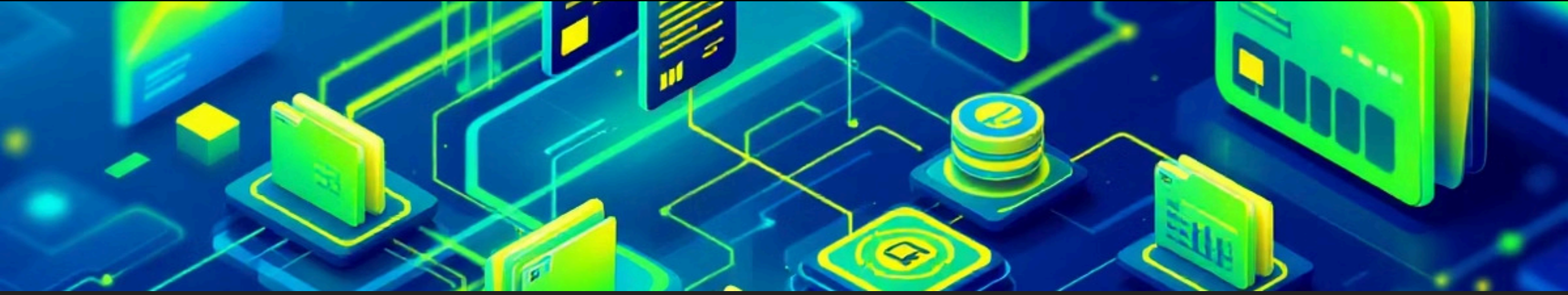
Gestione File e Gestione Errori in Python

Benvenuti in questo corso dedicato alla gestione dei file e alla gestione degli errori in Python. Questi argomenti rappresentano fondamenti essenziali per ogni sviluppatore Python, permettendo di creare applicazioni robuste e affidabili che interagiscono con il sistema operativo e gestiscono situazioni impreviste con eleganza.

Durante questa presentazione esploreremo le tecniche per manipolare file di diversi formati, implementare strategie di gestione degli errori e sviluppare codice più sicuro e maintainabile. Acquisiremo competenze pratiche immediatamente applicabili nei vostri progetti.



Introduzione alla Gestione dei File

Persistenza dei Dati

I file permettono di salvare informazioni oltre la durata del programma, garantendo la persistenza dei dati elaborati dalle nostre applicazioni.

Comunicazione tra Applicazioni

La gestione file facilita lo scambio di informazioni tra diversi programmi e sistemi, abilitando l'integrazione di soluzioni software complesse.

Elaborazione Dati

Molte applicazioni richiedono l'elaborazione di grandi quantità di dati memorizzati in file, dai log di sistema ai dataset per analisi statistiche.

La manipolazione dei file è una competenza fondamentale che permette alle applicazioni Python di interagire efficacemente con il sistema operativo e altri software, creando soluzioni integrate e funzionali.

Il Costrutto open() e le Modalità di Apertura

Modalità di Lettura

La modalità 'r' (read) apre il file in sola lettura. È la modalità predefinita e genera un errore se il file non esiste. Ideale per leggere configurazioni o dati esistenti.

```
file = open('dati.txt', 'r')
```

Modalità di Scrittura

La modalità 'w' (write) crea un nuovo file o sovrascrive completamente un file esistente. La modalità 'a' (append) aggiunge contenuto alla fine del file esistente.

```
file = open('output.txt', 'w')
file = open('log.txt', 'a')
```

Modalità Esclusiva

La modalità 'x' (exclusive creation) crea un nuovo file ma fallisce se il file esiste già. Utile per evitare sovrascritture accidentali di file importanti.

```
file = open('nuovo.txt', 'x')
```

Metodi di Lettura dei File

read()

Legge l'intero contenuto del file come singola stringa. Efficace per file piccoli ma può causare problemi di memoria con file molto grandi.

```
contenuto = file.read()
```

readline()

Legge una singola riga alla volta, incluso il carattere di fine riga. Perfetto per elaborare file riga per riga in modo efficiente.

```
riga = file.readline()
```

readlines()

Restituisce una lista contenente tutte le righe del file. Ogni elemento della lista corrisponde a una riga del file originale.

```
righe = file.readlines()
```

Metodi di Scrittura dei File

write()

Scrive una singola stringa nel file. Non aggiunge automaticamente caratteri di fine riga, offrendo controllo preciso sul formato di output.

writelines()

Scrive una sequenza di stringhe nel file. Ogni stringa della lista viene scritta consecutivamente senza separatori automatici.

flush()

Forza la scrittura immediata dei dati nel file, svuotando il buffer interno. Garantisce che le modifiche siano persistite immediatamente.

Questi metodi permettono un controllo granulare sulla scrittura dei dati, essenziale per generare file con formati specifici e garantire l'integrità delle informazioni salvate.

Apertura Sicura con il Costrutto with

Gestione Automatica delle Risorse

Il costrutto with garantisce la chiusura automatica del file anche in caso di errori durante l'elaborazione. Elimina la necessità di chiamare esplicitamente close().

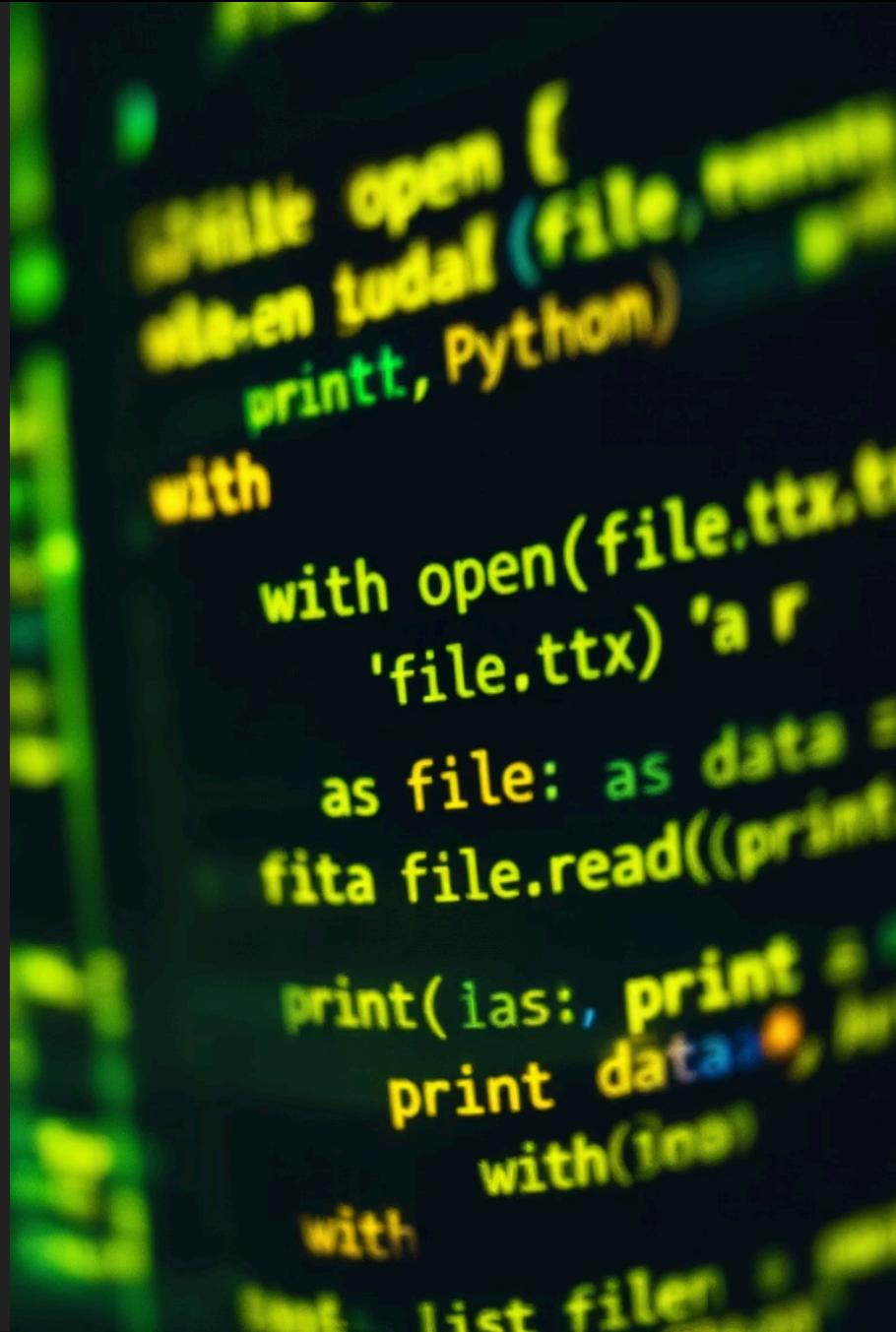
```
with open('file.txt', 'r') as f:
    contenuto = f.read()
```

Prevenzione Memory Leak

Senza la gestione appropriata, i file aperti possono causare memory leak e esaurimento delle risorse di sistema. Il with statement previene questi problemi automaticamente.

Codice Più Pulito

Riduce la verbosità del codice eliminando blocchi try/finally manuali. Il context manager gestisce internamente tutte le operazioni di pulizia necessarie.



Codifica dei File e Caratteri Speciali

Comprensione della Codifica

La codifica determina come i caratteri vengono rappresentati come byte nel file. UTF-8 è lo standard moderno che supporta tutti i caratteri Unicode, inclusi simboli speciali e lingue diverse.

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

Gestione Errori di Codifica

Quando si incontrano caratteri incompatibili, Python può gestire l'errore in diversi modi: ignore, replace, strict. La scelta dipende dal comportamento desiderato dell'applicazione.

```
f = open('file.txt', 'r', encoding='utf-8', errors='replace')
```

Best Practices

Specificare sempre esplicitamente la codifica per evitare comportamenti imprevisti su sistemi diversi. UTF-8 garantisce compatibilità universale e supporto per caratteri internazionali.

File CSV: Struttura e Utilizzo

Struttura Tabulare

I CSV organizzano dati in righe e colonne, con valori separati da virgole. Prima riga spesso contiene intestazioni delle colonne.

Interoperabilità

Formato universalmente supportato da fogli di calcolo, database e applicazioni di analisi dati. Facilita scambio informazioni tra sistemi diversi.

Semplicità

Formato testuale semplice, leggibile dall'uomo e facilmente editabile. Non richiede software specializzato per visualizzazione o modifica.

Configurabilità

Delimitatori personalizzabili (virgola, punto e virgola, tab). Gestione caratteri di escape per valori contenenti separatori.

Modulo csv: Lettura e Scrittura Avanzata

csv.reader()

Legge file CSV riga per riga, restituendo ogni riga come lista di valori. Gestisce automaticamente le virgolette e i caratteri di escape secondo le convenzioni CSV standard.

```
import csv
with open('dati.csv', 'r') as f:
    reader = csv.reader(f)
    for riga in reader:
        print(riga)
```

csv.DictReader()

Versione avanzata che restituisce ogni riga come dizionario, usando la prima riga come chiavi. Facilita l'accesso ai dati per nome colonna invece che per indice numerico.

```
with open('dati.csv', 'r') as f:
    reader = csv.DictReader(f)
    for riga in reader:
        print(riga['nome_colonna'])
```

Introduzione alle Eccezioni in Python

Gestione Robusta

Trasforma errori fatali in situazioni gestibili

Prevenzione Crash

Evita terminazioni improvvise del programma

Comunicazione Errori

Fornisce feedback significativo agli utenti

Debugging Facilitato

Identifica rapidamente problemi nel codice

Le eccezioni rappresentano il meccanismo standard di Python per gestire situazioni anomale durante l'esecuzione. Una gestione appropriata delle eccezioni distingue applicazioni professionali da script fragili, garantendo stabilità e user experience superiore.

Blocco try/except Fondamentale

try

Contiene codice potenzialmente problematico. Python esegue queste istruzioni monitorando eventuali eccezioni.

except

Definisce azioni da eseguire quando si verifica un'eccezione specifica. Permette recupero elegante da errori.

Esecuzione

Se nessuna eccezione: solo blocco try. Se eccezione: salto al blocco except corrispondente.

```
try:
    risultato = 10 / 0
except ZeroDivisionError:
    print("Impossibile dividere per zero")
    risultato = 0
```

Questa struttura base permette di intercettare errori specifici e implementare strategie di recupero appropriate, mantenendo l'applicazione stabile anche in presenza di input imprevisti o condizioni anomale.

Gestione di Eccezioni Multiple

Eccezioni Specifiche

Ogni tipo di errore richiede gestione dedicata

Ordine di Priorità

Eccezioni più specifiche prima di quelle generali

Catch-All Finale

Exception generale per errori imprevisti

```
try:
    file = open('dati.txt', 'r')
    numero = int(file.readline())
    risultato = 100 / numero
except FileNotFoundError:
    print("File non trovato")
except ValueError:
    print("Formato numero non valido")
except ZeroDivisionError:
    print("Divisione per zero")
except Exception as e:
    print(f"Errore imprevisto: {e}")
```

Blocchi else e finally

try

Codice principale da eseguire con monitoraggio eccezioni

except

Gestione errori specifici quando si verificano eccezioni

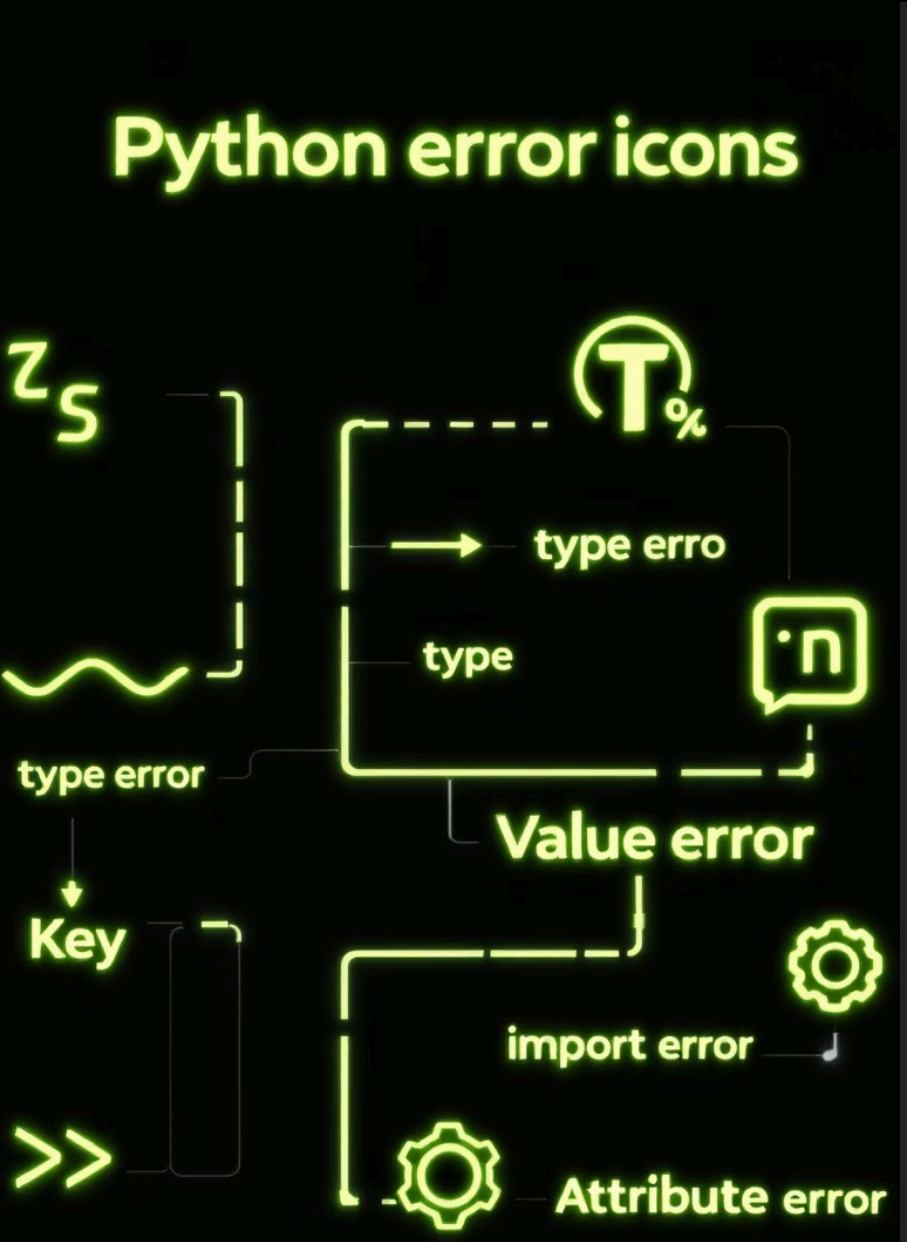
finally

Sempre eseguito, indipendentemente da errori o successo

else

Eseguito solo se NON si verificano eccezioni nel try

Il blocco else garantisce esecuzione di codice solo in caso di successo, mentre finally assicura operazioni di pulizia sempre necessarie come chiusura file o rilascio risorse, indipendentemente dall'esito dell'operazione principale.



Tipi Comuni di Errori in Python

FileNotFoundError

Si verifica quando si tenta di aprire un file inesistente. Comune nelle operazioni di lettura senza validazione preventiva dell'esistenza del file.

ValueError

Generato quando una funzione riceve argomento del tipo corretto ma valore inappropriato, come int('abc') o float('testo').

ZeroDivisionError

Eccezione matematica per divisioni o operazioni modulo per zero. Frequente in calcoli con input utente non validato.

KeyError

Sollevato quando si accede a chiave inesistente in dizionario. Prevenibile con get() o controlli di esistenza con 'in'.

Buone Pratiche nella Gestione degli Errori

1

Specificità

Cattura eccezioni specifiche invece di Exception generico per gestione mirata e debugging efficace

2

Logging

Registra sempre errori significativi con dettagli sufficienti per debugging futuro e monitoraggio applicazione

3

Recovery

Implementa strategie di recupero quando possibile, fornendo valori di fallback o operazioni alternative

4

User Experience

Mostra messaggi comprensibili agli utenti, nascondendo dettagli tecnici ma fornendo azioni correttive

Una gestione errori professionale combina prevenzione proattiva, recupero elegante e comunicazione chiara. Il codice robusto anticipa problemi comuni e fornisce percorsi alternativi, mantenendo l'applicazione funzionale anche in condizioni avverse. Documentate sempre le eccezioni gestite e i comportamenti attesi per facilitare manutenzione e debugging futuro.