Gestione dei File e delle Eccezioni in Python

Advanced Computer Programming

Prof. Luigi De Simone

Sommario

- Gestione dei file
- Gestione delle eccezioni
- Asserzioni

Riferimenti

- Tony Gaddis. **Introduzione a Python**. 5° ed. – Pearson, 2021
- Paul J. Deitel, Harvey M. Deitel. **Introduzione a Python. Per l'informatica e la data science.** Pearson, 2021
- **Python: How to Think Like a Computer Scientist interactive edition** https://runestone.academy/runestone/books/published/thinkcspy/index.html
- Allen Downey. **Think Python** - https://greenteapress.com/thinkpython2/thinkpython2.pdf

Gestione dei file in Python

- Quando eseguiamo un programma Python creiamo 3 **oggetti file** standard:
  - **`sys.stdin`**– standard input file object
  - **`sys.stdout`**– standard output file object
  - **`sys.stderr`**– standard error file object

- Esempio:

```
with open('accounts.txt', mode='w') as accounts:
    accounts.write('100 Jones 24.98\n’)
    accounts.write('200 Doe 345.67\n’)
    accounts.write('300 White 0.00\n’)
    accounts.write('400 Stone -42.16\n’)
    accounts.write('500 Rich 224.62\n')
```

L’istruzione **with**

- Le applicazioni **acquisiscono** risorse (e.g., file, connessioni di rete, connessioni a database, etc.) e dovremmo **rilasciare** le risorse non appena non sono più necessarie

- Python fornisce l’istruzione **`with`**:
  - **Esempio:**

```
with resource as r:
    DO_SOMETHING...
```

  - Il costrutto **`as`** è utilizzato per creare un alias in Python (uso **r** come alias di **resource**)
  - **Acquisisce una risorsa** (in questo caso, l'oggetto file per `accounts.txt`) e assegna il suo oggetto corrispondente a una variabile (in questo esempio account)
  - Consente all'applicazione di **utilizzare la risorsa tramite quella variabile**, e
  - **Chiama il metodo `close`** dell'oggetto risorsa per rilasciare la risorsa quando raggiungo la fine del blocco di istruzioni **with**

Le funzioni **`open`** e **`write`**

- La funzione **`open`** apre un file (e.g., accounts.txt) e lo associa a un oggetto file
- L'argomento ***mode*** specifica la modalità di apertura del file, indicando se aprire un file per la lettura dal file, per la scrittura nel file o per entrambi
- La modalità **'w'** apre il file per la scrittura, creando il file se non esiste:
  - Se non specifichiamo un percorso per il file, **Python lo crea nella cartella corrente**
  - L'apertura di un file per la scrittura **elimina tutti i dati esistenti nel file**
  - E' possibile utilizzare il metodo **`write`** sull'oggetto file per poter scrivere su file

Lettura da file

- E' possible **iterare su un oggetto file** grazie al costrutto `for`

```
for line in file
```

  dove leggiamo una riga alla volta dal file e la restituisce come stringa

- **Esempio**:

```
with open('accounts.txt', mode='r') as accounts:
    print(f'{"Account":<10}{"Name":<10}{"Balance":>10}')
    for record in accounts:
        account, name, balance = record.split()
        print(f'{account:<10}{name:<10}{balance:>10}')
```

Le funzioni **readlines** e **seek**

- Esiste anche Il metodo **`readlines()`** che può essere utilizzato per leggere un intero file di testo
- Il metodo restituisce ogni riga del file come **una stringa in una lista di stringhe**
- **ATTENZIONE:** Utilizzare `readlines` per file di grandi dimensioni può essere **un'operazione che richiede molto tempo**, e in tale tempo non possiamo utilizzare la lista di stringhe che viene restituita
- Il metodo **seek()** ci permette di posizionare il puntatore alla riga corrente
  - E.g.: **`file_object.seek(0)`**

Il modulo **os** per gestione file

- Il modulo **os** in Python consente di utilizzare **funzionalità dipendenti dal sistema operativo** in maniera semplice
- Per la gestione dei file possiamo utilizzare le funzioni di rimozione e rinomina

E.g.:

- Cancellare un file
  - `os.remove('accounts.txt')`
- Rinominare un file
  - `os.rename('temp_file.txt', 'accounts.txt')`

Formattazione dell'output

- Finora abbiamo incontrato **3 modi per scrivere valori**
  - Dichiarazioni di espressione
  - Funzione `print()`
  - Uso del metodo `write()` sugli oggetti file
- Per avere un controllo maggiore sulla formattazione dell'output rispetto alla semplice stampa di valori separati da spazi **esistono diversi modi in Python**
- E' possibile utlizzare i cosiddetti ***formatted string literals***. Per utilizzarli:
  - Iniziare una stringa con **f** o **F** prima del *single quote* (') o *double quote* (")
  - All'interno di questa stringa, si può scrivere un'espressione Python tra i caratteri **{ expression }** che può fare riferimento a variabili o a valori letterali
- Altri meccanismi sono definiti in https://docs.python.org/3/tutorial/inputoutput.html

Formattazione dell'output (Esempio)

```
import math
>>> print(f'The value of pi is approximately {math.pi:.3f}.')
The value of pi is approximately 3.142.
```

- Passando un numero intero dopo il ':', il campo sarà "*largo*" un numero minimo di caratteri. Questo è utile per allineare le colonne.

```
table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print(f'{name:10} ==> {phone:10d}')
...
Sjoerd     ==>       4127
Jack       ==>       4098
Dcab       ==>       7678
```

Eccezioni

- Accedere al di là dei limiti di una lista

  `test = [1,2,3]` then `test[4]` → `IndexError`

- Conversione in un tipo non appropriato

  `int(test)` → `TypeError`

- Riferirsi ad una variabile non esistente

  `a` → `NameError`

- Mischiare tipi di dato inappropriatamente

  `'3'/4` → `TypeError`

- Dimenticare di chiudere parentesi, apici (singoli e doppi), etc.

```
a = len([1,2,3]
print(a)
```

→ `SyntaxError`

Eccezioni e Asserzioni

- Cosa accade quando una funzione esegue del codice che produce **condizioni non attese**?
- Otteniamo una eccezione (**exception**)
  - Accedere al di là dei limiti di una lista

```
test = [1,7,4]
test[4]
```
→ `IndexError`

  - Conversioni a tipi inappropriati

```
int(test)
```
→ `TypeError`

  - Riferirsi a variabili non esistenti

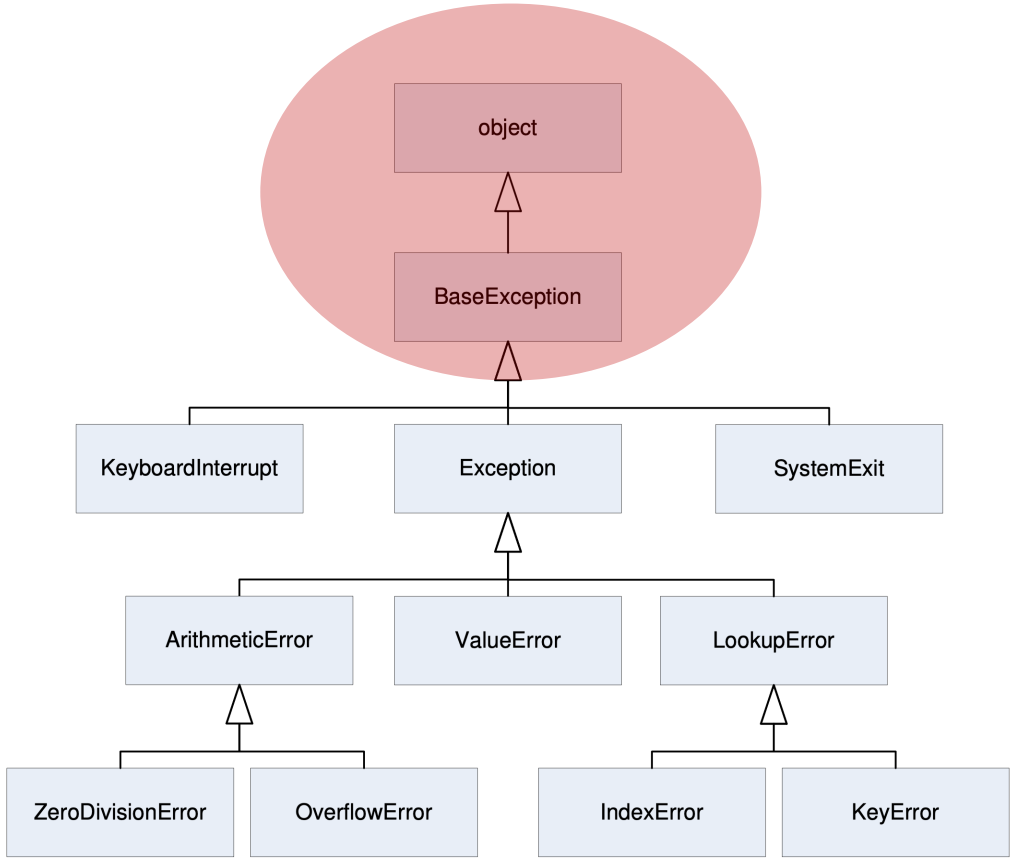
```
a
```
→ `NameError`

  - Operare su tipi di dati misti senza cast

```
'a'/4
```
→ `TypeError`

Altri tipi di eccezione

- Tipi di errori comuni:
  - `SyntaxError`: Python can't parse program
  - `NameError`: local or global name not found
  - `AttributeError`: attribute reference fails
  - `TypeError`: operand doesn't have correct type
  - `ValueError`: operand type okay, but value is illegal
  - `IOError`: IO system reports malfunction (e.g., file not found)



Gestire le eccezioni

- Il codice Python può fornire **handlers** per le eccezioni

```
try:
    a = int(input("Tell me one number:"))
    b = int(input("Tell me another number:"))
    print(a/b)
except:
    print("Bug in user input.")
```

- Le eccezioni sono
  - **sollevate** da qualunque istruzione nel body del blocco **`try`**
  - **gestite** dalle istruzioni nel blocco **`except`** e l'esecuzione continua nel body dell'istruzione `except`

Gestire eccezioni specifiche

- E' raccomandato avere **separare le clausole `except`** per gestire un tipo particolare di eccezione
- I blocchi `except` devono essere posizionati dal più specifico al più generico, in base alla gerarchia delle eccezioni

```
try:
    a = int(input("Tell me one number: "))
    b = int(input("Tell me another number: "))
    print("a/b = ", a/b)
    print("a+b = ", a+b)
except ValueError:
    print("Could not convert to a number.")
except ZeroDivisionError:
    print("Can't divide by zero")
except:
    print("Something went very wrong.")
```

only execute if these errors come up

for all other errors

Altri tipi di eccezione

- **`else:`**
  - Il body di questo ramo è eseguito quando l'esecuzione del blocco `try` associato **completa senza eccezioni**
- **`finally`**:
  - Il body di questo ramo è **sempre eseguito** dopo il blocco `try`, `else` e `except`, anche se si sollevano altri errori o si esegue un `break`, `continue` o `return`
  - E' utile per invocare codice di ***clean-up*** che dovrebbero essere eseguito a prescindere da quello che è accaduto (e.g., chiudere un file)

Cosa fare con le eccezioni?

- Cosa fare quando incontriamo degli errori?
- **Fallire in maniera silente**:
  - Sostituire i valori di default oppure continuare e basta
  - ma l'utente finale non sarà avvisato con nessun warning
- Ritornare un valore di **"errore"**
  - Che valore scegliere?
  - Potremmo complicare il codice
- Fermare l'esecuzione, **segnalare una condizione di errore**
  - in Python: **raise an exception** (analogo al `throw` in Java)

```
raise Exception("descriptive string")
```

L'istruzione **raise**

- Non ritornare valori speciali quando abbiamo un errore e poi controlliamo se questo "valore d'errore" viene ritornato
- Invece, **sollevare un'eccezione** quando non siamo capaci di fornire un risultato coerente con la specifica di una funzione

```
raise <exceptionName>(<arguments>)
```

```
raise ValueError("something is wrong")
```

keyword

name of error you want to raise

optional, but typically a string with a message

Esempio: sollevare una eccezione

```
def get_ratios(L1, L2):
    """ Assumes: L1 and L2 are lists of equal length of numbers
        Returns: a list containing L1[i]/L2[i] """
    ratios = []
    for index in range(len(L1)):
        try:
            ratios.append(L1[index]/L2[index])
        except ZeroDivisionError:
            ratios.append(float('nan')) #nan = not a number
        except:
            raise ValueError('get_ratios called with bad arg')
    return ratios
```

manage flow of program by raising own error

Gestione delle eccezioni: **Esempio**

- Assumiamo di avere una **lista** che descrive uno studente e voti presi a degli esami
- Ogni elemento è una lista composta di due parti
  - una **lista** che include **nome** e **cognome** di uno studente
  - una **lista** dei **voti** a degli esercizi

```
test_grades = [['peter', 'parker'], [80.0, 70.0, 85.0]],
               [['bruce', 'wayne'], [100.0, 80.0, 74.0]]]
```

- Creiamo una **nuova lista**, con nome/cognome, voto, **e media**

```
[['peter', 'parker'], [80.0, 70.0, 85.0], 78.33333],
[['bruce', 'wayne'], [100.0, 80.0, 74.0], 84.666667]]
```

Gestione delle eccezioni: **Esempio**

```
# ritorna le statistiche per soggetto
def get_stats(class_list):
    new_stats = []
    for elt in class_list:
        new_stats.append([elt[0], elt[1], avg(elt[1])])
    return new_stats

# calcola la media dei punteggi
def avg(grades):
    return sum(grades)/len(grades)
```

Gestione delle eccezioni: **Esempio**

- Se uno o più studenti **non hanno voti**, otteniamo un errore!

```
test_grades = [[['peter', 'parker'], [10.0, 5.0, 85.0]],
               [['bruce', 'wayne'], [10.0, 8.0, 74.0]],
               [['captain', 'america'], [8.0,10.0,96.0]],
               [['deadpool'], []]]
```

- `ZeroDivisionError: float division by zero`

Perché…

```
return sum(grades)/len(grades)
```

length is 0

Opzione 1: Gestiamo l'eccezione direttamente caso 1

- Decidiamo di **notificare** che qualcosa è andato storto

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError:
        print('warning: no grades data')
```

- Eseguendo:

```
warning: no grades data
[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], None]]
```

flagged the error

because `avg` did not return anything in the except

Opzione 2: Gestiamo l'eccezione direttamente caso 2

Decidiamo che uno studente con nessun voto ritorni **zero** per avg

```
def avg(grades):
try:
        return sum(grades)/len(grades)
except ZeroDivisionError:
        print('warning: no grades data')
        return 0.0
```

- Eseguendo:

```
warning: no grades data
[['peter', 'parker'], [10.0, 5.0, 85.0], 15.41666666],
[['bruce', 'wayne'], [10.0, 8.0, 74.0], 13.83333334],
[['captain', 'america'], [8.0, 10.0, 96.0], 17.5],
[['deadpool'], [], 0.0]]
```

still flag the error

now `avg` returns 0

Opzione 3: Gestiamo l'eccezione indirettamente

- Decidiamo di risollevare l'eccezione intercettata (raise) tramite il costrutto **`as`**

```
def avg(grades):
    try:
        return sum(grades)/len(grades)
    except ZeroDivisionError as e:
        raise e
```

Eseguendo:

```
ZeroDivisionError Traceback (most recent call last) Cell In[2], line 42
. . .
ZeroDivisionError: division by zero
```

Asserzioni in Python

- Vogliamo essere sicuri che certe **assunzioni** su una particolare condizione siano come attese (siano vere)
- Se una certa condizione **non risulta vera**, Python consente, attraverso **`assert`**, di sollevare un'eccezione di tipo `AssertionError`
- Buon esempio di **defensive programming**

Asserzioni: **Esempio**

```
def avg(grades):
    assert len(grades) != 0, 'no grades data'
    return sum(grades)/len(grades)
```

function ends immediately if assertion not met

- Solleviamo un `AssertionError` se per esempio abbiamo una lista vuota (`len(grades) == 0`) per i voti degli studenti
- Altrimenti l'esecuzione procede normalmente

Asserzioni per *programmazione difensiva*

- Le asserzioni non consentono al programmatore di controllare la risposta a condizioni inattese
- Assicurano di **terminare l'esecuzione** non appena la condizione nell'asserzione fallisce
- Tipicamente sono usate per **controllare gli input** di una funzione, ma possono essere usare anche per altri motivi
- Possono essere utilizzate per **controllare gli output** di una funzione per evitare la propagazione di valori errati
- Possono facilitare la ricerca di bug

Dove usare le asserzioni?

- L'obiettivo è trovare bug il prima possibile ed eliminarli quando si palesano
- Per complementare il **testing**
- Sollevare **exceptions** se l'utente fornisce valori errati in input
- Usare le **assertions per**
  - Controllare i **tipi** dei parametri/valori
  - Controllare se esistono **invarianti** su strutture dati
  - Controllare **il rispetto di vincoli** sui valori di ritorno
  - Controllare **violazioni** di vincoli su funzioni (e.g., nessun duplicato in una lista)