('form.html')

# Route per il saluto
@app.route('/saluto/<nome>')
def saluto(nome):
 return f"Ciao, {nome}!"

if __name__ == '__main__':
 app.run(debug=True)
```

Passo 3: Creazione dei template

Creiamo ora i template HTML che verranno utilizzati dall'applicazione. I template saranno salvati nella directory `templates`.

templates/index.html:

```
<!doctype html>

<html lang="it">

<head>

 <meta charset="utf-8">

 <title>Home Page</title>

 <link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">

</head>

<body>

 <h1>Benvenuto nella mia applicazione Flask</h1>

 <p><a href="{{ url_for('form') }}">Vai al
modulo</a></p>

</body>

</html>
```

templates/form.html:

```
<!doctype html>

<html lang="it">

<head>

 <meta charset="utf-8">

 <title>Modulo</title>

 <link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">

</head>

<body>

 <h1>Invia il tuo nome</h1>

 <form method="POST">

 <input type="text"  placeholder="Inserisci il tuo
nome">

 <button type="submit">Invia</button>

 </form>

</body>

</html>
```

Passo 4: Aggiunta di stili CSS

Per rendere la nostra applicazione più attraente, aggiungiamo un file CSS nella directory `static`.

static/style.css:

```
body {
 font-family: Arial, sans-serif;
 margin: 0;
 padding: 0;
 text-align: center;
 background-color: #f0f0f0;
}

h1 {
 color: #333;
 margin-top: 50px;
}

p, form {
 margin: 20px;
}

input[type="text"] {
 padding: 10px;
 font-size: 16px;
 margin: 10px 0;
}
```

```
button {
 padding: 10px 20px;
 font-size: 16px;
 background-color: #007BFF;
 color: #fff;
 border: none;
 cursor: pointer;
}

button:hover {
 background-color: #0056b3;
}
```

Passo 5: Esecuzione dell'applicazione

Ora che abbiamo configurato l'applicazione, creato i template e aggiunto gli stili CSS, possiamo eseguire l'applicazione.

```
python app.py
```

L'applicazione sarà disponibile all'indirizzo `http://127.0.0.1:5000/`. Visitando questo URL nel browser, si vedrà la homepage. Cliccando sul link "Vai al modulo",

si verrà reindirizzati alla pagina del modulo dove è possibile inserire un nome e inviarlo. Dopo l'invio, si verrà reindirizzati a una pagina che saluta l'utente per nome.

Passo 6: Aggiunta di funzionalità avanzate

6.1: Gestione dei messaggi flash

Flask permette di utilizzare messaggi flash per fornire feedback agli utenti. I messaggi flash sono particolarmente utili per notificare gli utenti di eventi come errori di validazione, salvataggi riusciti, ecc.

app.py:

```
from flask import Flask, render_template, request, redirect,
url_for, flash



app = Flask(__name__)

app.secret_key = 'supersecretkey'



@app.route('/')

def home():

 return render_template('index.html')



@app.route('/form', methods=['GET', 'POST'])

def form():

 if request.method == 'POST':
```

```
 nome = request.form['nome']
 if nome:
 flash('Nome inviato con successo!', 'success')
 return redirect(url_for('saluto', nome=nome))
 else:
 flash('Per favore, inserisci un nome.', 'error')
 return render_template('form.html')

@app.route('/saluto/<nome>')
def saluto(nome):
 return f"Ciao, {nome}!"

if __name__ == '__main__':
 app.run(debug=True)
```

templates/form.html:

```
<!doctype html>
<html lang="it">
<head>
 <meta charset="utf-8">
```

```
 <title>Modulo</title>

 <link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">

</head>

<body>

 <h1>Invia il tuo nome</h1>

 {% with messages =
get_flashed_messages(with_categories=true) %}

 {% if messages %}

 {% for category, message in messages %}

 <div class="flash {{ category }}">{{ message
}}</div>

 {% endfor %}

 {% endif %}

 {% endwith %}

 <form method="POST">

 <input type="text"  placeholder="Inserisci il tuo
nome">

 <button type="submit">Invia</button>

 </form>

</body>

</html>
```

static/style.css:

```
.flash {
 margin: 20px;
 padding: 10px;
 border-radius: 5px;
 text-align: center;
}

.flash.success {
 background-color: #d4edda;
 color: #155724;
}

.flash.error {
 background-color: #f8d7da;
 color: #721c24;
}
```

6.2: Interazione con un database

Per rendere l'applicazione più completa, possiamo aggiungere un database SQLite per memorizzare i dati inseriti dagli utenti.

1. **Installazione di Flask-SQLAlchemy**:

```
pip install Flask-SQLAlchemy
```

1. **Configurazione del database**:

app.py:

```
from flask import Flask, render_template, request, redirect,
url_for, flash

from flask_sqlalchemy import SQLAlchemy


app = Flask(__name__)

app.secret_key = 'supersecretkey'

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///nomi.db'

app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False


db = SQLAlchemy(app)


class Nome(db.Model):

 id = db.Column(db.Integer, primary_key=True)

 nome = db.Column(db.String(80), nullable=False)


@app.route('/')
```

```
def home():
 return render_template('index.html')

@app.route('/form', methods=['GET', 'POST'])
def form():
 if request.method == 'POST':
 nome = request.form['nome']
 if nome:
 nuovo_nome = Nome(nome=nome)
 db.session.add(nuovo_nome)
 db.session.commit()
 flash('Nome inviato con successo!', 'success')
 return redirect(url_for('saluto', nome=nome))
 else:
 flash('Per favore, inserisci un nome.', 'error')
 return render_template('form.html')

@app.route('/saluto/<nome>')
def saluto(nome):
 return f"Ciao, {nome}!"

if __name__ == '__main__':
```

```
 db.create_all() # Creazione del database e delle tabelle
 app.run(debug=True)
```

1. **Visualizzazione dei nomi salvati**:

app.py:

```
@app.route('/nomi')
def nomi():
 tutti_nomi = Nome.query.all()
 return render_template('nomi.html', nomi=tutti_nomi)
```

templates/nomi.html:

```
<!doctype html>
<html lang="it">
<head>
 <meta charset="utf-8">
 <title>Nomi salvati</title>
 <link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">
</head>
```

```
<body>

 <h1>Nomi salvati</h1>

 <ul>

 {% for nome in nomi %}

 <li>{{ nome.nome }}</li>

 {% endfor %}

 </ul>

 <p><a href="{{ url_for('home') }}">Torna alla
home</a></p>

</body>

</html>
```

In questa sezione, abbiamo sviluppato una semplice applicazione web utilizzando Flask, esplorando i vari passaggi necessari per configurare, sviluppare e eseguire l'applicazione. Abbiamo coperto la creazione di route, la gestione delle richieste HTTP, l'uso dei template per il rendering dinamico, la gestione dei messaggi flash e l'interazione con un database SQLite. Con queste basi, è possibile creare applicazioni web più complesse e funzionali utilizzando Flask.

9.3 INTRODUZIONE A DJANGO

Django è un framework web ad alto livello per lo sviluppo di applicazioni web complesse e scalabili. Creato da Adrian Holovaty e Simon Willison nel 2005, Django è progettato per facilitare lo sviluppo rapido e ordinato di applicazioni web, promuovendo il principio DRY (Don't Repeat Yourself) e la modularità del codice.

Caratteristiche principali di Django

1. **Completezza**:
   - Django è un framework completo, che include tutto il necessario per costruire un'applicazione web robusta. Offre strumenti integrati per la gestione del database, la validazione dei form, l'autenticazione degli utenti, la gestione dei file statici e molto altro. Questo permette agli sviluppatori di concentrarsi sulla logica dell'applicazione piuttosto che sulla configurazione e l'integrazione di diverse librerie.
2. **Modularità**:
   - Django è progettato per essere altamente modulare, permettendo di organizzare il codice in applicazioni riutilizzabili. Ogni applicazione può avere i propri modelli, viste, URL, template e

file statici, facilitando la manutenzione e il riutilizzo del codice.

3. **ORM (Object-Relational Mapping)**:
   - Django include un potente ORM che permette di interagire con il database utilizzando oggetti Python invece di scrivere query SQL. L'ORM di Django supporta vari database relazionali come PostgreSQL, MySQL, SQLite e Oracle.
4. **Sistema di template**:
   - Django utilizza un motore di template basato su tag e filtri per generare HTML dinamico. I template Django sono facili da leggere e scrivere, e separano chiaramente la logica della presentazione dalla logica dell'applicazione.
5. **Sicurezza**:
   - Django include molteplici funzionalità di sicurezza integrate per proteggere le applicazioni da attacchi comuni come SQL injection, cross-site scripting (XSS), cross-site request forgery (CSRF) e clickjacking.
6. **Admin Interface**:
   - Django fornisce un'interfaccia amministrativa automatica che permette di gestire i dati dell'applicazione attraverso un'interfaccia web. Questa interfaccia è altamente configurabile e può essere utilizzata per gestire utenti, gruppi, permessi e molto altro.

Confronto tra Flask e Django

1. **Dimensioni e complessità**:
   - Flask è un micro-framework leggero e flessibile, mentre Django è un framework completo e robusto. Flask è ideale per progetti piccoli o prototipi, dove la flessibilità e la velocità di sviluppo sono cruciali. Django, d'altra parte, è

progettato per applicazioni complesse e su larga scala, dove la completezza e la coerenza sono essenziali.

2. **Flessibilità vs. Convenienza**:
   - Flask offre più flessibilità nella scelta dei componenti e nella struttura dell'applicazione. Gli sviluppatori possono scegliere liberamente le librerie e i moduli da utilizzare. Django, invece, fornisce un insieme di componenti integrati che facilitano lo sviluppo rapido e coerente, ma con meno flessibilità rispetto a Flask.
3. **Curva di apprendimento**:
   - Flask ha una curva di apprendimento più bassa, rendendolo adatto per i principianti e per chi desidera avvicinarsi rapidamente allo sviluppo web. Django richiede più tempo per essere padroneggiato, ma offre una struttura più rigorosa e completa per lo sviluppo di applicazioni complesse.
4. **Comunità e documentazione**:
   - Entrambi i framework hanno una forte comunità e una documentazione eccellente. Django, essendo più vecchio e diffuso, ha una comunità più ampia e una maggiore quantità di risorse disponibili, incluse librerie di terze parti e plugin.

Installazione di Django

Per installare Django, si utilizza `pip`. È consigliabile creare un ambiente virtuale per isolare le dipendenze del progetto.

```
# Creazione di un ambiente virtuale

python -m venv env
```

```
# Attivazione dell'ambiente virtuale

# Su Windows

env\Scripts\activate

# Su macOS/Linux

source env/bin/activate


# Installazione di Django

pip install django
```

Creazione di un progetto Django

1. **Creare un nuovo progetto**:

```
django-admin startproject mio_progetto
```

Questo comando crea una struttura di directory per il progetto Django, includendo i file di configurazione necessari.

1. **Navigare nella directory del progetto**:

```
cd mio_progetto
```

1. **Creare una nuova applicazione**:

```
python manage.py startapp mia_app
```

Questo comando crea una nuova applicazione all'interno del progetto, con la propria struttura di directory.

Configurazione del progetto

Modifichiamo il file `settings.py` per aggiungere l'applicazione alla lista delle `INSTALLED_APPS`.
mio_progetto/settings.py:

```
INSTALLED_APPS = [
 # Applicazioni predefinite di Django
 'django.contrib.admin',
 'django.contrib.auth',
 'django.contrib.contenttypes',
 'django.contrib.sessions',
 'django.contrib.messages',
 'django.contrib.staticfiles',
 # Applicazione personalizzata
```

```
 'mia_app',

]
```

Definizione di una vista

Creiamo una vista che gestisce le richieste HTTP e restituisce una risposta.
mia_app/views.py:

```
from django.http import HttpResponse


def home(request):

 return HttpResponse("Ciao, mondo!")
```

Configurazione delle URL

Associare la vista a un URL specifico.
mia_app/urls.py:

```
from django.urls import path

from . import views


urlpatterns = [
```

```
 path('', views.home, name='home'),

]
```

Modificare il file delle URL del progetto per includere le URL dell'applicazione.

mio_progetto/urls.py:

```
from django.contrib import admin

from django.urls import include, path


urlpatterns = [

 path('admin/', admin.site.urls),

 path('', include('mia_app.urls')),

]
```

Esecuzione del server di sviluppo

Eseguire il server di sviluppo per testare l'applicazione.

```
python manage.py runserver
```

L'applicazione sarà disponibile all'indirizzo `http://127.0.0.1:8000/`.

Creazione di modelli e interazione con il database

Django utilizza l'ORM per definire i modelli che rappresentano le tabelle del database. Creiamo un modello per memorizzare i dati degli utenti.

mia_app/models.py:

```
from django.db import models


class Utente(models.Model):

 nome = models.CharField(max_length=100)

 email = models.EmailField()


def __str__(self):

 return self.nome
```

Registrare il modello nell'interfaccia amministrativa di Django.

mia_app/admin.py:

```
from django.contrib import admin

from .models import Utente


admin.site.register(Utente)
```

Eseguire le migrazioni per creare le tabelle nel database.

```
python manage.py makemigrations

python manage.py migrate
```

Creazione di form e validazione dei dati

Django include strumenti integrati per la creazione e la gestione dei form. Creiamo un form per l'inserimento dei dati degli utenti.

mia_app/forms.py:

```
from django import forms

from .models import Utente


class UtenteForm(forms.ModelForm):

 class Meta:

 model = Utente

 fields = ['nome', 'email']
```

Aggiorniamo la vista per gestire il form.

mia_app/views.py:

```
from django.shortcuts import render, redirect
```

```
from .forms import UtenteForm


def home(request):
 if request.method == 'POST':
 form = UtenteForm(request.POST)
 if form.is_valid():
 form.save()
 return redirect('home')
 else:
 form = UtenteForm()
 return render(request, 'home.html', {'form': form})
```

Creiamo un template per il form.
mia_app/templates/home.html:

```
<!doctype html>
<html lang="it">
<head>
 <meta charset="utf-8">
 <title>Home Page</title>
</head>
<body>
```

```
 <h1>Inserisci i tuoi dati</h1>

 <form method="POST">

 {% csrf_token %}

 {{ form.as_p }}

 <button type="submit">Invia</button>

 </form>

</body>

</html>
```

Django è un framework web ad alto livello che offre tutto il necessario per costruire applicazioni web complesse e scalabili. Con le sue numerose funzionalità integrate, Django facilita lo sviluppo rapido e ordinato, promuovendo la modularità e la sicurezza del codice. Sebbene abbia una curva di apprendimento più ripida rispetto a Flask, Django è ideale per progetti di grandi dimensioni e per chi cerca una soluzione completa e robusta. La scelta tra Flask e Django dipende dalle esigenze specifiche del progetto, dalla complessità dell'applicazione e dalle preferenze personali dello sviluppatore.

CAPITOLO 10: DATA SCIENCE CON PYTHON

10.1 Librerie principali: NumPy e Pandas

Python è diventato uno degli strumenti più utilizzati nella data science grazie alla sua semplicità e alla vasta gamma di librerie disponibili. Tra le librerie più importanti per la data science troviamo NumPy e Pandas. In questa sezione, esploreremo queste due librerie, il loro utilizzo e le loro applicazioni.

NumPy

NumPy (Numerical Python) è una libreria fondamentale per la computazione scientifica in Python. Fornisce supporto per array e matrici multidimensionali, insieme a una vasta collezione di funzioni matematiche per operare su questi array. NumPy è costruito in C, il che lo rende molto veloce e adatto per calcoli numerici intensivi.

Installazione

Per installare NumPy, si utilizza il gestore di pacchetti `pip`.

```
pip install numpy
```

Creazione di array

Gli array di NumPy sono simili alle liste di Python, ma con funzionalità aggiuntive per operazioni matematiche e analisi dei dati.

```
import numpy as np


# Creazione di un array unidimensionale

array_1d = np.array([1, 2, 3, 4, 5])

print(array_1d)


# Creazione di un array bidimensionale (matrice)

array_2d = np.array([[1, 2, 3], [4, 5, 6]])

print(array_2d)
```

Operazioni sugli array

NumPy fornisce una serie di operazioni per manipolare gli array.

```
# Operazioni aritmetiche

array = np.array([1, 2, 3, 4])

print(array + 2) # Aggiunta di uno scalare

print(array * 2) # Moltiplicazione per uno scalare


# Operazioni tra array

array_1 = np.array([1, 2, 3])

array_2 = np.array([4, 5, 6])

print(array_1 + array_2) # Somma degli array

print(array_1 * array_2) # Moltiplicazione elemento per
elemento


# Operazioni matematiche

print(np.sqrt(array)) # Radice quadrata

print(np.exp(array)) # Esponenziale

print(np.mean(array)) # Media
```

Indicizzazione e slicing

L'indicizzazione e lo slicing degli array di NumPy sono simili a quelli delle liste di Python, ma con funzionalità aggiuntive.

```
array = np.array([1, 2, 3, 4, 5])
```

```
# Indicizzazione
print(array[0]) # Primo elemento
print(array[-1]) # Ultimo elemento

# Slicing
print(array[1:4]) # Elementi dal secondo al quarto
print(array[:3]) # Primi tre elementi
print(array[::2]) # Elementi con passo 2

# Slicing in array bidimensionali
array_2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(array_2d[1:, 1:]) # Slicing di una sotto-matrice
```

Funzioni utili di NumPy

NumPy include molte funzioni utili per creare e manipolare array.

```
# Creazione di array di zeri e uno
zeros = np.zeros((2, 3))
ones = np.ones((3, 4))
```

```
# Creazione di array con valori casuali

random_array = np.random.random((2, 2))


# Creazione di array con una sequenza di numeri

sequence = np.arange(0, 10, 2)


# Reshape di un array

reshaped_array = np.arange(1, 7).reshape((2, 3))
```

Pandas

Pandas è una libreria potente per la manipolazione e l'analisi dei dati. Fornisce due strutture dati principali: `Series` (per dati unidimensionali) e `DataFrame` (per dati bidimensionali). Pandas è costruito sopra NumPy e offre strumenti per lavorare con dati etichettati o relazionali.

Installazione

Per installare Pandas, si utilizza `pip`.

```
pip install pandas
```

Creazione di Series e DataFrame

Le `Series` sono simili agli array unidimensionali di NumPy, ma con un indice etichettato. I `DataFrame` sono simili alle tabelle dei database o ai fogli di calcolo.

```
import pandas as pd


# Creazione di una Series

serie = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])

print(serie)


# Creazione di un DataFrame da un dizionario

data = {

 'Nome': ['Alice', 'Bob', 'Clara'],

 'Età': [25, 30, 35],

 'Città': ['Roma', 'Milano', 'Torino']

}

df = pd.DataFrame(data)

print(df)


# Creazione di un DataFrame da un array NumPy

array = np.array([[1, 2, 3], [4, 5, 6]])

df = pd.DataFrame(array, columns=['A', 'B', 'C'])
```

```
print(df)
```

Operazioni sui DataFrame

Pandas offre una vasta gamma di operazioni per manipolare i DataFrame.

```
# Selezione di colonne

print(df['Nome'])


# Selezione di righe tramite indicizzazione

print(df.iloc[1]) # Seconda riga


# Selezione di righe tramite etichette

print(df.loc[0]) # Prima riga

# Filtraggio dei dati

print(df[df['Età'] > 25])


# Aggiunta di nuove colonne

df['Reddito'] = [50000, 60000, 70000]

print(df)


# Rimozione di colonne
```

```
df.drop('Reddito', axis=1, inplace=True)

print(df)
```

Operazioni avanzate

Pandas include molte funzioni avanzate per l'analisi dei dati.

```
# Raggruppamento dei dati

grouped = df.groupby('Città').mean()

print(grouped)


# Merge di DataFrame

data_1 = {'ID': [1, 2, 3], 'Nome': ['Alice', 'Bob', 'Clara']}

data_2 = {'ID': [1, 2, 4], 'Età': [25, 30, 28]}

df1 = pd.DataFrame(data_1)

df2 = pd.DataFrame(data_2)

merged_df = pd.merge(df1, df2, on='ID', how='inner')

print(merged_df)


# Pivot table

pivot_table = df.pivot_table(values='Età', index='Nome',
```

```
columns='Città', aggfunc='mean')

print(pivot_table)
```

Lettura e scrittura di file

Pandas facilita la lettura e la scrittura di dati da e verso vari formati di file, come CSV, Excel, JSON e SQL.

```
# Lettura di un file CSV

df = pd.read_csv('file.csv')

print(df)


# Scrittura di un DataFrame in un file CSV

df.to_csv('output.csv', index=False)


# Lettura di un file Excel

df = pd.read_excel('file.xlsx', sheet_name='Sheet1')

print(df)


# Scrittura di un DataFrame in un file Excel

df.to_excel('output.xlsx', sheet_name='Sheet1', index=False)


# Lettura di un file JSON
```

```
df = pd.read_json('file.json')

print(df)


# Scrittura di un DataFrame in un file JSON

df.to_json('output.json')
```

Applicazioni di NumPy e Pandas

Analisi dei dati

NumPy e Pandas sono ampiamente utilizzati per l'analisi dei dati. Con NumPy, si possono eseguire calcoli numerici efficienti, mentre Pandas offre strumenti per la manipolazione e l'analisi di dati strutturati.

Esempio: Analisi esplorativa dei dati

Supponiamo di avere un dataset CSV contenente dati sulle vendite e vogliamo analizzare le vendite per regione.

```
import pandas as pd


# Lettura del dataset

df = pd.read_csv('sales_data.csv')
```

```
# Visualizzazione delle prime righe del dataset

print(df.head())


# Statistiche descrittive

print(df.describe())


# Vendite totali per regione

vendite_per_regione = df.groupby('Regione')['Vendite'].sum()

print(vendite_per_regione)


# Visualizzazione delle vendite per regione

vendite_per_regione.plot(kind='bar')
```

Machine Learning

NumPy e Pandas sono spesso utilizzati insieme a librerie di machine learning come Scikit-Learn per la preparazione dei dati.

Esempio: Preparazione dei dati per il machine learning

Supponiamo di avere un dataset con caratteristiche e etichette, e vogliamo prepararlo per l'addestramento di un modello di machine learning.

```
import pandas as pd
```

```
from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler


# Lettura del dataset

df = pd.read_csv('data.csv')


# Selezione delle caratteristiche (X) e delle etichette (y)

X = df.drop('Etichetta', axis=1)

y = df['Etichetta']


# Suddivisione dei dati in set di addestramento e di test

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)


# Standardizzazione delle caratteristiche

scaler = StandardScaler()

X_train_scaled = scaler.fit_transform(X_train)

X_test_scaled = scaler.transform(X_test)
```

NumPy e Pandas sono due delle librerie più potenti e versatili per la data science con Python. NumPy fornisce un supporto essenziale per l'elaborazione numerica e le operazioni su array multidimensionali, mentre Pandas offre strumenti avanzati per la manipolazione e l'analisi

dei dati strutturati. Con una combinazione di queste librerie, è possibile affrontare una vasta gamma di compiti di data science, dall'analisi esplorativa dei dati alla preparazione dei dati per il machine learning. La padronanza di queste librerie è fondamentale per chiunque voglia intraprendere una carriera nella data science.

10.2 ANALISI DEI DATI

Pandas è una libreria potente e versatile per l'analisi dei dati in Python. In questa sezione, esploreremo varie tecniche di analisi dei dati utilizzando Pandas, inclusa la manipolazione, l'esplorazione e la visualizzazione dei dati.

Importazione e ispezione dei dati

Il primo passo nell'analisi dei dati è importare i dati da una fonte esterna e ispezionarli per comprenderne la struttura e il contenuto.

Importazione dei dati

Pandas supporta la lettura di dati da vari formati di file, come CSV, Excel, JSON e SQL.

```
import pandas as pd


# Lettura di un file CSV

df = pd.read_csv('file.csv')
```

```
# Lettura di un file Excel

df = pd.read_excel('file.xlsx', sheet_name='Sheet1')


# Lettura di un file JSON

df = pd.read_json('file.json')
```

Ispezione dei dati

Dopo aver importato i dati, è importante ispezionarli per ottenere una comprensione iniziale della loro struttura e del loro contenuto.

```
# Visualizzazione delle prime 5 righe del DataFrame

print(df.head())


# Visualizzazione delle ultime 5 righe del DataFrame

print(df.tail())


# Informazioni sul DataFrame (numero di righe e colonne, tipi
di
dati, memoria utilizzata)

print(df.info())


# Statistiche descrittive del DataFrame
```

```
print(df.describe())
```

Pulizia dei dati

La pulizia dei dati è un passaggio cruciale nell'analisi dei dati. Spesso, i dati contengono valori mancanti, duplicati o errati che devono essere gestiti.

Gestione dei valori mancanti

Pandas fornisce vari metodi per gestire i valori mancanti.

```
# Identificazione dei valori mancanti

print(df.isnull().sum())


# Rimozione delle righe con valori mancanti

df.dropna(inplace=True)


# Sostituzione dei valori mancanti con un valore specifico

df.fillna(0, inplace=True)


# Sostituzione dei valori mancanti con la media della colonna

df.fillna(df.mean(), inplace=True)
```

Rimozione dei duplicati

I dati duplicati possono distorcere i risultati dell'analisi e devono essere rimossi.

```
# Identificazione dei duplicati

print(df.duplicated().sum())


# Rimozione dei duplicati

df.drop_duplicates(inplace=True)
```

Manipolazione dei dati

Pandas offre una serie di strumenti per manipolare i dati, inclusa la selezione, il filtraggio, l'ordinamento e l'aggiunta di nuove colonne.

Selezione e filtraggio dei dati

È possibile selezionare colonne specifiche o filtrare righe in base a condizioni.

```
# Selezione di una singola colonna

print(df['colonna'])
```

```
# Selezione di più colonne

print(df[['colonna1', 'colonna2']])


# Filtraggio delle righe in base a una condizione

print(df[df['colonna'] > 10])


# Filtraggio delle righe in base a più condizioni

print(df[(df['colonna1'] > 10) & (df['colonna2'] <
20)])
```

Ordinamento dei dati

Pandas permette di ordinare i dati in base a una o più colonne.

```
# Ordinamento in base a una colonna

df.sort_values(by='colonna', inplace=True)


# Ordinamento in base a più colonne

df.sort_values(by=['colonna1', 'colonna2'], ascending=[True,
False], inplace=True)
```

Aggiunta di nuove colonne

È possibile aggiungere nuove colonne al DataFrame, calcolate a partire dalle colonne esistenti.

```
# Aggiunta di una nuova colonna calcolata

df['nuova_colonna'] = df['colonna1'] + df['colonna2']
```

Analisi esplorativa dei dati (EDA)

L'analisi esplorativa dei dati (EDA) è il processo di sintesi delle caratteristiche principali di un dataset, spesso con metodi visivi. Pandas offre varie funzioni per l'EDA.

Statistiche descrittive

Pandas fornisce funzioni per calcolare statistiche descrittive come la media, la mediana e la deviazione standard.

```
# Media

print(df['colonna'].mean())


# Mediana

print(df['colonna'].median())


# Deviazione standard

print(df['colonna'].std())
```

```
# Statistiche descrittive per tutte le colonne

print(df.describe())
```

Visualizzazione dei dati

Pandas si integra bene con librerie di visualizzazione come Matplotlib e Seaborn per creare grafici e visualizzare i dati.

```
import matplotlib.pyplot as plt

import seaborn as sns

# Istogramma

df['colonna'].hist()

plt.show()


# Grafico a barre

df['colonna'].value_counts().plot(kind='bar')

plt.show()


# Grafico a dispersione

df.plot.scatter(x='colonna1', y='colonna2')
```

```
plt.show()


# Heatmap delle correlazioni

correlazioni = df.corr()

sns.heatmap(correlazioni, annot=True)

plt.show()
```

Raggruppamento e aggregazione

Pandas permette di raggruppare i dati e calcolare statistiche aggregate per ciascun gruppo.

Raggruppamento dei dati

Il metodo `groupby` di Pandas permette di raggruppare i dati in base a una o più colonne.

```
# Raggruppamento e calcolo della media

media_per_gruppo = df.groupby('colonna').mean()

print(media_per_gruppo)


# Raggruppamento e calcolo della somma

somma_per_gruppo = df.groupby('colonna').sum()

print(somma_per_gruppo)
```

```
# Raggruppamento e calcolo di statistiche multiple

statistiche_per_gruppo = df.groupby('colonna').agg(['mean',
'sum', 'count'])

print(statistiche_per_gruppo)
```

Unione di DataFrame

Pandas offre diversi metodi per unire DataFrame, inclusi join, merge e concat.

Join e merge

Il metodo `merge` permette di unire due DataFrame in base a una o più colonne chiave.

```
# Creazione di due DataFrame di esempio

df1 = pd.DataFrame({'ID': [1, 2, 3], 'Nome': ['Alice', 'Bob',
'Clara']})

df2 = pd.DataFrame({'ID': [1, 2, 4], 'Età': [25, 30, 28]})


# Join interno (inner join)

merge_inner = pd.merge(df1, df2, on='ID', how='inner')

print(merge_inner)


# Join esterno (outer join)
```

```
merge_outer = pd.merge(df1, df2, on='ID', how='outer')

print(merge_outer)
```

Concatenazione

Il metodo `concat` permette di concatenare due o più DataFrame lungo un asse specifico.

```
# Creazione di due DataFrame di esempio

df1 = pd.DataFrame({'Nome': ['Alice', 'Bob'], 'Età': [25, 30]})

df2 = pd.DataFrame({'Nome': ['Clara', 'David'], 'Età': [35, 40]})


# Concatenazione lungo l'asse delle righe

concat_righe = pd.concat([df1, df2], axis=0)

print(concat_righe)


# Concatenazione lungo l'asse delle colonne

df3 = pd.DataFrame({'Salario': [50000, 60000, 70000, 80000]})

concat_colonne = pd.concat([concat_righe, df3], axis=1)

print(concat_colonne)
```

Trasformazione dei dati

Pandas permette di applicare funzioni di trasformazione ai dati, come la normalizzazione e la

standardizzazione.

Normalizzazione

La normalizzazione è il processo di scalare i dati in modo che siano compresi in un intervallo specifico, ad esempio tra 0 e 1.

```
# Normalizzazione di una colonna

df['colonna_normalizzata'] = (df['colonna'] -
df['colonna'].min())      /      (df['colonna'].max()      -
df['colonna'].min())

print(df['colonna_normalizzata'])
```

Standardizzazione

La standardizzazione è il processo di scalare i dati in modo che abbiano una media di 0 e una deviazione standard di 1.

```
# Standardizzazione di una colonna

df['colonna_standardizzata'] = (df['colonna'] -
df['colonna'].mean()) / df['colonna'].std()

print(df['colonna_standardizzata'])
```

Gestione delle date e dei tempi

Pandas offre strumenti avanzati per lavorare con dati temporali.

Conversione delle date

Il metodo `pd.to_datetime` permette di convertire stringhe di date in oggetti datetime.

```
# Conversione di una colonna di stringhe di date

df['data'] = pd.to_datetime(df['data'])

print(df['data'])
```

Estrazione di componenti temporali

È possibile estrarre componenti temporali come l'anno, il mese e il giorno dagli oggetti datetime.

```
# Estrazione dell'anno, del mese e del giorno

df['anno'] = df['data'].dt.year

df['mese'] = df['data'].dt.month

df['giorno'] = df['data'].dt.day

print(df[['data', 'anno', 'mese', 'giorno']])
```

Filtraggio dei dati temporali

È possibile filtrare i dati in base a intervalli temporali.

```
# Filtraggio dei dati in base a un intervallo temporale

start_date = '2022-01-01'

end_date = '2022-12-31'

df_filtrato = df[(df['data'] >= start_date) &
(df['data'] <= end_date)]

print(df_filtrato)
```

Pandas è una libreria potente e versatile per l'analisi dei dati in Python. Offre una vasta gamma di strumenti per la manipolazione, la pulizia, l'esplorazione e la visualizzazione dei dati, rendendo il processo di analisi dei dati più efficiente e intuitivo. Con una comprensione approfondita delle tecniche di analisi dei dati utilizzando Pandas, è possibile affrontare una vasta gamma di compiti di data science, dalle analisi esplorative dei dati alla preparazione dei dati per il machine learning. La padronanza di queste tecniche è fondamentale per chiunque voglia intraprendere una carriera nella data science.

10.3 VISUALIZZAZIONE DEI DATI CON MATPLOTLIB E SEABORN

La visualizzazione dei dati è una componente essenziale della data science. Permette di esplorare i dati, identificare pattern e tendenze, e comunicare i risultati in modo chiaro ed efficace. In questa sezione, esploreremo due delle librerie più popolari per la visualizzazione dei dati in Python: Matplotlib e Seaborn.

Matplotlib

Matplotlib è una libreria di plotting 2D che offre un'ampia gamma di funzionalità per creare grafici di alta qualità. È estremamente flessibile e può essere utilizzata per creare una varietà di visualizzazioni.

Installazione

Per installare Matplotlib, si utilizza `pip`.

```
pip install matplotlib
```

Creazione di grafici di base

Matplotlib permette di creare grafici di vario tipo, come grafici a linee, grafici a barre, istogrammi, scatter plot, e molto altro.

```
import matplotlib.pyplot as plt


# Dati di esempio

x = [1, 2, 3, 4, 5]

y = [10, 20, 25, 30, 40]


# Grafico a linee

plt.plot(x, y)

plt.xlabel('X-axis')

plt.ylabel('Y-axis')

plt.title('Grafico a linee')

plt.show()
```

Istogrammi

Gli istogrammi sono utilizzati per visualizzare la distribuzione di un insieme di dati.

```
# Dati di esempio

import numpy as np

dati = np.random.randn(1000)


# Istogramma

plt.hist(dati, bins=30)

plt.xlabel('Valore')

plt.ylabel('Frequenza')

plt.title('Istogramma')

plt.show()
```

Scatter plot

Gli scatter plot sono utili per visualizzare la relazione tra due variabili.

```
# Dati di esempio

x = np.random.rand(50)

y = np.random.rand(50)


# Scatter plot

plt.scatter(x, y)

plt.xlabel('X-axis')
```

```
plt.ylabel('Y-axis')

plt.title('Scatter plot')

plt.show()
```

Grafici multipli

Matplotlib permette di creare grafici multipli in una singola figura utilizzando `subplot`.

```
# Dati di esempio

x = np.linspace(0, 10, 100)

y1 = np.sin(x)

y2 = np.cos(x)

# Grafici multipli

plt.figure(figsize=(10, 5))


plt.subplot(1, 2, 1)

plt.plot(x, y1)

plt.title('Sine wave')


plt.subplot(1, 2, 2)

plt.plot(x, y2)
```

```
plt.title('Cosine wave')


plt.show()
```

Seaborn

Seaborn è una libreria di visualizzazione dei dati basata su Matplotlib, che offre un'interfaccia di alto livello per creare grafici attraenti e informativi. Seaborn è particolarmente utile per la visualizzazione di dati statistici e per la creazione di grafici complessi con poche righe di codice.

Installazione

Per installare Seaborn, si utilizza `pip`.

```
pip install seaborn
```

Creazione di grafici di base

Seaborn semplifica la creazione di grafici comuni come grafici a linee, grafici a barre, scatter plot, istogrammi e altro ancora.

```
import seaborn as sns


# Dati di esempio

dati = sns.load_dataset('iris')
```

```
# Scatter plot con Seaborn

sns.scatterplot(x='sepal_length', y='sepal_width', data=dati)

plt.title('Scatter plot con Seaborn')

plt.show()
```

Grafici a barre

Seaborn permette di creare grafici a barre con funzioni specifiche come `barplot`.

```
# Grafico a barre con Seaborn

sns.barplot(x='species', y='sepal_length', data=dati)

plt.title('Grafico a barre con Seaborn')

plt.show()
```

Istogrammi e distribuzioni

Seaborn offre funzioni per creare istogrammi e visualizzare la distribuzione dei dati.

```
# Istogramma con Seaborn

sns.histplot(dati['sepal_length'], bins=20, kde=True)

plt.title('Istogramma con Seaborn')
```

```
plt.show()
```

Pair plot

Il `pairplot` di Seaborn è un modo eccellente per visualizzare le relazioni tra tutte le coppie di variabili in un DataFrame.

```
# Pair plot con Seaborn

sns.pairplot(dati, hue='species')

plt.title('Pair plot con Seaborn')

plt.show()
```

Heatmap

Le heatmap sono utili per visualizzare matrici di correlazione o altri tipi di dati bidimensionali.

```
# Matrice di correlazione

correlazioni = dati.corr()


# Heatmap con Seaborn

sns.heatmap(correlazioni, annot=True, cmap='coolwarm')
```

```
plt.title('Heatmap con Seaborn')

plt.show()
```

Box plot

I box plot sono utilizzati per visualizzare la distribuzione dei dati e identificare i valori anomali.

```
# Box plot con Seaborn

sns.boxplot(x='species', y='sepal_length', data=dati)

plt.title('Box plot con Seaborn')

plt.show()
```

Esempio pratico: Analisi dei dati con Matplotlib e Seaborn

Supponiamo di avere un dataset di vendite e vogliamo analizzare i dati e visualizzare i risultati. Useremo sia Matplotlib che Seaborn per creare varie visualizzazioni.

```
import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns
```

```
# Creazione di un DataFrame di esempio

dati = {

 'Mese': ['Gennaio', 'Febbraio', 'Marzo', 'Aprile', 'Maggio',
'Giugno'],

 'Vendite': [200, 220, 250, 230, 270, 300],

 'Spese': [150, 160, 170, 155, 180, 200]

}

df = pd.DataFrame(dati)



# Grafico a linee delle vendite e delle spese

plt.figure(figsize=(10, 5))

plt.plot(df['Mese'], df['Vendite'], label='Vendite',
marker='o')

plt.plot(df['Mese'], df['Spese'], label='Spese', marker='o')

plt.xlabel('Mese')

plt.ylabel('Valore')

plt.title('Vendite e Spese Mensili')

plt.legend()

plt.grid(True)

plt.show()



# Grafico a barre delle vendite

plt.figure(figsize=(10, 5))
```

```
sns.barplot(x='Mese', y='Vendite', data=df)
plt.xlabel('Mese')
plt.ylabel('Vendite')
plt.title('Vendite Mensili')
plt.show()

# Scatter plot delle vendite e delle spese
plt.figure(figsize=(10, 5))
sns.scatterplot(x='Vendite', y='Spese', data=df)
plt.xlabel('Vendite')
plt.ylabel('Spese')
plt.title('Vendite vs Spese')
plt.show()

# Heatmap delle correlazioni
correlazioni = df.corr()
plt.figure(figsize=(8, 6))
sns.heatmap(correlazioni, annot=True, cmap='coolwarm')
plt.title('Correlazione tra Vendite e Spese')
plt.show()
```

La visualizzazione dei dati è un aspetto cruciale della data science, poiché permette di esplorare i dati, identificare pattern e tendenze, e comunicare i risultati in modo efficace. Matplotlib e Seaborn sono due delle librerie più potenti e versatili per la visualizzazione dei dati in Python. Matplotlib offre un'ampia gamma di funzionalità per creare grafici di alta qualità, mentre Seaborn semplifica la creazione di grafici complessi e attraenti con poche righe di codice. Con una buona padronanza di queste librerie, è possibile creare visualizzazioni che rendono i dati più comprensibili e utili per l'analisi e la comunicazione dei risultati.

CAPITOLO 11: MACHINE LEARNING CON PYTHON

11.1 Introduzione al Machine Learning

Il machine learning (ML) è un ramo dell'intelligenza artificiale (AI) che si concentra sulla costruzione di sistemi che possono apprendere dai dati e fare previsioni o decisioni senza essere esplicitamente programmati per ogni compito. Questo capitolo fornisce un'introduzione ai concetti fondamentali del machine learning, esplorando i diversi tipi di machine learning, il flusso di lavoro tipico per lo sviluppo di modelli ML, e alcuni esempi di applicazioni del machine learning.

Concetti Fondamentali del Machine Learning

Il machine learning si basa sull'idea che i sistemi possono apprendere dai dati, identificare pattern e prendere decisioni con un intervento umano minimo. Alcuni concetti chiave del machine learning includono:

1. **Dati**:
    - Il machine learning richiede dati su cui allenare i modelli. I dati possono essere strutturati (come i dati dei fogli di calcolo) o non strutturati (come immagini e testo).

2. **Modello**:
    - Un modello è un'entità matematica che fa previsioni o classificazioni basate sui dati. Il modello viene addestrato sui dati storici per apprendere le relazioni tra le variabili.
3. **Caratteristiche (Features)**:
    - Le caratteristiche sono le variabili di input utilizzate dal modello per fare previsioni. Ad esempio, in un modello di previsione del prezzo delle case, le caratteristiche potrebbero includere la superficie, il numero di stanze, la localizzazione, ecc.
4. **Etichetta (Label)**:
    - L'etichetta è la variabile di output che il modello cerca di prevedere. Nell'esempio del prezzo delle case, l'etichetta sarebbe il prezzo della casa.
5. **Algoritmi**:
    - Gli algoritmi di machine learning sono le tecniche matematiche utilizzate per costruire i modelli. Alcuni esempi comuni includono regressione lineare, decision tree, reti neurali, e support vector machines (SVM).

Tipi di Machine Learning

Il machine learning può essere suddiviso in diverse categorie principali in base al tipo di apprendimento e ai dati disponibili:

1. **Apprendimento Supervisionato**:
    - Nell'apprendimento supervisionato, il modello viene addestrato su un dataset che include sia le caratteristiche che le etichette. L'obiettivo è apprendere una funzione che mappa le caratteristiche alle etichette.

- **Esempi di algoritmi supervisionati**: Regressione lineare, regressione logistica, decision tree, support vector machines (SVM), k-nearest neighbors (KNN), reti neurali.
- **Esempi di applicazioni**: Classificazione delle email come spam o non spam, previsione del prezzo delle case, riconoscimento delle immagini.

2. **Apprendimento Non Supervisionato**:
   - Nell'apprendimento non supervisionato, il modello viene addestrato solo con le caratteristiche, senza etichette. L'obiettivo è identificare pattern o strutture nei dati.
   - **Esempi di algoritmi non supervisionati**: K-means clustering, principal component analysis (PCA), t-distributed stochastic neighbor embedding (t-SNE), reti neurali autoencoder.
   - **Esempi di applicazioni**: Segmentazione dei clienti, riduzione della dimensionalità, rilevamento di anomalie.
3. **Apprendimento per Rinforzo**:
   - Nell'apprendimento per rinforzo, un agente apprende a fare decisioni attraverso prove ed errori, ricevendo feedback sotto forma di ricompense o penalità. L'obiettivo è massimizzare la ricompensa cumulativa.
   - **Esempi di algoritmi di apprendimento per rinforzo**: Q-learning, deep Q-networks (DQN), policy gradient methods.
   - **Esempi di applicazioni**: Giochi, robotica, sistemi di raccomandazione.
4. **Apprendimento Semi-Supervisionato**:
   - L'apprendimento semi-supervisionato utilizza una combinazione di dati etichettati e non etichettati per addestrare il modello. Questo

approccio è utile quando sono disponibili pochi dati etichettati e molti dati non etichettati.
    - **Esempi di applicazioni**: Classificazione delle immagini, elaborazione del linguaggio naturale.

Flusso di Lavoro Tipico del Machine Learning

Il flusso di lavoro del machine learning consiste in una serie di passaggi sistematici per sviluppare, addestrare e valutare un modello di machine learning. Di seguito viene descritto un flusso di lavoro tipico:

1. **Raccolta dei Dati**:
    - Il primo passo nel machine learning è raccogliere i dati pertinenti al problema che si vuole risolvere. I dati possono provenire da varie fonti come database, file CSV, API, e sensori.
2. **Pulizia e Preparazione dei Dati**:
    - I dati grezzi spesso contengono valori mancanti, outlier, e rumore che devono essere gestiti prima di addestrare il modello. Questo passaggio include l'imputazione dei valori mancanti, la rimozione degli outlier, la normalizzazione/scalatura dei dati e la trasformazione delle variabili.
3. **Selezione e Ingegneria delle Caratteristiche**:
    - La selezione delle caratteristiche comporta la scelta delle variabili più rilevanti per il problema. L'ingegneria delle caratteristiche comporta la creazione di nuove caratteristiche a partire dai dati esistenti per migliorare le prestazioni del modello.
4. **Divisione del Dataset**:
    - Il dataset viene diviso in set di addestramento e set di test. Il set di addestramento viene utilizzato

per addestrare il modello, mentre il set di test viene utilizzato per valutare le prestazioni del modello. Una divisione comune è 80% per l'addestramento e 20% per il test.

5. **Selezione dell'Algoritmo di Machine Learning**:
   - La scelta dell'algoritmo dipende dal tipo di problema (regressione, classificazione, clustering) e dalla natura dei dati. Si possono provare vari algoritmi per determinare quale funziona meglio.
6. **Addestramento del Modello**:
   - Il modello viene addestrato utilizzando il set di addestramento. Questo comporta l'utilizzo dell'algoritmo scelto per apprendere le relazioni tra le caratteristiche e le etichette.
7. **Valutazione del Modello**:
   - Le prestazioni del modello vengono valutate utilizzando il set di test. Le metriche di valutazione comuni includono accuratezza, precisione, recall, F1-score per i modelli di classificazione, e errore quadratico medio (MSE), errore assoluto medio (MAE) per i modelli di regressione.
8. **Ottimizzazione e Tuning del Modello**:
   - Il modello può essere migliorato ottimizzando i suoi iperparametri, utilizzando tecniche come la cross-validation e la ricerca a griglia (grid search). L'ingegneria delle caratteristiche e l'aggiunta di più dati possono anche migliorare le prestazioni del modello.
9. **Deploy del Modello**:
   - Una volta che il modello ha raggiunto prestazioni soddisfacenti, può essere deployato in un ambiente di produzione dove può fare previsioni su nuovi dati.
10. **Monitoraggio e Manutenzione**:

- Dopo il deploy, è importante monitorare le prestazioni del modello nel tempo e fare manutenzione periodica per garantire che il modello rimanga accurato e rilevante.

Esempi di Applicazioni del Machine Learning

Il machine learning ha una vasta gamma di applicazioni in diversi settori. Di seguito sono riportati alcuni esempi di come il machine learning viene utilizzato in vari ambiti:

1. **Finanza**:
   - **Rilevamento delle frodi**: Utilizzo di modelli di classificazione per identificare transazioni fraudolente.
   - **Trading algoritmico**: Utilizzo di modelli predittivi per sviluppare strategie di trading automatiche.
   - **Valutazione del rischio di credito**: Utilizzo di modelli di regressione e classificazione per valutare il rischio di credito dei richiedenti.
2. **Sanità**:
   - **Diagnostica medica**: Utilizzo di modelli di machine learning per diagnosticare malattie a partire da immagini mediche o dati clinici.
   - **Predizione delle malattie**: Utilizzo di modelli predittivi per identificare pazienti a rischio di sviluppare determinate malattie.
   - **Medicina personalizzata**: Utilizzo di algoritmi di clustering per creare piani di trattamento personalizzati basati sui dati genetici e clinici dei pazienti.
3. **Marketing e Vendite**:

- **Sistemi di raccomandazione**: Utilizzo di algoritmi di machine learning per raccomandare prodotti o servizi agli utenti basati sulle loro preferenze e comportamenti passati.
- **Segmentazione dei clienti**: Utilizzo di algoritmi di clustering per segmentare i clienti in gruppi omogenei per strategie di marketing mirate.
- **Previsione delle vendite**: Utilizzo di modelli di regressione per prevedere le vendite future basate su dati storici e fattori di mercato.

4. **Settore automobilistico**:
   - **Veicoli autonomi**: Utilizzo di algoritmi di machine learning per la percezione, la pianificazione del percorso e il controllo dei veicoli autonomi.
   - **Manutenzione predittiva**: Utilizzo di modelli predittivi per identificare guasti potenziali nei veicoli prima che si verifichino.
5. **Settore manifatturiero**:
   - **Ottimizzazione della catena di approvvigionamento**: Utilizzo di algoritmi di machine learning per ottimizzare i processi di approvvigionamento e ridurre i costi.
   - **Controllo della qualità**: Utilizzo di modelli di machine learning per identificare difetti nei prodotti durante il processo di produzione.
6. **Elaborazione del linguaggio naturale (NLP)**:
   - **Analisi del sentiment**: Utilizzo di modelli di NLP per analizzare il sentiment nei testi, come recensioni dei prodotti o commenti sui social media.
   - **Traduzione automatica**: Utilizzo di reti neurali per tradurre automaticamente testi da una lingua all'altra.
   - **Chatbot e assistenti virtuali**: Utilizzo di modelli di NLP per sviluppare chatbot e assistenti virtuali

che possono interagire con gli utenti in modo naturale.

Il machine learning è una tecnologia potente che sta trasformando vari settori, permettendo di prendere decisioni migliori e di automatizzare processi complessi. Comprendere i concetti fondamentali del machine learning, i diversi tipi di apprendimento, e il flusso di lavoro tipico è essenziale per chiunque voglia intraprendere una carriera in questo campo. Nel prossimo capitolo, esploreremo le principali librerie di machine learning in Python, concentrandoci su Scikit-learn, una delle librerie più popolari e versatili per l'implementazione di algoritmi di machine learning.

11.2 LIBRERIE PRINCIPALI: SCIKIT-LEARN

Scikit-learn è una delle librerie più popolari e versatili per l'implementazione di algoritmi di machine learning in Python. Questa libreria è costruita sopra altre librerie fondamentali come NumPy, SciPy e Matplotlib, e fornisce strumenti efficienti e semplici da utilizzare per l'analisi dei dati e il machine learning.

Panoramica di Scikit-learn

Scikit-learn offre una vasta gamma di algoritmi di machine learning per la classificazione, la regressione, il clustering e la riduzione della dimensionalità. Oltre agli algoritmi, Scikit-learn include strumenti per la pre-elaborazione dei dati, la selezione delle caratteristiche, la valutazione dei modelli e l'ottimizzazione degli iperparametri.

Installazione di Scikit-learn

Per installare Scikit-learn, si utilizza il gestore di pacchetti `pip`.

```
pip install scikit-learn
```

Struttura di Scikit-learn

Scikit-learn è organizzato in moduli che coprono diversi aspetti del machine learning. Alcuni dei moduli principali includono:

- **sklearn.datasets**: Moduli per caricare dataset di esempio.
- **sklearn.preprocessing**: Moduli per la pre-elaborazione dei dati.
- **sklearn.model_selection**: Moduli per la divisione dei dati e la selezione dei modelli.
- **sklearn.linear_model**: Moduli per la regressione e la classificazione lineare.
- **sklearn.tree**: Moduli per gli alberi decisionali.
- **sklearn.ensemble**: Moduli per i metodi ensemble come Random Forest e Gradient Boosting.
- **sklearn.cluster**: Moduli per il clustering.
- **sklearn.decomposition**: Moduli per la riduzione della dimensionalità.
- **sklearn.metrics**: Moduli per la valutazione dei modelli.

Utilizzo di Scikit-learn

In questa sezione, esploreremo come utilizzare Scikit-learn per implementare vari algoritmi di machine learning. Seguiremo un flusso di lavoro tipico, che include la pre-elaborazione dei dati, la creazione e l'addestramento di modelli, la valutazione delle prestazioni e l'ottimizzazione degli iperparametri.

Caricamento dei dataset

Scikit-learn include vari dataset di esempio che possono essere utilizzati per testare e comprendere gli algoritmi di machine learning.

```
from sklearn.datasets import load_iris


# Caricamento del dataset Iris

iris = load_iris()

X, y = iris.data, iris.target


print(X.shape) # Dimensioni delle caratteristiche

print(y.shape) # Dimensioni delle etichette
```

Pre-elaborazione dei dati

La pre-elaborazione dei dati è un passaggio cruciale nel machine learning. Include attività come la scalatura delle caratteristiche, la codifica delle variabili categoriche e la gestione dei valori mancanti.

```
from sklearn.preprocessing import StandardScaler


# Standardizzazione delle caratteristiche

scaler = StandardScaler()

X_scaled = scaler.fit_transform(X)
```

```
print(X_scaled[:5])    #    Prime    cinque    righe    delle
caratteristiche
scalate
```

Divisione dei dati

Dividere i dati in set di addestramento e set di test è essenziale per valutare le prestazioni del modello.

```
from sklearn.model_selection import train_test_split


# Divisione dei dati in set di addestramento e di test

X_train, X_test, y_train, y_test = train_test_split(X_scaled,
y, test_size=0.2, random_state=42)


print(X_train.shape, X_test.shape) # Dimensioni dei set di
addestramento e di test
```

Regressione Lineare

La regressione lineare è uno degli algoritmi più semplici e utilizzati per problemi di regressione.

```
from sklearn.linear_model import LinearRegression
```

```
# Creazione e addestramento del modello di regressione lineare

model = LinearRegression()

model.fit(X_train, y_train)


# Previsione sui dati di test

y_pred = model.predict(X_test)


# Valutazione del modello

from sklearn.metrics import mean_squared_error


mse = mean_squared_error(y_test, y_pred)

print(f'Mean Squared Error: {mse}')
```

Regressione Logistica

La regressione logistica è un algoritmo di classificazione utilizzato per problemi binari e multi-classe.

```
from sklearn.linear_model import LogisticRegression


# Creazione e addestramento del modello di regressione logistica

model = LogisticRegression()

model.fit(X_train, y_train)
```

```
# Previsione sui dati di test

y_pred = model.predict(X_test)


# Valutazione del modello

from sklearn.metrics import accuracy_score,
classification_report


accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy}')


report = classification_report(y_test, y_pred)

print(report)
```

Alberi Decisionali

Gli alberi decisionali sono algoritmi non parametrici utilizzati per la classificazione e la regressione.

```
from sklearn.tree import DecisionTreeClassifier


# Creazione e addestramento del modello di albero decisionale

model = DecisionTreeClassifier()

model.fit(X_train, y_train)
```

```
# Previsione sui dati di test

y_pred = model.predict(X_test)


# Valutazione del modello

accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy}')


report = classification_report(y_test, y_pred)

print(report)
```

Metodi Ensemble

I metodi ensemble combinano più modelli di base per migliorare le prestazioni predittive.

Random Forest

La Random Forest è un metodo ensemble che combina più alberi decisionali.

```
from sklearn.ensemble import RandomForestClassifier


# Creazione e addestramento del modello Random Forest

model = RandomForestClassifier(n_estimators=100,
```

```
random_state=42)

model.fit(X_train, y_train)


# Previsione sui dati di test

y_pred = model.predict(X_test)


# Valutazione del modello

accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy}')


report = classification_report(y_test, y_pred)

print(report)
```

Gradient Boosting

Il Gradient Boosting è un metodo ensemble che costruisce modelli sequenziali per ridurre l'errore residuo.

```
from sklearn.ensemble import GradientBoostingClassifier


# Creazione e addestramento del modello Gradient Boosting

model = GradientBoostingClassifier(n_estimators=100,
learning_rate=0.1, random_state=42)
```

```
model.fit(X_train, y_train)


# Previsione sui dati di test

y_pred = model.predict(X_test)


# Valutazione del modello

accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy}')


report = classification_report(y_test, y_pred)

print(report)
```

Clustering

Il clustering è una tecnica di apprendimento non supervisionato utilizzata per raggruppare i dati in base alla somiglianza.

K-means

Il K-means è un algoritmo di clustering che suddivide i dati in k cluster distinti.

```
from sklearn.cluster import KMeans
```

```
# Creazione e addestramento del modello K-means
kmeans = KMeans(n_clusters=3, random_state=42)
kmeans.fit(X)

# Predizione dei cluster
clusters = kmeans.predict(X)

# Visualizzazione dei cluster
import matplotlib.pyplot as plt

plt.scatter(X[:, 0], X[:, 1], c=clusters, cmap='viridis')
plt.title('Clustering con K-means')
plt.xlabel('Caratteristica 1')
plt.ylabel('Caratteristica 2')
plt.show()
```

Riduzione della dimensionalità

La riduzione della dimensionalità è utilizzata per ridurre il numero di caratteristiche nei dati, migliorando l'efficienza computazionale e la visualizzazione.

PCA (Principal Component Analysis)

Il PCA è una tecnica di riduzione della dimensionalità che trasforma i dati in un nuovo spazio con meno dimensioni.

```
from sklearn.decomposition import PCA


# Creazione e addestramento del modello PCA

pca = PCA(n_components=2)

X_pca = pca.fit_transform(X)


# Visualizzazione dei dati trasformati

plt.scatter(X_pca[:, 0], X_pca[:, 1], c=y, cmap='viridis')

plt.title('Riduzione della dimensionalità con PCA')

plt.xlabel('Componente principale 1')

plt.ylabel('Componente principale 2')

plt.show()
```

Valutazione e selezione del modello

La valutazione e la selezione del modello sono passaggi cruciali per garantire che il modello scelto sia il migliore per il problema in questione.

Cross-validation

La cross-validation è una tecnica di valutazione che divide i dati in k-fold per validare il modello su diverse porzioni dei dati.

```
from sklearn.model_selection import cross_val_score


# Valutazione del modello con cross-validation

scores = cross_val_score(model, X, y, cv=5)

print(f'Cross-validation scores: {scores}')

print(f'Mean score: {scores.mean()}')
```

Grid Search

Il Grid Search è una tecnica di ottimizzazione degli iperparametri che esplora una griglia di possibili combinazioni di iperparametri.

```
from sklearn.model_selection import GridSearchCV


# Definizione dei parametri da ottimizzare

param_grid = {

 'n_estimators': [50, 100, 200],

 'max_depth': [None, 10, 20, 30]

}


# Creazione del Grid Search
```

```
grid_search =
GridSearchCV(RandomForestClassifier(random_state=42),
param_grid,
cv=5)

grid_search.fit(X_train, y_train)



# Migliori parametri trovati

print(f'Best parameters: {grid_search.best_params_}')

print(f'Best score: {grid_search.best_score_}')
```

Scikit-learn è una libreria potente e versatile per l'implementazione di algoritmi di machine learning in Python. Offre una vasta gamma di strumenti per la pre-elaborazione dei dati, la creazione e l'addestramento di modelli, la valutazione delle prestazioni e l'ottimizzazione degli iperparametri. Con una buona padronanza di Scikit-learn, è possibile affrontare una vasta gamma di problemi di machine learning e sviluppare modelli accurati e affidabili. Nel prossimo capitolo, esploreremo i passi specifici per creare, addestrare e valutare modelli di machine learning utilizzando Scikit-learn.

11.3 CREAZIONE DI MODELLI DI MACHINE LEARNING

In questa sezione, esploreremo in dettaglio il processo di creazione, addestramento e valutazione dei modelli di machine learning utilizzando Scikit-learn. Copriremo ogni passo del flusso di lavoro del machine learning, dalla preparazione dei dati alla selezione del modello, all'addestramento, alla valutazione e all'ottimizzazione.

Passo 1: Preparazione dei dati

Raccolta dei dati

Il primo passo nella creazione di un modello di machine learning è raccogliere i dati rilevanti per il problema da risolvere. I dati possono provenire da varie fonti, come file CSV, database, API o sensori. È essenziale assicurarsi che i dati raccolti siano di alta qualità e pertinenti all'obiettivo dell'analisi.

```
import pandas as pd


# Esempio di caricamento di un dataset da un file CSV

df = pd.read_csv('dataset.csv')
```

Esplorazione dei dati

L'esplorazione dei dati è un passaggio fondamentale per comprendere la struttura e le caratteristiche del dataset. Include la visualizzazione delle prime righe, la descrizione statistica, la verifica dei valori mancanti e la comprensione delle distribuzioni delle variabili.

```
# Visualizzazione delle prime righe del dataset

print(df.head())


# Descrizione statistica del dataset

print(df.describe())


# Verifica dei valori mancanti

print(df.isnull().sum())
```

Pulizia dei dati

La pulizia dei dati è essenziale per rimuovere o correggere i valori mancanti, gli outlier e gli errori nei dati. Questo passaggio può includere l'imputazione dei

valori mancanti, la rimozione degli outlier e la correzione dei dati errati.

```
# Imputazione dei valori mancanti con la media della colonna

df.fillna(df.mean(), inplace=True)


# Rimozione delle righe con valori mancanti

df.dropna(inplace=True)
```

Ingegneria delle caratteristiche

L'ingegneria delle caratteristiche comporta la creazione di nuove caratteristiche a partire dai dati esistenti, la trasformazione delle caratteristiche e la selezione delle caratteristiche più rilevanti. Questo passaggio può migliorare significativamente le prestazioni del modello.

```
# Creazione di una nuova caratteristica

df['nuova_caratteristica'] = df['caratteristica1'] *
df['caratteristica2']


# Selezione delle caratteristiche più rilevanti

X = df[['caratteristica1', 'caratteristica2',
'nuova_caratteristica']]

y = df['etichetta']
```

Divisione dei dati

Per valutare le prestazioni del modello, è essenziale dividere i dati in set di addestramento e set di test. Il set di addestramento viene utilizzato per addestrare il modello, mentre il set di test viene utilizzato per valutare le sue prestazioni su dati non visti.

```
from sklearn.model_selection import train_test_split



# Divisione dei dati in set di addestramento e di test

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Passo 2: Selezione del modello

Scelta dell'algoritmo di machine learning

La scelta dell'algoritmo di machine learning dipende dal tipo di problema da risolvere (regressione, classificazione, clustering) e dalla natura dei dati. Scikit-learn offre una vasta gamma di algoritmi per diversi tipi di problemi.

Modelli di Regressione

1. **Regressione Lineare**:
    - Utilizzata per problemi di regressione, dove l'obiettivo è prevedere una variabile continua.

```
from sklearn.linear_model import LinearRegression
# Creazione del modello di regressione lineare
model =LinearRegression()
```

1. **Regressione Ridge**:
    - Una variante della regressione lineare che include un termine di regolarizzazione per ridurre l'overfitting.

```
from sklearn.linear_model import Ridge


# Creazione del modello di regressione Ridge

model = Ridge(alpha=1.0)
```

Modelli di Classificazione

1. **Regressione Logistica**:
    - Utilizzata per problemi di classificazione binaria e multi-classe.

```
from sklearn.linear_model import LogisticRegression


# Creazione del modello di regressione logistica

model = LogisticRegression()
```

1. **Support Vector Machines (SVM)**:
    - Utilizzate per problemi di classificazione con margini massimi di separazione.

```
from sklearn.svm import SVC
# Creazione del modello SVM
model =SVC(kernel= 'linear')
```

Modelli Ensemble

1. **Random Forest**:
    - Un metodo ensemble che combina più alberi decisionali per migliorare le prestazioni predittive.

```
from sklearn.ensemble import RandomForestClassifier
# Creazione del modello Random Forest
model =RandomForestClassifier(n_estimators= 100,
random_state= 42)
```

1. **Gradient Boosting**:
    - Un metodo ensemble che costruisce modelli sequenziali per ridurre l'errore residuo.

```
from sklearn.ensemble import
GradientBoostingClassifier



# Creazione del modello Gradient Boosting

model = GradientBoostingClassifier(n_estimators=100,
```

```
learning_rate=0.1, random_state=42)
```

Modelli di Clustering

1. **K-means**:
   - Utilizzato per raggruppare i dati in k cluster distinti.

```
from sklearn.cluster import KMeans
# Creazione del modello K-means
model =KMeans(n_clusters= 3, random_state= 42)
```

1. **DBSCAN**:
   - Un algoritmo di clustering basato sulla densità che non richiede il numero di cluster come input.

```
from sklearn.cluster import DBSCAN
# Creazione del modello DBSCAN
model =DBSCAN(eps= 0.5, min_samples= 5)
```

Passo 3: Addestramento del modello

Addestramento del modello

Una volta scelto l'algoritmo, il passo successivo è addestrare il modello utilizzando il set di addestramento. Questo comporta l'utilizzo dell'algoritmo scelto per apprendere le relazioni tra le caratteristiche e le etichette.

```
# Addestramento del modello
```

```
model.fit(X_train, y_train)
```

Passo 4: Valutazione del modello

Previsione sui dati di test

Dopo aver addestrato il modello, è possibile utilizzare il set di test per fare previsioni e valutare le prestazioni del modello.

```
# Previsione sui dati di test

y_pred = model.predict(X_test)
```

Valutazione delle prestazioni

Le prestazioni del modello vengono valutate utilizzando varie metriche di valutazione. Le metriche di valutazione dipendono dal tipo di problema (regressione o classificazione).

Valutazione dei modelli di regressione

1. **Mean Squared Error (MSE)**:
    - La MSE misura l'errore quadratico medio tra le previsioni e i valori reali.

```
from sklearn.metrics import mean_squared_error
```

```
# Calcolo della MSE
mse =mean_squared_error(y_test, y_pred)
print( f'Mean Squared Error: {mse}')
```

1. **Mean Absolute Error (MAE)**:
    - La MAE misura l'errore assoluto medio tra le previsioni e i valori reali.

```
from sklearn.metrics import mean_absolute_error
# Calcolo della MAE
mae =mean_absolute_error(y_test, y_pred)
print( f'Mean Absolute Error: {mae}')
```

1. **R-squared (R^2)**:
    - L'R^2 misura la proporzione della varianza nei dati di output che è spiegata dal modello.

```
from sklearn.metrics import r2_score
# Calcolo dell'R²
r2 = r2_score(y_test, y_pred)
print( f'R-squared: {r2}')
```

Valutazione dei modelli di classificazione

1. **Accuracy**:
    - L'accuracy misura la proporzione di previsioni corrette sul totale delle previsioni.

```
from sklearn.metrics import accuracy_score


# Calcolo dell'accuracy

accuracy = accuracy_score(y_test, y_pred)

print(f'Accuracy: {accuracy}')
```

1. **Precision, Recall, e F1-score**:
    - La precision misura la proporzione di previsioni positive corrette rispetto al totale delle previsioni positive.
    - Il recall misura la proporzione di previsioni positive corrette rispetto al totale dei veri positivi.
    - L'F1-score è la media armonica tra precision e recall.

```
  from sklearn.metrics import precision_score, recall_score,
f1_score
  # Calcolo della precision, del recall e dell'F1-score
  precision   =precision_score(y_test,   y_pred,   average=
'weighted')
  recall =recall_score(y_test, y_pred, average= 'weighted')
  f1 = f1_score(y_test, y_pred, average= 'weighted')
  print( f'Precision: {precision}')
  print( f'Recall: {recall}')
  print( f'F1-score: {f1}')
```

1. **Classification Report**:
    - Il classification report fornisce una panoramica delle principali metriche di valutazione per ogni classe.

```
  from sklearn.metrics import classification_report
  # Generazione del classification report
  report =classification_report(y_test, y_pred)
  print(report)
```

1. **Confusion Matrix**:

- La confusion matrix mostra il numero di previsioni corrette e errate per ciascuna classe.

```
from sklearn.metrics import confusion_matrix
# Calcolo della confusion matrix
cm =confusion_matrix(y_test, y_pred)
print(cm)
# Visualizzazione della confusion matrix
import seaborn as sns
import matplotlib.pyplot as plt
sns.heatmap(cm, annot= True, fmt= 'd', cmap= 'Blues')
plt.xlabel( 'Predicted')
plt.ylabel( 'Actual')
plt.title( 'Confusion Matrix')
plt.show()
```

Passo 5: Ottimizzazione del modello

Cross-validation

La cross-validation è una tecnica di valutazione che divide i dati in k-fold per validare il modello su diverse porzioni dei dati. Questa tecnica aiuta a ridurre il rischio di overfitting e a ottenere una stima più accurata delle prestazioni del modello.

```
from sklearn.model_selection import cross_val_score


# Valutazione del modello con cross-validation

scores = cross_val_score(model, X, y, cv=5)

print(f'Cross-validation scores: {scores}')

print(f'Mean score: {scores.mean()}')
```

Grid Search

Il Grid Search è una tecnica di ottimizzazione degli iperparametri che esplora una griglia di possibili combinazioni di iperparametri per trovare la combinazione ottimale.

```
from sklearn.model_selection import GridSearchCV



# Definizione dei parametri da ottimizzare

param_grid = {

 'n_estimators': [50, 100, 200],

 'max_depth': [None, 10, 20, 30]

}



# Creazione del Grid Search

grid_search =
GridSearchCV(RandomForestClassifier(random_state=42),
param_grid,
cv=5)

grid_search.fit(X_train, y_train)



# Migliori parametri trovati

print(f'Best parameters: {grid_search.best_params_}')

print(f'Best score: {grid_search.best_score_}')
```

Random Search

Il Random Search è una tecnica di ottimizzazione degli iperparametri che esplora casualmente le combinazioni di iperparametri, piuttosto che esaminare ogni possibile combinazione.

```
from sklearn.model_selection import RandomizedSearchCV



# Definizione dei parametri da ottimizzare

param_dist = {

 'n_estimators': [50, 100, 200],

 'max_depth': [None, 10, 20, 30],

 'min_samples_split': [2, 5, 10],

 'min_samples_leaf': [1, 2, 4]

}



# Creazione del Random Search

random_search =
RandomizedSearchCV(RandomForestClassifier(random_state=42),
param_dist, n_iter=100, cv=5, random_state=42)

random_search.fit(X_train, y_train)



# Migliori parametri trovati

print(f'Best parameters: {random_search.best_params_}')

print(f'Best score: {random_search.best_score_}')
```

La creazione, l'addestramento e la valutazione dei modelli di machine learning richiede un approccio

sistematico e attento. Dalla preparazione dei dati alla selezione del modello, all'addestramento, alla valutazione e all'ottimizzazione, ogni passo è cruciale per garantire che il modello finale sia accurato e affidabile. Scikit-learn fornisce una vasta gamma di strumenti e algoritmi che facilitano l'implementazione di ogni fase del flusso di lavoro del machine learning. Con una buona padronanza di questi strumenti, è possibile sviluppare modelli di machine learning efficaci per una vasta gamma di applicazioni.

CAPITOLO 12: AUTOMAZIONE E SCRIPTING

12.1 Automazione delle attività con Python

Python è un linguaggio estremamente versatile e potente per l'automazione delle attività quotidiane. Grazie alla sua sintassi semplice e alla vasta gamma di librerie disponibili, Python è ideale per automatizzare processi ripetitivi, migliorare l'efficienza e ridurre il rischio di errori manuali. In questo capitolo, esploreremo come utilizzare Python per automatizzare diverse attività comuni, inclusa la gestione dei file, l'invio di email, l'interazione con le API e la manipolazione dei dati.

Gestione dei file

Uno dei compiti più comuni nell'automazione è la gestione dei file. Python offre moduli integrati come `os`, `shutil` e `glob` per lavorare con il file system, copiare, spostare, rinominare ed eliminare file e directory.

Lavorare con il modulo os

Il modulo `os` fornisce una serie di funzioni per interagire con il sistema operativo, permettendo di

eseguire operazioni come la navigazione nel file system e la manipolazione dei file.

```
import os


# Ottenere il percorso della directory corrente

current_directory = os.getcwd()

print(f"Directory corrente: {current_directory}")


# Cambiare directory

os.chdir('/path/to/directory')

print(f"Nuova directory corrente: {os.getcwd()}")


# Creare una nuova directory

os.mkdir('nuova_directory')

print("Directory creata")


# Rimuovere una directory

os.rmdir('nuova_directory')

print("Directory rimossa")


# Elenco dei file in una directory

files = os.listdir('.')

print(f"File nella directory corrente: {files}")
```

Copiare, spostare e rinominare file con il modulo shutil

Il modulo `shutil` fornisce funzioni per copiare, spostare e rinominare file e directory.

```
import shutil


# Copiare un file

shutil.copy('file_originale.txt', 'file_copia.txt')

print("File copiato")


# Spostare un file

shutil.move('file_copia.txt', '/path/to/destination')

print("File spostato")


# Rinominare un file

os.rename('file_originale.txt', 'file_rinominato.txt')

print("File rinominato")
```

Ricerca di file con il modulo glob

Il modulo `glob` permette di cercare file che corrispondono a un pattern specifico.

```
import glob


# Ricerca di file con estensione .txt

file_txt = glob.glob('*.txt')

print(f"File trovati: {file_txt}")
```

Invio di email

Automatizzare l'invio di email può essere molto utile per notifiche, report e altre comunicazioni periodiche. Python offre il modulo `smtplib` per inviare email tramite il protocollo SMTP.

```
import smtplib

from email.mime.text import MIMEText

from email.mime.multipart import MIMEMultipart


# Configurazione dell'email

sender_email = "tuo_email@example.com"

receiver_email = "destinatario@example.com"

password = "tuo_password"
```

```
# Creazione del messaggio

message = MIMEMultipart()

message["From"] = sender_email

message["To"] = receiver_email

message["Subject"] = "Oggetto dell'email"


# Corpo dell'email

body = "Questo è il corpo dell'email"

message.attach(MIMEText(body, "plain"))


# Connessione al server SMTP e invio dell'email

with smtplib.SMTP_SSL("smtp.example.com", 465) as server:

 server.login(sender_email, password)

 server.sendmail(sender_email, receiver_email,
message.as_string())


print("Email inviata con successo")
```

Interazione con le API

Le API (Application Programming Interface) permettono di interagire con applicazioni e servizi web per ottenere dati o eseguire operazioni. Python offre il

modulo `requests` per inviare richieste HTTP e interagire con le API.

```
import requests


# Esempio di richiesta GET

response = requests.get('https://api.example.com/dati')

if response.status_code == 200:

 data = response.json()

 print(data)

else:

 print(f"Errore nella richiesta: {response.status_code}")


# Esempio di richiesta POST

payload = {'chiave': 'valore'}

response = requests.post('https://api.example.com/invia',
json=payload)

if response.status_code == 201:

 print("Dati inviati con successo")

else:

 print(f"Errore nella richiesta: {response.status_code}")
```

Manipolazione dei dati

La manipolazione dei dati è una delle aree in cui Python eccelle grazie a librerie come `pandas` e `numpy`. Queste librerie permettono di caricare, analizzare e trasformare i dati in modo efficiente.

```
import pandas as pd


# Caricamento dei dati da un file CSV

df = pd.read_csv('dati.csv')

print(df.head())


# Filtraggio dei dati

df_filtrato = df[df['colonna'] > 10]

print(df_filtrato)


# Raggruppamento e aggregazione

df_aggregato = df.groupby('categoria').agg({'valore': 'sum'})

print(df_aggregato)


# Salvataggio dei dati in un nuovo file CSV

df_aggregato.to_csv('dati_aggregati.csv', index=False)

print("Dati salvati con successo")
```

Automazione delle attività con i task scheduler

Per eseguire script di automazione a intervalli regolari, è possibile utilizzare i task scheduler come `cron` su Unix/Linux o Task Scheduler su Windows.

Esempio di cron job su Unix/Linux

Per aggiungere un cron job su Unix/Linux, si utilizza il comando `crontab -e` per modificare il file crontab e aggiungere una nuova voce.

```
# Esempio di cron job che esegue uno script Python ogni giorno alle
8:00 AM

0 8 * * * /usr/bin/python3 /path/to/script.py
```

Esempio di Task Scheduler su Windows

Per creare un'attività pianificata su Windows, si utilizza l'Utilità di pianificazione (Task Scheduler) e si segue la procedura guidata per creare una nuova attività, specificando il percorso dello script Python e l'intervallo di esecuzione.

Automazione di operazioni sui fogli di calcolo

Lavorare con fogli di calcolo è un'attività comune che può essere automatizzata utilizzando librerie come

`openpyxl` per file Excel e `csv` per file CSV.

Lavorare con file Excel

```
import openpyxl

# Caricamento di un file Excel
workbook = openpyxl.load_workbook('file.xlsx')
sheet = workbook.active

# Lettura di valori da una cella
valore = sheet['A1'].value
print(f"Valore in A1: {valore}")

# Scrittura di valori in una cella
sheet['A2'] = "Nuovo Valore"
workbook.save('file_modificato.xlsx')
print("File Excel aggiornato e salvato")
```

Lavorare con file CSV

```
import csv

# Lettura di un file CSV
```

```
with open('file.csv', mode='r') as file:

 reader = csv.reader(file)

 for row in reader:

 print(row)



# Scrittura di un file CSV

with open('file_modificato.csv', mode='w', newline='') as
file:

 writer = csv.writer(file)

 writer.writerow(['Colonna1', 'Colonna2'])

 writer.writerow(['Valore1', 'Valore2'])

print("File CSV creato e salvato")
```

Python è uno strumento potente per automatizzare una vasta gamma di attività quotidiane. Grazie alla sua semplicità e alla ricchezza delle sue librerie, è possibile automatizzare la gestione dei file, l'invio di email, l'interazione con le API, la manipolazione dei dati e molto altro. Con l'uso di task scheduler, è possibile eseguire script di automazione a intervalli regolari, migliorando l'efficienza e riducendo il rischio di errori manuali. Iniziare con l'automazione delle attività con Python può portare a significativi miglioramenti in termini di produttività e precisione.

12.2 PARSING DI FILE E AUTOMAZIONE WEB

In questa sezione, esploreremo come utilizzare Python per il parsing di file e l'automazione di attività web. La capacità di estrarre e manipolare dati da vari tipi di file e di interagire con siti web è fondamentale per molte applicazioni di automazione. Python offre una vasta gamma di librerie per facilitare queste attività.

Parsing di file

Il parsing di file consiste nell'analisi e nell'estrazione di informazioni strutturate da file di testo, XML, JSON e altri formati. Python fornisce strumenti potenti per lavorare con questi tipi di file.

Parsing di file di testo

I file di testo sono uno dei formati di file più semplici e comuni. Python offre strumenti integrati per leggere e manipolare file di testo.

```
# Lettura di un file di testo

with open('file.txt', 'r') as file:
```

```
 contenuto = file.read()
 print(contenuto)

# Lettura riga per riga
with open('file.txt', 'r') as file:
 for riga in file:
 print(riga.strip())
```

Parsing di file CSV

Il formato CSV (Comma Separated Values) è ampiamente utilizzato per memorizzare dati tabulari. Python offre il modulo `csv` per leggere e scrivere file CSV.

```
import csv

# Lettura di un file CSV
with open('file.csv', 'r') as file:
 reader = csv.reader(file)
 for row in reader:
 print(row)

# Scrittura di un file CSV
```

```
with open('output.csv', 'w', newline='') as file:
 writer = csv.writer(file)
 writer.writerow(['Colonna1', 'Colonna2'])
 writer.writerow(['Valore1', 'Valore2'])
```

Parsing di file JSON

Il formato JSON (JavaScript Object Notation) è utilizzato per memorizzare e scambiare dati strutturati. Python offre il modulo `json` per lavorare con file JSON.

```
import json

# Lettura di un file JSON
with open('file.json', 'r') as file:
 dati = json.load(file)
 print(dati)

# Scrittura di un file JSON
dati = {'nome': 'Alice', 'età': 25}
with open('output.json', 'w') as file:
 json.dump(dati, file, indent=4)
```

Parsing di file XML

Il formato XML (eXtensible Markup Language) è utilizzato per rappresentare dati strutturati in un formato leggibile sia dall'uomo che dalla macchina. Python offre moduli come `xml.etree.ElementTree` per lavorare con file XML.

```
import xml.etree.ElementTree as ET


# Lettura di un file XML

tree = ET.parse('file.xml')

root = tree.getroot()


# Iterazione sugli elementi XML

for child in root:

 print(child.tag, child.attrib, child.text)


# Scrittura di un file XML

root = ET.Element('root')

child = ET.SubElement(root, 'child')

child.text = 'Questo è un testo'

tree = ET.ElementTree(root)

tree.write('output.xml')
```

Automazione web

L'automazione web consiste nell'interagire con siti web per estrarre dati, compilare form, cliccare su pulsanti e navigare tra le pagine. Python offre librerie potenti come `requests`, `BeautifulSoup` e `Selenium` per facilitare queste attività.

Web scraping con BeautifulSoup

BeautifulSoup è una libreria utilizzata per il parsing di documenti HTML e XML, rendendo facile l'estrazione di dati da pagine web.

Installazione di BeautifulSoup

Per installare BeautifulSoup, si utilizza `pip`.

```
pip install beautifulsoup4

pip install lxml
```

Utilizzo di BeautifulSoup per l'estrazione di dati

```
import requests
```

```
from bs4 import BeautifulSoup


# Invio di una richiesta HTTP alla pagina web

url = 'https://example.com'

response = requests.get(url)


# Parsing del contenuto HTML

soup = BeautifulSoup(response.content, 'lxml')


# Estrazione dei dati

titolo = soup.title.text

print(f'Titolo della pagina: {titolo}')


# Estrazione di tutti i link

for link in soup.find_all('a'):

 print(link.get('href'))
```

Automazione del browser con Selenium

Selenium è una libreria utilizzata per automatizzare i browser web. Permette di interagire con elementi delle pagine web come campi di input, pulsanti e link.

Installazione di Selenium

Per installare Selenium, si utilizza `pip`. È inoltre necessario scaricare il WebDriver appropriato per il browser scelto (ad esempio, ChromeDriver per Google Chrome).

```
pip install selenium
```

Utilizzo di Selenium per l'automazione del browser

```
from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.webdriver.common.keys import Keys


# Configurazione del WebDriver

driver =
webdriver.Chrome(executable_path='/path/to/chromedriver')


# Apertura di una pagina web

driver.get('https://example.com')


# Interazione con elementi della pagina

search_box = driver.find_element(By.NAME, 'q')
```

```
search_box.send_keys('Python')

search_box.send_keys(Keys.RETURN)


# Estrazione dei risultati

results = driver.find_elements(By.CSS_SELECTOR, 'h3')

for result in results:

 print(result.text)


# Chiusura del browser

driver.quit()
```

Automazione di attività web con APIs

Le API (Application Programming Interface) permettono di interagire con servizi web senza dover utilizzare il web scraping. Molti siti web e servizi offrono API per l'accesso ai dati.

Utilizzo delle API con il modulo requests

Il modulo `requests` permette di inviare richieste HTTP per interagire con le API.

```
import requests
```

```
# Esempio di richiesta GET

response = requests.get('https://api.example.com/dati')

if response.status_code == 200:

 dati = response.json()

 print(dati)

else:

 print(f'Errore nella richiesta: {response.status_code}')


# Esempio di richiesta POST

payload = {'chiave': 'valore'}

response = requests.post('https://api.example.com/invia',
json=payload)

if response.status_code == 201:

 print("Dati inviati con successo")

else:

 print(f'Errore nella richiesta: {response.status_code}')
```

Esempio pratico: Automazione di un processo completo

Supponiamo di dover automatizzare un processo che include l'estrazione di dati da una pagina web, la manipolazione dei dati e l'invio dei risultati via email.

Step 1: Estrazione dei dati da una pagina web

Utilizzeremo BeautifulSoup per estrarre i dati da una pagina web.

```
import requests

from bs4 import BeautifulSoup


# Invio di una richiesta HTTP alla pagina web

url = 'https://example.com/dati'

response = requests.get(url)


# Parsing del contenuto HTML

soup = BeautifulSoup(response.content, 'lxml')


# Estrazione dei dati

dati = []

for row in soup.find_all('tr'):

 cells = row.find_all('td')

 if len(cells) > 1:

 dati.append([cell.text for cell in cells])


print(dati)
```

Step 2: Manipolazione dei dati

Utilizzeremo pandas per manipolare i dati estratti.

```
import pandas as pd


# Creazione di un DataFrame dai dati estratti

df = pd.DataFrame(dati, columns=['Colonna1', 'Colonna2',
'Colonna3'])


# Esempio di manipolazione dei dati

df['Colonna3'] = pd.to_numeric(df['Colonna3'],
errors='coerce')

df_aggregato = df.groupby('Colonna1').agg({'Colonna3':
'sum'})

print(df_aggregato)
```

Step 3: Invio dei risultati via email

Utilizzeremo smtplib per inviare i risultati via email.

```
import smtplib

from email.mime.text import MIMEText

from email.mime.multipart import MIMEMultipart


# Configurazione dell'email
```

```
sender_email = "tuo_email@example.com"

receiver_email = "destinatario@example.com"

password = "tuo_password"


# Creazione del messaggio

message = MIMEMultipart()

message["From"] = sender_email

message["To"] = receiver_email

message["Subject"] = "Risultati dell'automazione"


# Corpo dell'email

body = df_aggregato.to_html()

message.attach(MIMEText(body, "html"))


# Connessione al server SMTP e invio dell'email

with smtplib.SMTP_SSL("smtp.example.com", 465) as server:

 server.login(sender_email, password)

 server.sendmail(sender_email, receiver_email,
message.as_string())


print("Email inviata con successo")
```

L'automazione del parsing di file e delle attività web con Python può migliorare significativamente l'efficienza e ridurre gli errori manuali. Python offre potenti librerie come BeautifulSoup, Selenium e requests per facilitare queste attività. Con la giusta combinazione di strumenti, è possibile automatizzare processi complessi che includono l'estrazione e la manipolazione dei dati, l'interazione con API e siti web, e l'invio di notifiche e report. Questa automazione consente di risparmiare tempo e risorse, permettendo di concentrarsi su attività più strategiche e a valore aggiunto.

12.3 SCRIPTING DI SISTEMA

Lo scripting di sistema è una delle aree più potenti e versatili in cui Python può essere utilizzato. Gli script di sistema permettono di automatizzare attività amministrative, configurare ambienti, gestire processi e risorse, monitorare sistemi e molto altro. In questo capitolo, esploreremo come utilizzare Python per lo scripting di sistema, coprendo vari argomenti come l'esecuzione di comandi di sistema, la gestione dei processi, la manipolazione dei file di sistema, e il monitoraggio delle risorse.

Esecuzione di comandi di sistema

L'esecuzione di comandi di sistema è una delle funzionalità fondamentali nello scripting di sistema. Python fornisce il modulo `subprocess` per eseguire comandi di sistema e interagire con i processi.

Utilizzo del modulo subprocess

Il modulo `subprocess` permette di eseguire comandi di sistema e di ottenere i loro output.

```
import subprocess
```

```
# Esecuzione di un comando semplice

result = subprocess.run(['ls', '-l'], capture_output=True,
text=True)

print(result.stdout)



# Esecuzione di un comando con gestione degli errori

try:

 result = subprocess.run(['mkdir', 'nuova_directory'],
check=True, capture_output=True, text=True)

 print(result.stdout)

except subprocess.CalledProcessError as e:

 print(f"Errore durante l'esecuzione del comando: {e}")
```

Gestione dei processi

Python permette di avviare, terminare e monitorare processi di sistema, il che è utile per gestire applicazioni e servizi.

Avvio e terminazione dei processi

```
import subprocess

import os
```

```
# Avvio di un processo

process = subprocess.Popen(['ping', 'google.com'],
stdout=subprocess.PIPE, stderr=subprocess.PIPE)



# Monitoraggio del processo

try:

 stdout, stderr = process.communicate(timeout=10)

 print(stdout.decode())

except subprocess.TimeoutExpired:

 process.kill()

 stdout, stderr = process.communicate()

 print("Processo terminato dopo il timeout")
```

Manipolazione dei file di sistema

La manipolazione dei file di sistema include operazioni come la lettura, la scrittura, la copia, lo spostamento e l'eliminazione di file e directory.

Utilizzo del modulo os

Il modulo `os` fornisce funzioni per interagire con il sistema operativo e manipolare file e directory.

```
import os
```

```
# Creazione di una nuova directory
os.mkdir('nuova_directory')
print("Directory creata")

# Rimozione di una directory
os.rmdir('nuova_directory')
print("Directory rimossa")

# Creazione di un file
with open('file.txt', 'w') as file:
 file.write('Contenuto del file')
print("File creato e scritto")

# Lettura di un file
with open('file.txt', 'r') as file:
 contenuto = file.read()
 print(contenuto)

# Rimozione di un file
os.remove('file.txt')
print("File rimosso")
```

Utilizzo del modulo shutil

Il modulo `shutil` fornisce funzioni per copiare, spostare e rinominare file e directory.

```
import shutil


# Copia di un file

shutil.copy('file_originale.txt', 'file_copia.txt')

print("File copiato")


# Spostamento di un file

shutil.move('file_copia.txt',
'nuova_directory/file_copia.txt')

print("File spostato")


# Rinomina di un file

os.rename('file_originale.txt', 'file_rinominato.txt')

print("File rinominato")
```

Monitoraggio delle risorse di sistema

Il monitoraggio delle risorse di sistema, come l'uso della CPU, la memoria e lo spazio su disco, è cruciale per la gestione e l'ottimizzazione delle prestazioni dei sistemi.

Utilizzo del modulo psutil

Il modulo `psutil` è una libreria potente per il monitoraggio delle risorse di sistema e la gestione dei processi.

Installazione di psutil

Per installare `psutil`, si utilizza `pip`.

```
pip install psutil
```

Monitoraggio dell'uso della CPU e della memoria

```
import psutil


# Uso della CPU

cpu_percent = psutil.cpu_percent(interval=1)

print(f"Uso della CPU: {cpu_percent}%")


# Uso della memoria

memory_info = psutil.virtual_memory()

print(f"Uso della memoria: {memory_info.percent}%")
```

```
# Uso dello spazio su disco

disk_usage = psutil.disk_usage('/')

print(f"Uso dello spazio su disco: {disk_usage.percent}%")
```

Monitoraggio dei processi

```
import psutil


# Elenco di tutti i processi attivi

for proc in psutil.process_iter(['pid', 'name', 'username']):

 print(proc.info)


# Dettagli di un processo specifico

pid = 1234 # Sostituire con un PID valido

try:

 process = psutil.Process(pid)

 print(f"Nome del processo: {process.name()}")

 print(f"Stato del processo: {process.status()}")

 print(f"Uso della memoria: {process.memory_info().rss /
(1024 * 1024)} MB")

except psutil.NoSuchProcess:
```

```
 print(f"Nessun processo trovato con PID {pid}")
```

Automazione delle attività di sistema

L'automazione delle attività di sistema include operazioni come la gestione degli utenti, la configurazione dei servizi, e l'automazione dei backup.

Gestione degli utenti

La gestione degli utenti può includere la creazione di nuovi utenti, la modifica delle autorizzazioni e l'eliminazione degli utenti esistenti.

Utilizzo del modulo subprocess per la gestione degli utenti

```
import subprocess


# Creazione di un nuovo utente (solo su Unix/Linux)

try:

 subprocess.run(['sudo', 'useradd', 'nuovo_utente'],
check=True)

 print("Nuovo utente creato")

except subprocess.CalledProcessError as e:

 print(f"Errore durante la creazione dell'utente: {e}")


# Modifica delle autorizzazioni dell'utente
```

```
try:

 subprocess.run(['sudo', 'usermod', '-aG', 'sudo',
'nuovo_utente'], check=True)

 print("Autorizzazioni modificate")

except subprocess.CalledProcessError as e:

 print(f"Errore durante la modifica delle autorizzazioni:
{e}")



# Eliminazione di un utente

try:

 subprocess.run(['sudo', 'userdel', 'nuovo_utente'],
check=True)

 print("Utente eliminato")

except subprocess.CalledProcessError as e:

 print(f"Errore durante l'eliminazione dell'utente: {e}")
```

Automazione dei backup

Automatizzare i backup è essenziale per garantire che i dati importanti siano protetti e facilmente recuperabili in caso di perdita di dati.

Script di backup

Un semplice script di backup può essere creato utilizzando moduli come `os` e `shutil`.

```
import os

import shutil

import datetime


# Directory da salvare

source_dir = '/path/to/source_directory'

# Directory di destinazione del backup

backup_dir = '/path/to/backup_directory'


# Creazione di un nome unico per il backup

backup_name =
f"backup_{datetime.datetime.now().strftime('%Y%m%d%H%M%S')}"

backup_path = os.path.join(backup_dir, backup_name)


# Creazione del backup

shutil.copytree(source_dir, backup_path)

print(f"Backup completato: {backup_path}")
```

Pianificazione degli script

Per eseguire script di sistema a intervalli regolari, è possibile utilizzare strumenti di pianificazione come `cron` su Unix/Linux o Task Scheduler su Windows.

Esempio di cron job su Unix/Linux

Per aggiungere un cron job su Unix/Linux, si utilizza il comando `crontab -e` per modificare il file crontab e aggiungere una nuova voce.

```
# Esempio di cron job che esegue uno script Python ogni giorno alle
8:00 AM

0 8 * * * /usr/bin/python3 /path/to/script.py
```

Esempio di Task Scheduler su Windows

Per creare un'attività pianificata su Windows, si utilizza l'Utilità di pianificazione (Task Scheduler) e si segue la procedura guidata per creare una nuova attività, specificando il percorso dello script Python e l'intervallo di esecuzione.

Lo scripting di sistema con Python è una potente tecnica che permette di automatizzare e gestire una vasta gamma di attività amministrative, migliorando l'efficienza e riducendo il rischio di errori manuali. Grazie a moduli come `subprocess`, `os`, `shutil` e `psutil`, Python offre strumenti flessibili e potenti per l'interazione con il sistema operativo, la gestione dei processi, la manipolazione dei file di sistema e il monitoraggio delle risorse. Automatizzare le attività di sistema può portare a significativi miglioramenti in termini di produttività e affidabilità, permettendo agli amministratori di sistema di concentrarsi su attività più strategiche e a valore aggiunto.

CAPITOLO 13: TESTING E DEBUGGING

13.1 Introduzione al testing

Il testing è una parte essenziale del ciclo di sviluppo del software, fondamentale per assicurare che il codice funzioni correttamente e rispetti i requisiti specificati. Attraverso il testing, è possibile identificare e correggere bug, migliorare la qualità del software e assicurarsi che le modifiche non introducano nuovi problemi. Questo capitolo offre un'introduzione al testing, esplorando i diversi tipi di test, i benefici del testing e le best practice per implementare una strategia di testing efficace.

Importanza del Testing

Il testing è cruciale per diversi motivi:

- **Assicurazione della qualità**: Garantisce che il software soddisfi i requisiti e funzioni come previsto, aumentando la fiducia degli utenti nell'applicazione.
- **Prevenzione dei bug**: Identifica e corregge i bug prima che il software venga rilasciato in produzione, riducendo il rischio di errori critici che possono causare perdite economiche o danni alla reputazione.
- **Manutenibilità**: Facilita la modifica del codice, assicurando che le nuove modifiche non

compromettano le funzionalità esistenti e che il software rimanga facilmente aggiornabile.
- **Documentazione**: I test servono come documentazione vivente del comportamento del software, aiutando i nuovi sviluppatori a comprendere il codice e le sue funzionalità.

Tipi di Testing

Esistono diversi tipi di test che possono essere eseguiti per garantire la qualità del software. Questi test possono essere categorizzati in base alla loro granularità e allo scopo specifico:

Test Unitari

I test unitari verificano il funzionamento di singole unità di codice, come funzioni o metodi. Sono i test più granulari e mirano a garantire che ogni componente funzioni correttamente in isolamento.

Esempio: Verificare che una funzione che somma due numeri restituisca il risultato corretto.

```
def somma(a, b):
 return a + b


def test_somma():
 assert somma(2, 3) == 5
 assert somma(-1, 1) == 0
```

Test di Integrazione

I test di integrazione verificano l'interazione tra più componenti o moduli del software per assicurarsi che funzionino correttamente insieme.

Esempio: Verificare che un modulo di autenticazione funzioni correttamente con il database degli utenti.

```
def test_autenticazione():
   utente = autenticazione('username', 'password')
   assert utente is not None
```

Test Funzionali

I test funzionali verificano che il software rispetti i requisiti funzionali specificati. Si concentrano sul comportamento del sistema nel suo complesso piuttosto che sui singoli componenti.

Esempio: Verificare che un utente possa registrarsi, effettuare il login e accedere a una pagina protetta.

```
def test_registrazione_login():
    assert registra_utente('username', 'password')
    assert effettua_login('username', 'password')
```

```
    assert accedi_pagina_protetta('username') == 'Accesso
consentito'
```

Test di Accettazione

I test di accettazione verificano che il software soddisfi i requisiti del cliente o dell'utente finale. Questi test sono spesso eseguiti alla fine del ciclo di sviluppo per garantire che il prodotto sia pronto per il rilascio.

Esempio: Verificare che tutte le funzionalità di un'applicazione e-commerce funzionino come previsto per l'utente finale.

Test di Carico e Prestazioni

I test di carico e prestazioni valutano come il software si comporta sotto carico, testando la velocità, la scalabilità e la stabilità. Questi test sono cruciali per applicazioni che devono gestire un grande volume di utenti o dati.

Esempio: Verificare che un sito web possa gestire 10.000 utenti contemporaneamente senza degradare le prestazioni.

Test di Sicurezza

I test di sicurezza verificano che il software sia protetto contro le vulnerabilità e gli attacchi. Questi test includono l'analisi del codice, il test delle vulnerabilità e la verifica dei controlli di accesso.

Esempio: Verificare che l'applicazione non sia vulnerabile a SQL injection o XSS (Cross-Site Scripting).

Benefici del Testing

Implementare una strategia di testing robusta offre numerosi vantaggi:

- **Rilevamento precoce dei bug**: I test aiutano a identificare e correggere i bug nelle fasi iniziali del ciclo di sviluppo, riducendo i costi di correzione e minimizzando i rischi di malfunzionamenti.
- **Affidabilità**: Un software ben testato è più affidabile e meno soggetto a guasti imprevisti, migliorando l'esperienza utente e aumentando la fiducia nel prodotto.
- **Manutenzione facilitata**: I test rendono il codice più manutenibile, permettendo di apportare modifiche e aggiungere nuove funzionalità con maggiore sicurezza e minor rischio di errori.
- **Miglioramento continuo**: I test consentono di verificare rapidamente l'impatto delle modifiche e garantire che le nuove funzionalità non rompano il comportamento esistente, facilitando l'evoluzione continua del software.
- **Documentazione del codice**: I test servono come documentazione vivente del comportamento del software, facilitando la comprensione del codice da parte di nuovi sviluppatori e migliorando la comunicazione tra i membri del team.

Best Practice per il Testing

Per implementare una strategia di testing efficace, è importante seguire alcune best practice:

- **Scrivere test indipendenti**: Ogni test dovrebbe essere indipendente dagli altri, in modo che possa essere eseguito isolatamente senza influenzare gli altri test.

Questo assicura che i test siano affidabili e possano essere eseguiti in qualsiasi ordine.

- **Automatizzare i test**: Automatizzare il più possibile i test per ridurre il carico di lavoro manuale e garantire una copertura di test costante e ripetibile. L'automazione dei test consente di eseguirli frequentemente e di integrare il testing nel processo di sviluppo continuo.
- **Eseguire i test frequentemente**: Eseguire i test regolarmente, preferibilmente ad ogni modifica del codice, per identificare rapidamente eventuali regressioni e garantire che il software rimanga stabile e affidabile.
- **Mantenere i test aggiornati**: Aggiornare i test ogni volta che il codice cambia, assicurandosi che riflettano sempre lo stato attuale del software. I test obsoleti possono causare falsi positivi o negativi, compromettendo l'efficacia del processo di testing.
- **Scrivere test chiari e leggibili**: I test dovrebbero essere facili da leggere e comprendere, con nomi significativi e una struttura chiara. Test ben scritti facilitano la manutenzione e la comprensione del codice.
- **Utilizzare il principio del "test first"**: Implementare i test prima di scrivere il codice, seguendo metodologie come TDD (Test-Driven Development) per garantire che il codice soddisfi i requisiti fin dall'inizio. Questo approccio incentiva la scrittura di codice pulito e ben progettato.

Il testing è un componente essenziale del ciclo di sviluppo del software che contribuisce a garantire la qualità, la stabilità e la manutenibilità del codice. Implementando strategie di testing efficaci e seguendo le best practice, gli sviluppatori possono ridurre il rischio di bug e migliorare l'esperienza utente complessiva.

Mantenere un approccio proattivo al testing è fondamentale per il successo di qualsiasi progetto software.

13.2 UNITTEST E PYTEST

Nel contesto dello sviluppo software in Python, `unittest` e `pytest` sono due dei framework di testing più diffusi e utilizzati per garantire la qualità del codice. Entrambi i framework offrono strumenti per scrivere, organizzare ed eseguire test, aiutando gli sviluppatori a identificare e risolvere problemi in modo efficiente. Questo capitolo esplora le caratteristiche principali di `unittest` e `pytest`, fornendo esempi pratici per mostrare come utilizzarli nel processo di sviluppo.

Unittest

`unittest` è il framework di testing standard di Python, incluso nella libreria standard. È basato sullo stile dei test di unità di JUnit, un framework di testing per Java, e fornisce una struttura di base per scrivere e organizzare test, nonché per eseguire test e verificare i risultati.

Caratteristiche Principali

- **Integrazione con Python**: Essendo parte della libreria standard, `unittest` non richiede installazioni aggiuntive ed è sempre disponibile in qualsiasi ambiente Python.

- **Struttura dei Test**: Utilizza classi e metodi per organizzare i test, consentendo una facile gestione di test complessi e la possibilità di eseguire setup e teardown prima e dopo ogni test.
- **Asserzioni**: Offre una serie di metodi di asserzione per verificare le condizioni dei test, come `assertEqual`, `assertTrue`, `assertRaises`, ecc.
- **Compatibilità**: Supporta il concetto di fixture, che permette di preparare l'ambiente di test prima di eseguire il codice di test vero e proprio.

Esempio di Utilizzo:

```
import unittest


# Funzione da testare

def somma(a, b):

 return a + b


# Classe di test che eredita da unittest.TestCase

class TestSomma(unittest.TestCase):


 # Metodo di setup eseguito prima di ogni test

 def setUp(self):

 self.a = 2

 self.b = 3
```

```
# Metodo di teardown eseguito dopo ogni test
 def tearDown(self):
 pass


# Test che verifica se la somma è corretta
 def test_somma_positiva(self):
 risultato = somma(self.a, self.b)
 self.assertEqual(risultato, 5)


# Test per verificare il comportamento con numeri negativi
 def test_somma_negativa(self):
 risultato = somma(-1, 1)
 self.assertEqual(risultato, 0)


# Avvia i test
if __name__ == '__main__':
 unittest.main()
```

Pytest

`pytest` è un framework di testing avanzato per Python che offre funzionalità potenti e una sintassi semplice e

intuitiva. È particolarmente apprezzato per la sua capacità di gestire test complessi e per la flessibilità che offre agli sviluppatori.

Caratteristiche Principali

- **Sintassi Semplice**: Non richiede l'uso di classi e metodi per organizzare i test, rendendo la scrittura dei test più diretta e leggibile.
- **Fixture Avanzate**: Supporta un sistema di fixture avanzato che permette di configurare test in modo modulare e riutilizzabile, facilitando la configurazione e la pulizia dell'ambiente di test.
- **Estendibilità**: Offre un sistema di plugin ricco che consente di estendere le funzionalità di base con componenti aggiuntivi per la gestione di test paralleli, il monitoraggio della copertura del codice, e molto altro.
- **Supporto per Parametri**: Consente di parametrizzare i test, permettendo l'esecuzione dello stesso test con diversi set di dati.

Esempio di Utilizzo:

```
# Funzione da testare

def somma(a, b):

 return a + b



# Test semplice che verifica la somma

def test_somma():
```

```
 assert somma(2, 3) == 5
 assert somma(-1, 1) == 0

# Esempio di utilizzo delle fixture
import pytest

@pytest.fixture
def numeri():
 return 2, 3


# Utilizzo della fixture nel test
def test_somma_con_fixture(numeri):
 a, b = numeri
 assert somma(a, b) == 5


# Esecuzione del test tramite pytest
if __name__ == '__main__':
 import pytest
 pytest.main()
```

Esecuzione di Pytest

Per eseguire i test scritti con `pytest`, basta utilizzare il comando `pytest` dalla riga di comando nella directory che

contiene i file di test. Pytest rileverà automaticamente i file che iniziano con `test_` e le funzioni che iniziano con `test`.

```
pytest nome_del_file_test.py
```

Confronto tra Unittest e Pytest

Entrambi i framework offrono strumenti potenti per il testing, ma presentano differenze chiave che possono rendere uno più adatto rispetto all'altro a seconda del progetto.

- **Facilità d'uso**: `pytest` offre una sintassi più semplice e diretta rispetto a `unittest`, rendendo più facile la scrittura e la lettura dei test.
- **Configurazione delle Fixture**: `pytest` fornisce un sistema di fixture più avanzato e flessibile rispetto a quello di `unittest`, che consente una configurazione modulare e riutilizzabile dei test.
- **Compatibilità**: `unittest` è incluso nella libreria standard di Python, mentre `pytest` richiede l'installazione di un pacchetto esterno.
- **Estendibilità**: `pytest` offre un sistema di plugin che permette di estendere le funzionalità di base con componenti aggiuntivi, mentre `unittest` è più limitato in questo senso.

Sia `unittest` che `pytest` sono strumenti potenti per il testing in Python, ciascuno con i propri punti di forza e debolezze. La scelta tra i due dipende spesso dalle esigenze specifiche del progetto e dalle preferenze

personali dello sviluppatore. Mentre `unittest` è una scelta solida per progetti più semplici o per coloro che preferiscono rimanere all'interno della libreria standard di Python, `pytest` è ideale per progetti più complessi o per coloro che cercano una maggiore flessibilità e facilità d'uso. In ogni caso, l'importante è implementare una strategia di testing robusta per garantire la qualità del software e facilitare la manutenzione e l'evoluzione del codice.

13.3 TECNICHE DI DEBUGGING

Il debugging è un processo essenziale nello sviluppo del software, che consiste nell'identificare, analizzare e risolvere i problemi nel codice. Mentre i test aiutano a rilevare la presenza di bug, il debugging è il passo successivo che mira a scoprire la causa dei problemi e correggerli. In questo capitolo, esploreremo le tecniche di debugging più efficaci, gli strumenti disponibili in Python e le best practice per risolvere i problemi in modo efficiente.

Approcci al Debugging

Il debugging può essere un processo complesso e richiede un approccio sistematico per essere efficace. Ecco alcuni approcci comuni al debugging:

1. Reproduzione del Problema

Prima di tutto, è importante riuscire a riprodurre il problema in modo consistente. Un caso di test ripetibile aiuta a isolare il problema e a garantire che la soluzione sia efficace. Per fare ciò, è utile raccogliere tutte le informazioni disponibili sull'errore, come i passi per riprodurlo, i dati di input, e l'ambiente in cui si verifica.

2. Analisi del Codice

Analizzare il codice è uno dei primi passi per identificare gli errori logici o sintattici che potrebbero causare il problema. Una revisione del codice può spesso rivelare errori evidenti o indicare parti del codice che necessitano di maggiore attenzione.

3. Utilizzo di Logger

L'inserimento di dichiarazioni di log nel codice aiuta a tracciare il flusso dell'applicazione e identificare dove il comportamento diverge dalle aspettative. I logger possono fornire informazioni dettagliate sullo stato del programma in vari punti, facilitando l'individuazione della causa del problema.

```
import logging


logging.basicConfig(level=logging.DEBUG)



def somma(a, b):

 logging.debug(f"somma() chiamato con {a} e {b}")

 return a + b


somma(2, 3)
```

4. Debugger Interattivo

Un debugger consente di eseguire il codice passo passo, ispezionare variabili e lo stato dell'applicazione in punti critici. Utilizzare un debugger permette di esplorare il

contesto del problema senza dover modificare il codice con istruzioni di stampa o log temporanee.

5. Test di Ipotesi

Formulare ipotesi su cosa potrebbe causare il problema e utilizzare esperimenti per testare queste ipotesi può essere un approccio utile. Prova a modificare una cosa alla volta e osserva come queste modifiche influenzano il comportamento del programma.

6. Controllo delle Dipendenze

Assicurati che tutte le dipendenze del tuo progetto siano aggiornate e compatibili tra loro. A volte, i problemi possono derivare da conflitti tra versioni di librerie o da modifiche introdotte in aggiornamenti recenti.

Strumenti di Debugging in Python

Python offre diversi strumenti e librerie che facilitano il processo di debugging:

PDB (Python Debugger)

`pdb` è il debugger integrato di Python, che consente di eseguire il codice passo passo, impostare breakpoint e ispezionare lo stato dell'applicazione. È particolarmente utile per esplorare il flusso di esecuzione e le variabili in tempo reale.

Esempio di utilizzo di `pdb`:

```
import pdb
```

```
def somma(a, b):

 pdb.set_trace()

 return a + b


somma(2, 3)
```

Quando si esegue questo codice, il debugger si interromperà all'inizio della funzione `somma`, consentendo di esaminare le variabili `a` e `b` prima di continuare.

IPython

IPython è un interprete interattivo di Python che offre strumenti di debug avanzati. Utilizzando il magic command `%debug`, è possibile entrare in modalità debug subito dopo un'eccezione, consentendo di esaminare lo stato del programma nel momento in cui si è verificato l'errore.

Loguru

Loguru è una libreria di logging che offre un'interfaccia semplice e potente per il logging, facilitando il tracciamento dei problemi. Fornisce strumenti per il logging asincrono, la gestione dei file di log e la formattazione avanzata dei messaggi.

```
from loguru import logger
```

```
logger.add("file_{time}.log", rotation="1 day")


def somma(a, b):
 logger.debug("Calcolando la somma di {} e {}", a, b)
 return a + b
```

Sentry

Sentry è un servizio di monitoraggio degli errori che raccoglie informazioni sugli errori in tempo reale e offre strumenti per analizzarli e risolverli rapidamente. Integra funzionalità di tracking delle versioni e gestione degli incidenti per aiutare a mantenere la stabilità del software.

Best Practice per il Debugging

Ecco alcune best practice per un debugging efficace:

- **Inizia con i Log**: Utilizza il logging per ottenere informazioni dettagliate sul comportamento dell'applicazione, in modo da poter individuare rapidamente i problemi senza interrompere il flusso di esecuzione.
- **Mantieni la Calma**: Il debugging può essere frustrante, ma è importante mantenere la calma e affrontare il problema in modo sistematico.
- **Divide et Impera**: Dividi il problema in parti più piccole e affrontale una alla volta. Concentrati sul capire cosa funziona correttamente e cosa no.

- **Documenta i Problemi e le Soluzioni**: Tieni traccia dei problemi riscontrati e delle soluzioni adottate, in modo da poterli consultare in futuro e condividere le conoscenze con il tuo team.
- **Usa gli Strumenti Giusti**: Familiarizza con gli strumenti di debugging e scegli quelli più adatti alle tue esigenze e al tuo flusso di lavoro.
- **Verifica l'Ambiente**: Assicurati che l'ambiente di sviluppo e produzione siano configurati correttamente e in modo coerente, in modo da ridurre le discrepanze che possono causare problemi difficili da riprodurre.

Il debugging è un'abilità fondamentale per ogni sviluppatore, che richiede pazienza, attenzione ai dettagli e un approccio metodico. Utilizzando le tecniche e gli strumenti di debugging discussi in questo capitolo, gli sviluppatori possono risolvere i problemi in modo più efficiente e migliorare la qualità del software. Un debugging efficace non solo aiuta a risolvere i problemi attuali, ma contribuisce anche a prevenire futuri errori e a garantire che il software rimanga stabile e affidabile nel tempo.

CAPITOLO 14: PYTHON AVANZATO

14.1 Decoratori

I decoratori in Python sono una funzionalità potente e flessibile che consente di modificare il comportamento di funzioni o metodi. Grazie ai decoratori, è possibile aggiungere funzionalità a una funzione esistente senza modificarne il codice originale. Questo è particolarmente utile per aspetti trasversali come logging, autenticazione e gestione delle eccezioni.

Introduzione ai Decoratori

Un decoratore è essenzialmente una funzione che prende un'altra funzione come argomento e restituisce una nuova funzione che estende o modifica il comportamento della funzione originale. I decoratori sono spesso utilizzati come un metodo per rispettare il principio DRY (Don't Repeat Yourself) nel codice.

Ecco un esempio di un semplice decoratore:

```
def decoratore_semplice(funzione):

 def wrapper():
```

```
 print("Prima dell'esecuzione della funzione")
 funzione()
 print("Dopo l'esecuzione della funzione")
 return wrapper

@decoratore_semplice
def mia_funzione():
 print("Esecuzione della funzione")

mia_funzione()
```

L'output sarà:

```
Prima dell
'esecuzione della funzione

Esecuzione della funzione

Dopo l
'esecuzione della funzione
```

Decoratori con Argomenti

I decoratori possono anche accettare argomenti, consentendo una maggiore flessibilità. Per fare ciò, è necessario annidare un ulteriore livello di funzioni.

```
def decoratore_con_argomenti(argomento):
   def decoratore_reale(funzione):
       def wrapper(*args, **kwargs):
          print(f"Argomento del decoratore: {argomento}")
         return funzione(*args, **kwargs)
       return wrapper
    return decoratore_reale

@decoratore_con_argomenti("Hello")
def mia_funzione_con_argomenti():
   print("Esecuzione della funzione")

mia_funzione_con_argomenti()
```

L'output sarà:

```
Argomento del decoratore: Hello
Esecuzione della funzione
```

Decoratori Multipli

È possibile applicare più decoratori a una singola funzione. In questo caso, i decoratori vengono applicati dall'interno verso l'esterno.

```
def decoratore_uno(funzione):
    def wrapper(*args, **kwargs):
        print("Decoratore Uno")
        return funzione(*args, **kwargs)
     return wrapper

def decoratore_due(funzione):
    def wrapper(*args, **kwargs):
        print("Decoratore Due")
        return funzione(*args, **kwargs)
    return wrapper

@decoratore_uno
@decoratore_due
def funzione_decorata():
     print("Esecuzione della funzione decorata")

funzione_decorata()
```

L'output sarà:

```
Decoratore Uno
Decoratore Due
```

```
Esecuzione della funzione decorata
```

Decoratori di Classe

I decoratori possono essere utilizzati anche con le classi. Un decoratore di classe è una funzione che prende una classe come argomento e restituisce una nuova classe con funzionalità estese o modificate.

```
def decoratore_classe(cls):
    class NuovaClasse(cls):
        def nuovo_metodo(self):
           print("Nuovo metodo aggiunto dalla decoratore")
   return NuovaClasse

@decoratore_classe
class MiaClasse:
     def metodo_esistente(self):
         print("Metodo esistente")

oggetto = MiaClasse()
oggetto.metodo_esistente()
oggetto.nuovo_metodo()
```

L'output sarà:

```
Metodo esistente

Nuovo metodo aggiunto dalla decoratore
```

Funzioni Decoratori Incorporate

Python include alcune funzioni decoratori incorporate che sono molto utili. Alcuni esempi comuni includono `@staticmethod`, `@classmethod` e `@property`.

- `@staticmethod`: Trasforma un metodo in un metodo statico, che non richiede un'istanza della classe per essere chiamato.
- `@classmethod`: Trasforma un metodo in un metodo di classe, che può essere chiamato sulla classe stessa piuttosto che su un'istanza.
- `@property`: Permette di definire metodi come proprietà, che possono essere accedute come attributi.

Ecco un esempio di ciascuno:

```
class Esempio:

   def __init__(self, valore):

      self._valore = valore


@staticmethod

 def metodo_statico():
```

```
    print("Questo è un metodo statico")

@classmethod
 def metodo_classe(cls):
    print(f"Questo è un metodo di classe chiamato su {cls}")

@property
 def valore(self):
    return self._valore

@valore.setter
 def valore(self, nuovo_valore):
    self._valore = nuovo_valore

Esempio.metodo_statico()
Esempio.metodo_classe()
oggetto = Esempio(10)
print(oggetto.valore)
oggetto.valore = 20
print(oggetto.valore)
```

L'output sarà:

```
Questo è un metodo statico

Questo è un metodo di classe chiamato su <class
'__main__.Esempio'>

10

20
```

I decoratori sono strumenti estremamente potenti nel toolkit di un programmatore Python. Consentono di estendere e modificare il comportamento delle funzioni e delle classi in modo elegante e modulare. Con una buona comprensione dei decoratori, è possibile scrivere codice più pulito, riutilizzabile e manutenibile.

14.2 GENERATORI

I generatori sono una caratteristica potente di Python che consente di creare iteratori in modo semplice e conciso. I generatori forniscono un modo per produrre una sequenza di valori nel tempo, invece di calcolare e restituire tutti i valori contemporaneamente. Questo è particolarmente utile per gestire sequenze di grandi dimensioni o infinite, poiché i generatori utilizzano la memoria in modo molto più efficiente.

14.2.1 Introduzione ai Generatori

Un generatore è una funzione che utilizza la parola chiave `yield` per restituire un valore. Ogni volta che la funzione generator viene chiamata, l'esecuzione riprende da dove è stata interrotta l'ultima `yield`. Questo permette di creare iteratori in modo molto più semplice rispetto alla definizione di una classe con i metodi `__iter__()` e `__next__()`.

Ecco un esempio di un semplice generatore:

```
def mio_generatore():

    yield 1
```

```
    yield 2

    yield 3


gen = mio_generatore()

for valore in gen:

     print(valore)
```

L'output sarà:

```
1

2

3
```

Funzionamento dei Generatori

Quando una funzione generatore viene chiamata, non viene eseguita immediatamente. Invece, restituisce un oggetto generatore che supporta il protocollo iterator. Ogni chiamata al metodo `__next__()` dell'oggetto generatore esegue la funzione fino alla successiva `yield`.

Ecco un esempio per illustrare questo comportamento:

```
def contatore(max):
```

```
    n = 0
    while n < max:
      yield n
      n += 1

gen = contatore(3)
print(next(gen)) # Output: 0
print(next(gen)) # Output: 1
print(next(gen)) # Output: 2
# print(next(gen)) # Solleverà StopIteration
```

Generatori e Memoria

Uno dei vantaggi principali dei generatori è che sono molto efficienti in termini di memoria. Poiché i generatori producono un valore alla volta, non è necessario memorizzare l'intera sequenza in memoria.

Ecco un esempio che confronta l'uso della memoria tra una lista e un generatore:

```
import sys

# Lista
lista = [i for i in range(1000000)]
```

```
print(f"Dimensione della lista: {sys.getsizeof(lista)}
bytes")



# Generatore

generatore = (i for i in range(1000000))

print(f"Dimensione del generatore:
{sys.getsizeof(generatore)} bytes")
```

L'output mostrerà che la lista occupa significativamente più memoria rispetto al generatore.

14.2.4 Generatori Infiniti

I generatori sono particolarmente utili per rappresentare sequenze infinite, dove non è possibile o pratico generare tutti i valori in anticipo.

Ecco un esempio di un generatore infinito che produce numeri pari:

```
def numeri_pari():

     n = 0

     while True:

          yield n

          n += 2


gen = numeri_pari()
```

```
for _ in range(5):
    print(next(gen))
```

L'output sarà:

```
0
2
4
6
8
```

Generatori con Funzioni Incorporate

Python include alcune funzioni incorporate che facilitano il lavoro con i generatori. Alcune di queste funzioni sono `enumerate()`, `zip()`, `map()`, e `filter()`.

Ecco un esempio di utilizzo di `map()` con un generatore:

```
def quadrati(n):
    for i in range(n):
      yield i * i

gen = quadrati(5)
quad_gen = map(lambda x: x + 1, gen)
for val in quad_gen:
    print(val)
```

L'output sarà:

```
1

2

5

10

17
```

Decoratori per Generatori

Così come le funzioni, anche i generatori possono essere decorati. Un decoratore per un generatore può essere utilizzato per estendere o modificare il comportamento del generatore originale.

Ecco un esempio di un decoratore per un generatore:

```
def decoratore_generatore(funzione):

    def wrapper(*args, **kwargs):

    gen = funzione(*args, **kwargs)

    for valore in gen:

        yield valore + 1
```

```
  return wrapper


@decoratore_generatore
def mio_generatore():
    yield 1
    yield 2
    yield 3


for valore in mio_generatore():
    print(valore)
```

L'output sarà:

```
2
3
4
```

I generatori sono strumenti potenti e flessibili che consentono di gestire in modo efficiente sequenze di dati, specialmente quando la memoria è una risorsa limitata. Grazie alla loro semplicità e alla capacità di produrre valori su richiesta, i generatori sono una parte essenziale del toolkit di ogni programmatore Python avanzato.

14.3 CONTEXT MANAGER

I context manager in Python sono strumenti che permettono di gestire risorse in modo sicuro ed efficiente. Utilizzando i context manager, è possibile assicurarsi che le risorse vengano acquisite e rilasciate correttamente, anche in presenza di eccezioni. Questo rende il codice più pulito e più facile da mantenere.

Introduzione ai Context Manager

I context manager vengono utilizzati principalmente per gestire risorse come file, connessioni di rete o lock. La sintassi `with` di Python fornisce un modo conveniente per utilizzare i context manager.

Ecco un esempio di base di utilizzo di un context manager per gestire un file:

```
with open('file.txt', 'r') as file:

     contenuto = file.read()

     print(contenuto)
```

In questo esempio, il file viene aperto e letto all'interno del blocco `with`. Quando l'esecuzione esce dal blocco `with`, il file viene chiuso automaticamente, anche se si verifica un'eccezione durante la lettura del file.

Creazione di un Context Manager con `__enter__` e `__exit__`

È possibile creare un context manager personalizzato definendo una classe che implementa i metodi `__enter__` e `__exit__`. Il metodo `__enter__` viene chiamato all'inizio del blocco `with`, mentre `__exit__` viene chiamato alla fine del blocco.

Ecco un esempio di un context manager personalizzato:

```
class MioContextManager:

 def __enter__(self):

    print("Inizio del blocco context manager")

    return self


def __exit__(self, exc_type, exc_value, traceback):

    print("Fine del blocco context manager")

    if exc_type is not None:

        print(f"Gestione dell'eccezione: {exc_value}")

 return True # Sopprime l'eccezione


with MioContextManager() as manager:
```

```
    print("All'interno del blocco with")

    # sollevare un'eccezione per testare la gestione delle
eccezioni

    raise ValueError("Qualcosa è andato storto")
```

L'output sarà:

```
Inizio del blocco context manager

All'interno del blocco with

Fine del blocco context manager

Gestione dell'eccezione: Qualcosa è andato storto
```

Il Modulo `contextlib`

Il modulo `contextlib` di Python fornisce utilità per la gestione dei context manager, inclusa una funzione di decoratore chiamata `@contextmanager` che permette di creare context manager utilizzando generatori. Questo rende la creazione di context manager più semplice e leggibile.

Ecco un esempio di utilizzo di `@contextmanager`:

```
from contextlib import contextmanager


@contextmanager

def mio_context_manager():
```

```
    print("Inizio del blocco context manager")
    try:
        yield
    finally:
        print("Fine del blocco context manager")

with mio_context_manager():
    print("All'interno del blocco with")
    # sollevare un'eccezione per testare la gestione delle eccezioni
    raise ValueError("Qualcosa è andato storto")
```

L'output sarà:

```
Inizio del blocco context manager

All'interno del blocco with

Fine del blocco context manager

Traceback (most recent call last):

 ...

ValueError: Qualcosa è andato storto
```

Utilizzo Avanzato dei Context Manager

I context manager possono essere utilizzati per una varietà di scopi avanzati, come la gestione dei lock nei programmi multithreading o la gestione delle transazioni nei database.

Ecco un esempio di utilizzo di un context manager per gestire un lock:

```
import threading

from contextlib import contextmanager


lock = threading.Lock()


@contextmanager

def gestisci_lock():

    lock.acquire()

    try:

       yield

    finally:

       lock.release()


def thread_task():

 with gestisci_lock():
```

```
      print(f"Lock acquisito nel thread
{threading.current_thread().name}")

      # Simula un lavoro che richiede il lock

      import time

      time.sleep(1)

      print(f"Lock rilasciato nel thread
{threading.current_thread().name}")



thread1 = threading.Thread(target=thread_task)

thread2 = threading.Thread(target=thread_task)


thread1.start()

thread2.start()



thread1.join()

thread2.join()
```

L'output mostrerà che i thread acquisiscono e rilasciano il lock in modo sicuro:

```
Lock acquisito nel thread Thread
-1

Lock rilasciato nel thread Thread
-1
```

```
Lock acquisito nel thread Thread
-2

Lock rilasciato nel thread Thread
-2
```

I context manager sono strumenti potenti che permettono di gestire risorse in modo sicuro e efficiente. Con la sintassi `with` e il modulo `contextlib`, è possibile creare e utilizzare context manager per una vasta gamma di applicazioni, migliorando la leggibilità e la robustezza del codice Python.

CAPITOLO 15: PROGETTI PRATICI

15.1 Creazione di un'applicazione CLI

Le applicazioni a linea di comando (CLI) sono strumenti potenti che permettono di interagire con i programmi tramite il terminale. Sono ampiamente utilizzate per script di automazione, strumenti di amministrazione di sistema e applicazioni di sviluppo. In questo capitolo, vedremo come sviluppare una semplice applicazione CLI utilizzando Python e la libreria `argparse`.

Introduzione a `argparse`

`argparse` è una libreria standard di Python che facilita la creazione di interfacce a linea di comando. Consente di definire gli argomenti che l'applicazione accetta e di generare automaticamente messaggi di aiuto.

Ecco un esempio di base di un'applicazione CLI che utilizza `argparse`:

```
import argparse


def main():

   parser = argparse.ArgumentParser(description='Esempio di
```

```
applicazione CLI.')
   parser.add_argument('nome', type=str, help='Il tuo nome')
   parser.add_argument('--eta', type=int, help='La tua età')

args = parser.parse_args()


print(f"Ciao, {args.nome}!")
 if args.eta:
    print(f"Hai {args.eta} anni.")

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

Per eseguire questa applicazione, si può utilizzare il terminale:

```
$ python cli_app.py Alice --eta 30
Ciao, Alice!
Hai 30 anni.
```

Progettazione dell'Applicazione CLI

Per creare un'applicazione CLI completa, è importante pianificare le funzionalità e come gli utenti interagiranno con l'applicazione. Nel nostro esempio, svilupperemo una CLI per gestire una lista di cose da fare (to-do list).

Le funzionalità principali saranno:

- Aggiungere un elemento alla lista
- Visualizzare tutti gli elementi
- Rimuovere un elemento

Implementazione della CLI To-Do List

• Struttura del Progetto

La struttura del progetto sarà la seguente:

cli_todo/
├── cli_todo.py
└── todo.py

• Definizione delle Funzionalità di Base

Iniziamo definendo le funzionalità di base nel file `todo.py`:

```
# todo.py

import json

import os

FILE_PATH = 'todo.json'


def carica_todo():

     if os.path.exists(FILE_PATH):
```

```
        with open(FILE_PATH, 'r') as file:
            return json.load(file)
    return []

def salva_todo(todo_list):
    with open(FILE_PATH, 'w') as file:
        json.dump(todo_list, file)

def aggiungi_todo(descrizione):
    todo_list = carica_todo()
    todo_list.append({'descrizione': descrizione,
'completato': False})
    salva_todo(todo_list)

def visualizza_todo():
 todo_list = carica_todo()
 for idx, todo in enumerate(todo_list, start=1):
     stato = '✔' if todo['completato'] else '✘'
     print(f"{idx}. {todo['descrizione']} [{stato}]")

def rimuovi_todo(indice):
    todo_list = carica_todo()
    if 0 < indice <= len(todo_list):
```

```
        del todo_list[indice - 1]

        salva_todo(todo_list)

 else:

      print("Indice non valido")
```

- **Integrazione con `argparse`**

Ora, integriamo queste funzionalità nella nostra applicazione CLI nel file `cli_todo.py`:

```
# cli_todo.py


import argparse

import todo



def main():

    parser =
argparse.ArgumentParser(description='Applicazione  CLI  To-Do
List.')

    subparsers = parser.add_subparsers(dest='comando')



parser_aggiungi      =      subparsers.add_parser('aggiungi',
help='Aggiungi
un nuovo to-do')

    parser_aggiungi.add_argument('descrizione', type=str,
help='Descrizione del to-do')



subparsers.add_parser('visualizza', help='Visualizza tutti i
to-do')
```

```
parser_rimuovi       =       subparsers.add_parser('rimuovi',
help='Rimuovi un
to-do')

    parser_rimuovi.add_argument('indice', type=int,
help='Indice del to-do da rimuovere')


    args = parser.parse_args()


if args.comando == 'aggiungi':

       todo.aggiungi_todo(args.descrizione)

    elif args.comando == 'visualizza':

       todo.visualizza_todo()

    elif args.comando == 'rimuovi':

       todo.rimuovi_todo(args.indice)

    else:

        parser.print_help()


if __name__ == "__main__":

    main()
```

- **Esecuzione dell'Applicazione CLI**

Ecco come utilizzare la nostra applicazione CLI To-Do List:

A. **Aggiungere un elemento:**

```
$ python cli_todo.py aggiungi "Compra il latte"
```

B. **Visualizzare tutti gli elementi:**

```
$ python cli_todo .py visualizza 1. Compra il latte [✘]
```

C. **Rimuovere un elemento:**

```
$ python cli_todo.py rimuovi 1
```

Abbiamo visto come creare un'applicazione CLI utilizzando Python e `argparse`. Abbiamo progettato una semplice applicazione per gestire una lista di cose da fare, implementando funzionalità per aggiungere, visualizzare e rimuovere elementi. Con queste basi, è possibile estendere l'applicazione aggiungendo ulteriori funzionalità come la modifica degli elementi, il completamento degli elementi e la persistenza dei dati in un database. I context manager possono essere utilizzati per gestire le risorse in modo sicuro ed efficiente, migliorando ulteriormente la robustezza dell'applicazione.

15.2 SVILUPPO DI UN 'APPLICAZIONE WEB COMPLETA

In questo capitolo, svilupperemo un'applicazione web completa utilizzando Flask, uno dei framework web più popolari per Python. Flask è leggero e flessibile, rendendolo ideale per progetti sia piccoli che grandi. La nostra applicazione sarà una semplice piattaforma di blog dove gli utenti possono leggere e pubblicare articoli.

Introduzione a Flask

Flask è un micro framework per lo sviluppo di applicazioni web in Python. È progettato per essere semplice e facile da usare, fornendo le basi necessarie per costruire applicazioni web.

Per installare Flask, usa pip:

```
pip install Flask
```

Creazione della Struttura del Progetto

Organizziamo il progetto con la seguente struttura:

```
blog/
├── app.py
├── models.py
├── forms.py
├── templates/
│ ├── base.html
│ ├── index.html
│ ├── new_post.html
└── static/
 ├── style.css
```

Configurazione di Flask

Iniziamo creando il file `app.py`, dove configureremo Flask e definiremo le route principali dell'applicazione.

```
from flask import Flask, render_template, redirect, url_for, request

from forms import PostForm

from models import Post, db
```

```
app = Flask(__name__)

app.config['SECRET_KEY'] = 'supersecretkey'

app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///blog.db'

app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False

db.init_app(app)


@app.before_first_request

def create_tables():

    db.create_all()


@app.route('/')

def index():

    posts = Post.query.all()

    return render_template('index.html', posts=posts)


@app.route('/new', methods=['GET', 'POST'])

def new_post():

    form = PostForm()

   if form.validate_on_submit():

      new_post = Post(title=form.title.data,
content=form.content.data)

      db.session.add(new_post)
```

```
        db.session.commit()

        return redirect(url_for('index'))

    return render_template('new_post.html', form=form)


if __name__ == '__main__':

    app.run(debug=True)
```

Definizione dei Modelli

Creiamo il file `models.py` per definire il modello di dati per i post del blog. Utilizzeremo SQLAlchemy, un ORM (Object-Relational Mapper) per gestire il database.

```
from flask_sqlalchemy import SQLAlchemy


db = SQLAlchemy()



class Post(db.Model):

    id = db.Column(db.Integer, primary_key=True)

    title = db.Column(db.String(100), nullable=False)

    content = db.Column(db.Text, nullable=False)

    date_posted = db.Column(db.DateTime,
default=db.func.current_timestamp())
```

Creazione dei Form

Creiamo il file `forms.py` per definire i form utilizzati nell'applicazione. Utilizzeremo Flask-WTF, un'estensione di Flask per l'integrazione con WTForms.

```
from flask_wtf import FlaskForm

from wtforms import StringField, TextAreaField, SubmitField

from wtforms.validators import DataRequired



class PostForm(FlaskForm):

      title = StringField('Title',
validators=[DataRequired()])

      content = TextAreaField('Content',
validators=[DataRequired()])

      submit = SubmitField('Post')
```

Creazione dei Template

Creiamo i template HTML nella cartella `templates`. Iniziamo con `base.html`, che sarà il layout di base per tutti i template.

- **`templates/base.html`**

```
<!DOCTYPE html>

<html lang="en">

<head>
```

```
     <meta charset="UTF-8">

     <title>{% block title %}Blog{% endblock
%}</title>

     <link rel="stylesheet" href="{{ url_for('static',
filename='style.css') }}">

</head>

<body>

    <div class="container">

          {% block content %}{% endblock %}

  </div>

</body>

</html>
```

- **templates/index.html**

```
{% extends 'base.html' %}


{% block title %}Home{% endblock %}


{% block content %}

   <h1>Blog Posts</h1>

   <a href="{{ url_for('new_post') }}">Create New
Post</a>

 <ul>
```

```
     {% for post in posts %}
       <li>
            <h2>{{ post.title }}</h2>
            <p>{{ post.content }}</p>
            <small>Posted on {{ post.date_posted
}}</small>
       </li>
     {% endfor %}
 </ul>
{% endblock %}
```

- **templates/new_post.html**

```
{% extends 'base.html' %}

{% block title %}New Post{% endblock %}

{% block content %}
   <h1>Create New Post</h1>
   <form method="POST">
      {{ form.hidden_tag() }}
```

```
      <p>
          {{ form.title.label }}<br>
          {{ form.title(size=32) }}
      </p>
      <p>
           {{ form.content.label }}<br>
           {{ form.content(cols=32, rows=4) }}
      </p>
      <p>{{ form.submit() }}</p>
    </form>
{% endblock %}
```

Stile CSS

Creiamo un file CSS di base per stilizzare la nostra applicazione. Salviamo questo file come `static/style.css`.

```
body {
 font-family: Arial, sans-serif;
 margin: 0;
 padding: 0;
 background-color: #f4f4f4;
}
```

```
.container {
 width: 80%;
 margin: auto;
 overflow: hidden;
}

h1, h2, h3 {
 color: #333;
}

ul {
 list-style: none;
 padding: 0;
}

li {
 background: #fff;
 margin: 20px 0;
 padding: 20px;
 border: 1px solid #ddd;
}

form {
 background: #fff;
```

```
 padding: 20px;
 margin-top: 20px;
 border: 1px solid #ddd;
}

input[type="text"], textarea {
 width: 100%;
 padding: 10px;
 margin: 10px 0;
}

input[type="submit"] {
 display: inline-block;
 background: #333;
 color: #fff;
 padding: 10px 20px;
 border: none;
 cursor: pointer;
}

input[type="submit"]:hover {
 background: #555;
```

```
}
```

Esecuzione dell'Applicazione

Per eseguire l'applicazione, utilizza il comando:

```
python app.py
```

Apri il browser e naviga a `http://127.0.0.1:5000/` per vedere l'applicazione in azione. Dovresti essere in grado di visualizzare i post del blog, creare nuovi post e vedere l'elenco aggiornato dei post.

Abbiamo sviluppato una semplice applicazione web utilizzando Flask, implementando una piattaforma di blog con funzionalità per creare e visualizzare post. Questa struttura può essere estesa ulteriormente per includere funzionalità avanzate come autenticazione degli utenti, gestione dei commenti e categorizzazione dei post. Flask, con la sua flessibilità e potenza, è una scelta eccellente per sviluppare applicazioni web sia piccole che grandi.

15.3 IMPLEMENTAZIONE DI UN PROGETTO DI DATA SCIENCE

In questo capitolo, guideremo attraverso l'implementazione di un progetto di data science completo, partendo dall'analisi dei dati fino alla visualizzazione dei risultati. Utilizzeremo Python e alcune delle sue librerie più popolari per la data science, tra cui Pandas, NumPy, Matplotlib e Scikit-learn. Il progetto che svilupperemo sarà una semplice analisi dei dati sui prezzi delle case, con lo scopo di creare un modello di machine learning per prevedere i prezzi delle case basato su diverse caratteristiche.

Raccolta e Caricamento dei Dati

Per questo progetto, utilizzeremo un dataset pubblico sui prezzi delle case. Puoi trovare dataset pertinenti su piattaforme come Kaggle. Assumiamo che abbiamo scaricato il dataset e lo abbiamo salvato come `house_prices.csv`.

- **Caricamento dei Dati**

Iniziamo caricando il dataset utilizzando Pandas:

```
import pandas as pd
```

```
# Carica il dataset

df = pd.read_csv('house_prices.csv')


# Visualizza le prime righe del dataset

print(df.head())
```

Esplorazione dei Dati

L'esplorazione dei dati è un passaggio cruciale in qualsiasi progetto di data science. Consente di comprendere meglio il dataset e di identificare eventuali problemi o pattern.

• Statistiche Descrittive

Utilizziamo Pandas per ottenere alcune statistiche descrittive di base:

```
# Statistiche descrittive

print(df.describe())


# Informazioni sul dataset

print(df.info())
```

- **Visualizzazione dei Dati**

Utilizziamo Matplotlib e Seaborn per creare alcune visualizzazioni dei dati:

```
import matplotlib.pyplot as plt

import seaborn as sns


# Istogramma dei prezzi delle case

plt.figure(figsize=(10, 6))

sns.histplot(df['price'], bins=30, kde=True)

plt.title('Distribuzione dei Prezzi delle Case')

plt.xlabel('Prezzo')

plt.ylabel('Frequenza')

plt.show()


# Scatter plot di superficie vs prezzo

plt.figure(figsize=(10, 6))

sns.scatterplot(x='sqft_living', y='price', data=df)

plt.title('Superficie vs Prezzo')

plt.xlabel('Superficie (sqft)')

plt.ylabel('Prezzo')
```

```
plt.show()
```

Preparazione dei Dati

Prima di costruire un modello di machine learning, dobbiamo preparare i dati. Questo passaggio include la gestione dei valori mancanti, la codifica delle variabili categoriche e la suddivisione dei dati in set di addestramento e test.

• Gestione dei Valori Mancanti

Verifichiamo la presenza di valori mancanti e li gestiamo in modo appropriato:

```
# Controlla i valori mancanti

print(df.isnull().sum())


# Ad esempio, riempiamo i valori mancanti con la mediana

df.fillna(df.median(), inplace=True)
```

• Codifica delle Variabili Categoriche

Convertiamo le variabili categoriche in numeriche:

```
# Converti variabili categoriche usando il metodo get_dummies

df = pd.get_dummies(df, drop_first=True)
```

- **Suddivisione dei Dati**

Suddividiamo i dati in set di addestramento e test:

```
from sklearn.model_selection import train_test_split


# Definisci le variabili indipendenti (X) e dipendenti (y)

X = df.drop('price', axis=1)

y = df['price']


# Suddividi i dati in set di addestramento e test

X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
```

Addestramento del Modello

Utilizziamo Scikit-learn per addestrare un modello di regressione lineare sui dati di addestramento:

```
from sklearn.linear_model import LinearRegression
```

```
# Crea un modello di regressione lineare

model = LinearRegression()


# Addestra il modello

model.fit(X_train, y_train)
```

Valutazione del Modello

Valutiamo il modello sui dati di test per vedere come si comporta:

```
from sklearn.metrics import mean_squared_error, r2_score


# Predici i prezzi delle case sui dati di test

y_pred = model.predict(X_test)


# Calcola l'errore quadratico medio (MSE) e il coefficiente
di
determinazione (R^2)

mse = mean_squared_error(y_test, y_pred)

r2 = r2_score(y_test, y_pred)


print(f"Mean Squared Error: {mse}")

print(f"R^2 Score: {r2}")
```

Visualizzazione dei Risultati

Visualizziamo i risultati per interpretare meglio le prestazioni del modello:

```
# Scatter plot dei prezzi reali vs prezzi predetti

plt.figure(figsize=(10, 6))

plt.scatter(y_test, y_pred, alpha=0.3)

plt.plot([y_test.min(), y_test.max()], [y_test.min(),
y_test.max()], '--r', lw=2)

plt.xlabel('Prezzi Reali')

plt.ylabel('Prezzi Predetti')

plt.title('Prezzi Reali vs Prezzi Predetti')

plt.show()
```

Abbiamo implementato un progetto di data science completo utilizzando Python. Questo processo ha incluso la raccolta e il caricamento dei dati, l'esplorazione e la visualizzazione dei dati, la preparazione dei dati, l'addestramento di un modello di machine learning, la valutazione del modello e la visualizzazione dei risultati. Seguendo questi passaggi, puoi applicare tecniche simili a una vasta gamma di problemi di data science.

CONCLUSIONE

Questo libro ha fornito una panoramica completa della programmazione in Python, coprendo una vasta gamma di argomenti che vanno dai concetti di base alle tecniche avanzate. Abbiamo iniziato con le basi della sintassi di Python, inclusi variabili, tipi di dati, operatori e strutture di controllo, elementi fondamentali per qualsiasi programmatore.

Successivamente, abbiamo esplorato le strutture dati come liste, tuple e dizionari, che sono essenziali per gestire e manipolare i dati in Python. Abbiamo anche affrontato i concetti avanzati di Programmazione Orientata agli Oggetti (OOP), inclusi classi, oggetti, ereditarietà, metodi speciali e MRO (Method Resolution Order). Questi concetti permettono di creare programmi più modulari e manutenibili.

Inoltre, abbiamo trattato la creazione di applicazioni CLI e web, fornendo esempi pratici su come sviluppare un'applicazione a linea di comando e una web app utilizzando Flask. Abbiamo anche affrontato l'implementazione di un progetto di data science, dalla raccolta e analisi dei dati alla costruzione di modelli di machine learning.

Infine, abbiamo esplorato concetti avanzati come i context manager, i decoratori e i generatori, che

consentono di scrivere codice Python più efficiente e leggibile. Questi strumenti avanzati sono fondamentali per sfruttare appieno la potenza e la flessibilità di Python.

Riassumendo, questo libro ha coperto:

1. **Sintassi di Base di Python**: Variabili, tipi di dati, operatori e strutture di controllo (`if`, `else`, `elif`).
2. **Strutture Dati**: Liste, tuple, dizionari e le loro operazioni.
3. **Programmazione Orientata agli Oggetti (OOP)**: Classi, oggetti, ereditarietà, metodi speciali, classi astratte e interfacce.
4. **Applicazioni Pratiche**: Sviluppo di applicazioni CLI, web app con Flask, e progetti di data science.
5. **Strumenti Avanzati**: Context manager, decoratori e generatori.

Speriamo che questo libro ti abbia fornito una solida base di conoscenze e competenze in Python, e che ti senta pronto ad affrontare progetti complessi con fiducia. Python è un linguaggio potente e versatile, e padroneggiarlo può aprire numerose opportunità nel campo della programmazione e della tecnologia.

Buona programmazione e continua a esplorare le infinite possibilità offerte da Python!