

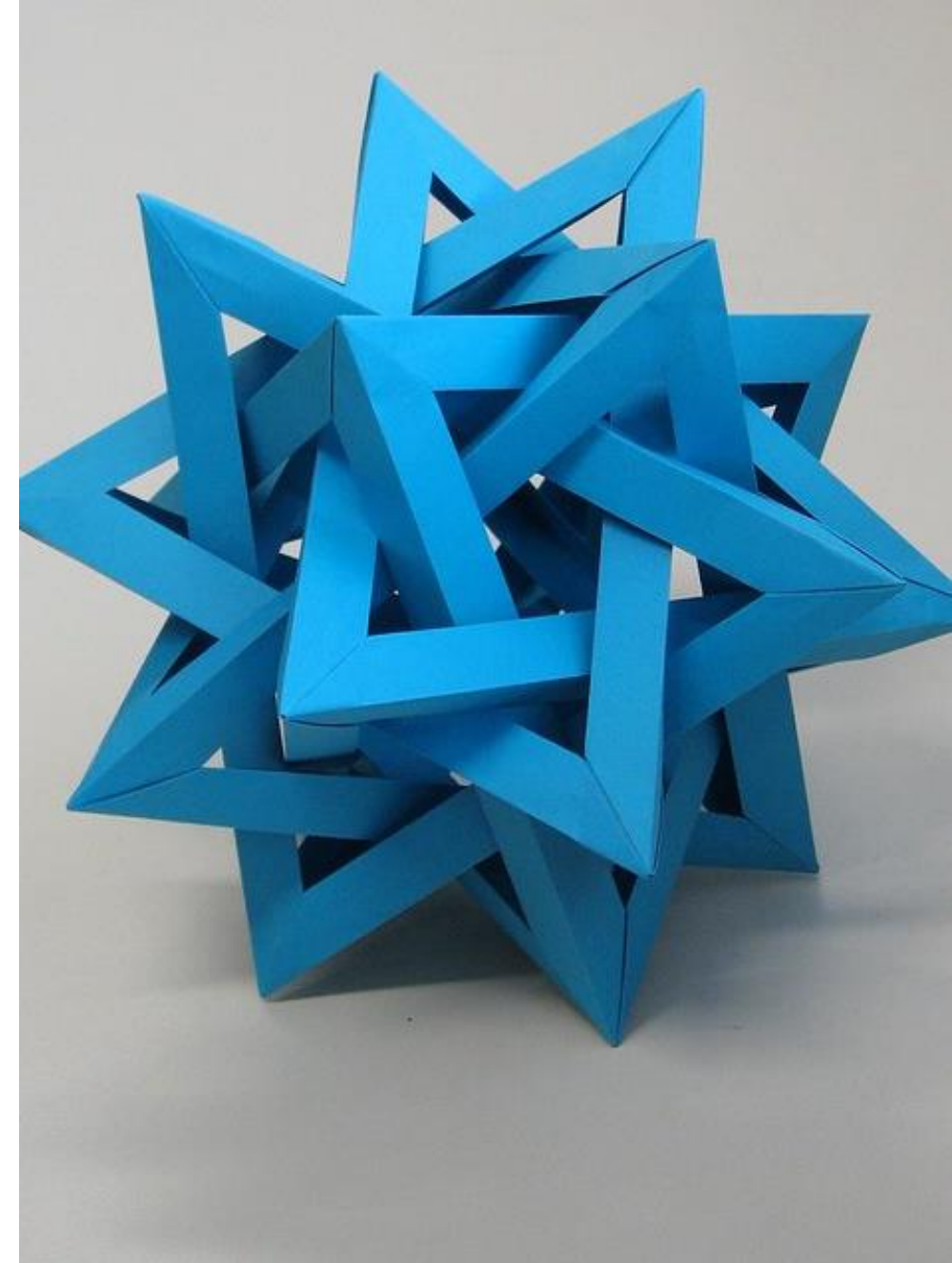


Unità P7: File e Gestione Errori

LEGGERE, ANALIZZARE E SCRIVERE FILE.
GESTIRE ERRORI ED ECCEZIONI.



Chapter 7



Obiettivi dell'Unità

- Leggere e scrivere file di testo
- Elaborare sequenze di dati provenienti da file
- Generare e gestire errori ed eccezioni

In questa unità si apprenderà come scrivere programmi che manipolano i file

Leggere e scrivere file di testo



7.1

Leggere e scrivere file di testo

- I file di testo sono usati molto spesso per memorizzare informazioni
 - Sono il tipo più 'portabile' di file di dati
- Esempi di file di testo comprendono
 - File creati con semplici editor di testo, come Blocco Note di Windows
 - Codice sorgente in Python
 - File HTML
 - File CSV (comma-separated values: valori separati da virgole)
 - ...

Aprire file: Lettura

- Per accedere ad un file, occorre innanzitutto **aprirlo**
- Supponiamo di dover leggere dei dati da un file chiamato `input.txt`, salvato nella stessa cartella del nostro programma
- Per aprire un file per la lettura, bisogna fornire il nome del file come primo argomento alla funzione **open**, e la stringa **"r"** come secondo argomento:

```
infile = open("input.txt", "r")
```

- Viene restituito un “oggetto file”, che useremo per leggere/scrivere il contenuto

Aprire file: Lettura (2)

- Importante, da ricordare:

- Quando si apre un file in modalità di lettura, **il file deve esistere** (ed essere accessibile all'utente), o si verificherà un'*eccezione*
- L'oggetto file restituito dalla funzione `open` **deve essere salvato** in una variabile
 - Tutte le successive operazioni per accedere al contenuto del file verranno eseguite tramite questo oggetto file.

```
infile = open("input.txt", "r")
```

Aprire file: Scrittura

- Per aprire un file per la scrittura, si fornisce il nome del file come primo argomento alla funzione `open`, e la stringa `"w"` come secondo argomento:

```
outfile = open("output.txt", "w")
```

- Se il file di output non esiste ancora, viene creato un file vuoto
- Se il file di output esiste già, il suo contenuto viene svuotato

Chiudere file: importante

- Dopo avere finito di elaborare (leggere/scrivere) i dati presenti in un file, assicurarsi di **chiudere il file** usando il metodo `close()` :

```
infile.close()  
outfile.close()
```

- Se il programma dovesse uscire senza chiudere un file aperto in scrittura, l'output potrebbe non essere salvato completamente nel file

Sintassi: aprire e chiudere file

Sintassi 7.1 Apertura e chiusura di file

Esempio

Memorizza in variabili gli oggetti restituiti, di tipo file.

Il nome del file da aprire

```
infile = open("input.txt", "r")
```

```
outfile = open("output.txt", "w")
```

Specifica la modalità di apertura:
"r" per leggere,
"w" per scrivere.

Chiude i file dopo l'elaborazione.

```
# Leggi dati da infile.
```

```
# Scrivi dati in outfile.
```

```
infile.close()
```

```
outfile.close()
```

Se non si chiude un file aperto in scrittura, può darsi che alcuni dati non siano stati scritti nel file.

Modi di apertura dei file

Mode	Description
r	Opens a file for reading. (default)
w	Opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Opens a file for exclusive creation. If the file already exists, the operation fails.
a	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	Opens in text mode. (default)
b	Opens in binary mode.
+	Opens a file for updating (reading and writing)

Leggere da un file

- All'apertura del file, un «cursore» immaginario viene posizionato all'inizio del file
- Per leggere **una linea (riga) di testo** da un file, chiamare il metodo `readline()` sull'oggetto file ottenuto all'apertura del file:

```
line = infile.readline()
```

- `readline()` legge il testo, partendo dalla posizione attuale del cursore, e continua **fino a quando non incontra la fine della linea**
 - La fine della linea è identificata dal carattere '`\n`'
 - Il cursore viene quindi spostato all'inizio della linea successiva

Leggere da un file (2)

- Supponiamo per esempio che `input.txt` contenga le linee
`flying`
`circus`
- La prima chiamata a `readline()` restituisce la stringa
`"flying\n"`
 - Ricordiamo che `\n` rappresenta il carattere di «a capo» (newline) che indica la fine della linea
- Chiamando `readline()` una seconda volta, restituirà la stringa
`"circus\n"`

Leggere da un file (3)

- Se chiamassimo ulteriormente `readline()`, verrebbe fornita la stringa vuota `""` poiché si è raggiunta la **fine del file**
- Se il file contenesse una riga vuota, in tal caso `readline()` restituirebbe una stringa che contiene unicamente il carattere newline `"\n"`

Leggere più righe da un file

- Si legge **ripetutamente** una singola riga dal file, **finché** non raggiungiamo il valore sentinella
- Il **valore sentinella è la stringa vuota**, che viene generata dal metodo `readline()` dopo aver raggiunto la **fine del file**

```
line = infile.readline()
while line != "" :
    # Process the line.
    line = infile.readline()
```

Convertire i valori letti da file

- Come con la funzione `input()`, i dati letti dal metodo `readline()` sono esclusivamente **stringhe**
- Se il file contenesse dati **numerici**, le stringhe dovranno essere **convertite** in valori numerici usando le note funzioni `int()` o `float()` :

```
# un singolo dato floating point su ogni linea  
value = float(line)
```

- Il carattere di fine linea al fondo della stringa viene ignorato dalle funzioni di conversione, senza generare errori

Scrivere su un file

- Si può scrivere in un file (aperto in scrittura) utilizzando il metodo `write()` :

```
outfile.write("Hello, World!\n")
```

- A differenza di `print()`, quando si scrive del testo su un file di output, occorre **aggiungere esplicitamente il carattere di fine linea**, per poter passare alla linea successiva
- Può essere comodo usare il metodo `write()` con **stringhe formattate**:

```
outfile.write("Number of entries: %d\nTotal: %8.2f\n" % (count, total))
```

```
outfile.write(f"Number of entries: {count}\nTotal: {total:8.2f}\n")
```


Esempio: Lettura e scrittura di file

- Supponiamo di avere un file di testo che contiene una sequenza di numeri in virgola mobile, memorizzati uno per linea
- Si vogliono leggere tali valori e scriverli in un nuovo file di output, scrivendoli in modo ben allineato (incolonnando i punti decimali), e seguiti dal loro valore totale e dal loro valore medio
- Ad esempio, il file di input può avere il seguente contenuto:

32.0

54.0

67.5

80.25

115.0

Esempio: Lettura e scrittura di file (2)

- Il file di **output** dovrà contenere

```
32.00
54.00
67.50
80.25
115.00
-----
Total: 348.75
Average: 69.75
```

Esempio

- Aprire il file total.py

Errore frequente

- Backslash (barra rovesciata) nei nomi di file
 - Quando si usa una stringa costante che contiene un percorso di file (che può comprendere il carattere `\`), occorre raddoppiare ciascun `\\`:

```
infile = open("c:\\homework\\input.txt", "r")
```

- Infatti la barra rovesciata `\` viene interpretata come **carattere di 'escape'**, che modifica l'interpretazione del carattere successivo
 - Ad esempio `\n` rappresenta l'a-capo
- Se un utente inserisce il nome di file come `input()` al programma, **non** è necessario raddoppiare la barra rovesciata

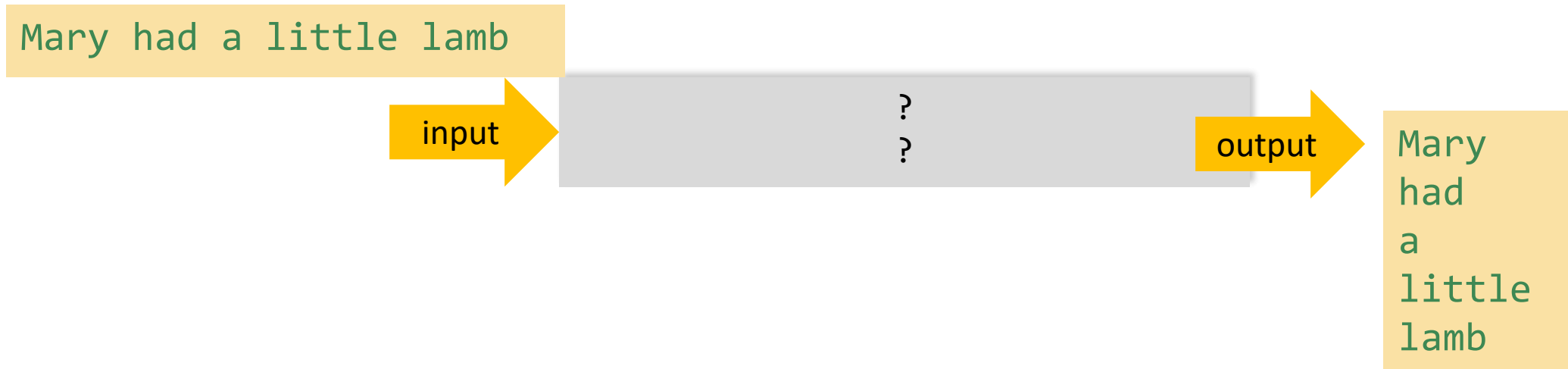
Input e Output testuali



7.2

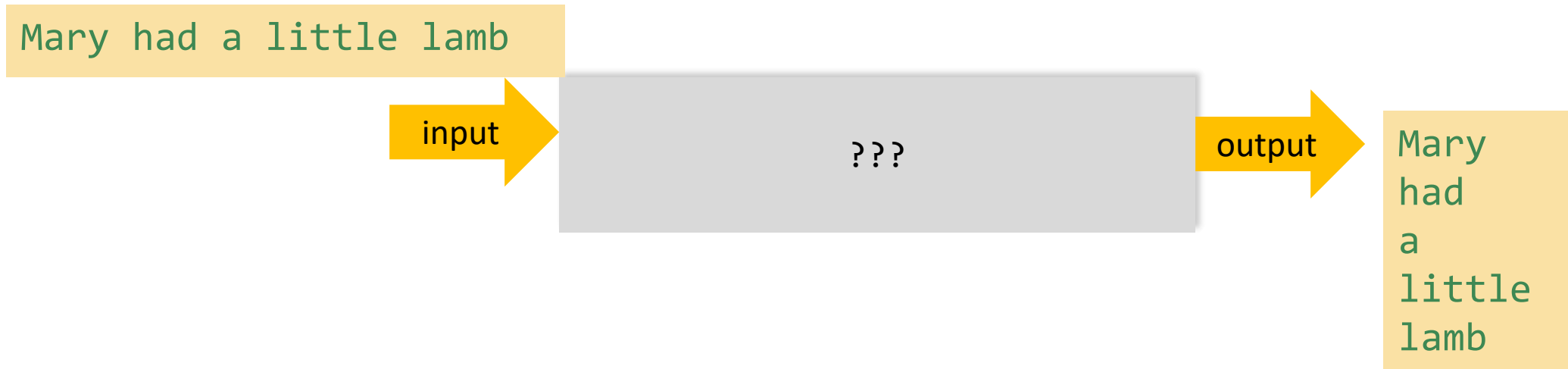
Input e Output testuale

- Come elaborare file di testo con contenuti complessi?
- Come gestire le sfide che spesso emergono lavorando con dati reali?
- Esempio: leggere le **singole parole** di un testo e salvarle una per riga



Input e Output testuale

- Come elaborare file di testo con contenuti complessi?
- Come gestire le sfide che spesso emergono lavorando con dati reali?
- Esempio: leggere le **single parole** di un testo e salvarle una per riga



Elaborare input testuali

- A seconda dei casi, può essere necessario elaborare il testo:
 - Un parola per volta
 - Una linea per volta
 - Un carattere per volta
- Python fornisce diversi metodi, come: `read()`, `split()` e `strip()` per aiutarci in queste situazioni

L'elaborazione di input testuali è necessaria in praticamente tutti i programmi che devono interagire con l'utente

Input per linee «automatico»

- Python può trattare un file aperto in input come se fosse **un contenitore di stringhe**, in cui ciascuna **linea** corrisponde ad una singola **stringa**
 - È come se la `readline()` fosse «implicita»
 - In quanto contenitore, possiamo usarlo in un ciclo `for...in`

- Esempio: leggi e stampa tutte le linee del file

```
for line in infile :  
    print(line)
```

- All'inizio di ciascuna iterazione del ciclo, alla variabile `line` viene assegnato il valore di una stringa che contiene la linea successiva nel file
- Il file è un tipo particolare di contenitore
 - Dopo avere letto il file, occorre **chiuderlo e riaprirlo**, se si vuole iterare nuovamente su di esso

Esempio di lettura di un file

- Consideriamo un file che contiene un insieme di parole, una per riga:

spam

and

eggs

Eliminare i fine-linea (1)

- Ricordiamo che ciascuna linea letta termina con un carattere newline (`\n`)
- Normalmente si vuole rimuovere tale carattere, prima di usare la stringa ottenuta
- Dopo aver detto la prima linea, la variabile contiene:

s p a m \n

Eliminare i fine-linea (2)

- Per eliminare il carattere newline si può usare il metodo `rstrip()` sulla stringa:

```
line = line.rstrip()
```

- Risultato:

```
s p a m
```

- Attenzione: saranno cancellati anche eventuali spazi terminali
 - `line.rstrip('\n')` elimina solo il newline

`str.rstrip([chars])`

Return a copy of the string with trailing characters removed. The `chars` argument is a string specifying the set of characters to be removed. If omitted or `None`, the `chars` argument defaults to removing whitespace. The `chars` argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '  spacious  '.rstrip()
'  spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

<https://docs.python.org/3/library/stdtypes.html#str.rstrip>

Metodi per eliminare caratteri iniziali/finali

Tabella 1
Metodi per eliminare
caratteri da una stringa

Metodo	Restituisce
<code>s.lstrip()</code> <code>s.lstrip(caratteri)</code>	Una nuova versione di <i>s</i> in cui eventuali caratteri di spaziatura (spazi, caratteri di tabulazione e <i>newline</i>) sono stati eliminati <i>a sinistra</i> , cioè all'inizio, di <i>s</i> (la lettera <i>l</i> di <code>lstrip</code> sta, appunto, per <i>left</i> , sinistra). Se è presente la stringa <i>caratteri</i> , vengono eliminati i caratteri presenti in essa invece dei caratteri di spaziatura.
<code>s.rstrip()</code> <code>s.rstrip(caratteri)</code>	Come <code>lstrip</code> , ma i caratteri vengono eliminati <i>a destra</i> , cioè alla fine, di <i>s</i> (la lettera <i>r</i> di <code>rstrip</code> sta per <i>right</i> , destra).
<code>s.strip()</code> <code>s.strip(caratteri)</code>	Simile a <code>lstrip</code> e <code>rstrip</code> , ma i caratteri vengono eliminati tanto a sinistra quanto a destra di <i>s</i> .

Esempi di eliminazione caratteri

Enunciati	Risultato	Commento
<pre>string = "James\n" result = string.rstrip()</pre>	J a m e s	Il carattere <i>newline</i> alla fine della stringa è stato eliminato.
<pre>string = "James \n" result = string.rstrip()</pre>	J a m e s	Anche lo spazio alla fine della stringa è stato eliminato.
<pre>string = "James \n" result = string.rstrip("\n")</pre>	J a m e s	È stato eliminato soltanto il carattere <i>newline</i> .
<pre>name = " Mary " result = name.strip()</pre>	M a r y	I caratteri di spaziatura sono stati eliminati tanto all'inizio quanto alla fine della stringa.
<pre>name = " Mary " result = name.lstrip()</pre>	M a r y	I caratteri di spaziatura sono stati eliminati soltanto all'inizio della stringa.

Leggere parole

- Talvolta è necessario leggere le singole parole da un file di testo
- Per esempio, supponiamo che il file contenga due linee di testo
`Mary had a little lamb,`
`whose fleece was white as snow`

Leggere parole (2)

- Vorremmo stampare a video le parole, una per riga

Mary

had

a

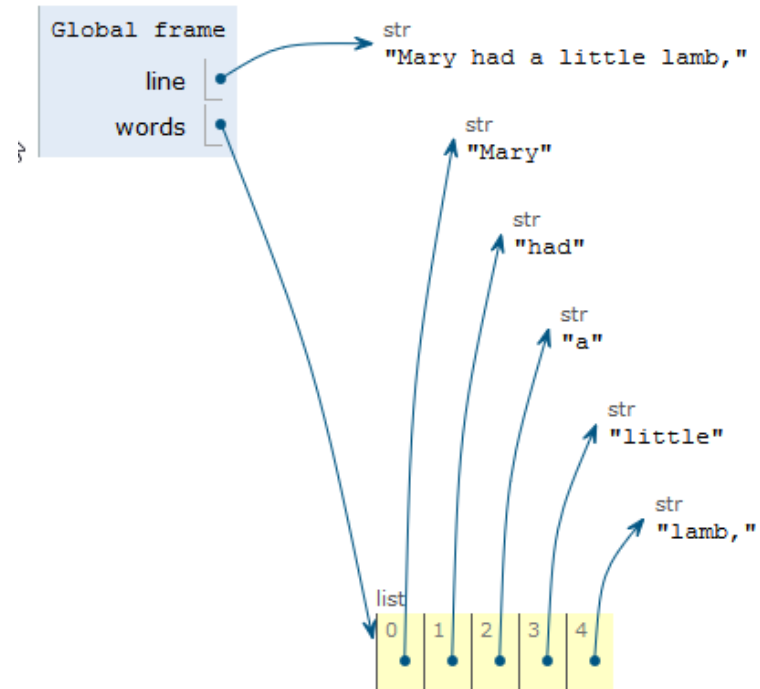
little

. . .

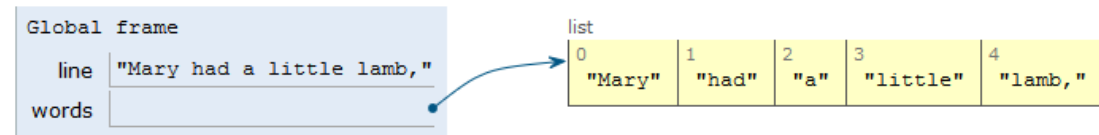
- Poiché non esiste alcun metodo per leggere una singola parola, occorre leggere una **linea** e poi dividerla nelle **parole** componenti

```
wordlist = line.split()
```


PythonTutor



```
line = 'Mary had a little lamb,'  
words = line.split()
```



(può essere visualizzata in modi diversi, i dati sono gli stessi)

Leggere parole (3)

- Si noti che alcune parole (l'ultima ad esempio) possono contenere della **punteggiatura** (es. la virgola)
- Volendo **isolare** le parole dai segni di punteggiatura, possiamo cancellarli con il metodo **rstrip()**:

```
word = word.rstrip(". , ? !")
```

Leggere parole: esempio completo

```
inputFile = open("lyrics.txt", "r")
for line in inputFile :
    wordList = line.split()
    for word in wordList :
        word = word.rstrip(".,?!")
        print(word)

inputFile.close()
```

Esempio

- Aprire il file lyrics.py

Metodi per suddividere una stringa

Metodo	Restituisce
<code>s.split()</code> <code>s.split(<i>sep</i>)</code> <code>s.split(<i>sep</i>, <i>maxsplit</i>)</code>	Una lista contenente le parole estratte dalla stringa <i>s</i> . Se viene fornita la stringa <i>sep</i> , questa viene usata come delimitatore, altrimenti si usa una sequenza di spazi di qualsiasi lunghezza. Se è presente l'argomento <i>maxsplit</i> , questo sarà il numero massimo di suddivisioni eseguite, generando una lista di lunghezza massima <i>maxsplit</i> + 1.
<code>s.rsplit(<i>sep</i>, <i>maxsplit</i>)</code>	Come <code>split</code> , ma le suddivisioni vengono effettuate partendo dalla fine della stringa anziché dall'inizio.
<code>s.splitlines()</code>	Una lista contenente le singole righe della stringa <i>s</i> , che viene, quindi, suddivisa usando come delimitatore il carattere <code>\n</code> .

Esempi di suddivisione di una stringa

Enunciati	Risultato	Commento
<pre>string = "a,bc,d" string.split(",")</pre>	<pre>"a" "bc" "d"</pre>	La stringa viene suddivisa in corrispondenza di ciascuna virgola.
<pre>string = "a b c" string.split()</pre>	<pre>"a" "b" "c"</pre>	La stringa viene suddivisa usando lo spazio come delimitatore e spazi consecutivi sono considerati come un unico separatore.
<pre>string = "a b c" string.split(" ")</pre>	<pre>"a" "b" "" "c"</pre>	La stringa viene suddivisa usando lo spazio come delimitatore, ma, essendoci un argomento esplicito, spazi consecutivi vengono considerati come delimitatori consecutivi.
<pre>string = "a:bc:d" string.split(":", 1)</pre>	<pre>"a" "bc:d"</pre>	La stringa viene suddivisa in due parti, partendo dall'inizio: la suddivisione avviene, quindi, in corrispondenza del primo carattere "due punti".
<pre>string = "a:bc:d" string.rsplit(":", 1)</pre>	<pre>"a:bc" "d"</pre>	La stringa viene suddivisa in due parti, partendo dalla fine: la suddivisione avviene, quindi, in corrispondenza dell'ultimo carattere "due punti".

Leggere caratteri

- Il metodo `read()` può ricevere un **argomento** che specifica il **numero di caratteri** da leggere
- Il metodo restituisce una stringa che contiene i caratteri
- Se l'argomento fornito vale **1**, il metodo `read()` ritorna una stringa che contiene **il prossimo carattere** del file

```
char = inputFile.read(1)
```

- Quando si raggiunge la **fine** del file, ritorna una stringa vuota ""

Algoritmo: leggere caratteri

```
char = inputFile.read(1)
while char != "" :
    Elabora il carattere
    char = inputFile.read(1)
```


Leggere dati composti (record)

- Un file di testo può contenere una sequenza di **dati composti** (**record** di dati), in cui ciascun record contiene diversi **campi** (**field**)
 - Solitamente, un record per linea
- Esempio: un file contiene dati degli studenti, i cui record sono composti da un numero di matricola, dal nome, dall'indirizzo e dall'anno di immatricolazione
- Lavorando con file che contengono record dati, solitamente si legge l'intero record prima di elaborarlo

```
For each record in the file:  
  Read the entire record  
  Process the record
```

Un record per linea, campi delimitati

- Il formato più comune memorizza un record dati in ogni linea del file
 - Esempio: nazione e popolazione
- I campi del record sono spesso separati da un delimitatore specifico, come “:” (o un altro carattere)
- Si possono estrarre facilmente i singoli campi usando il metodo `split()`
 - China:1330044605
 - India:1147995898
 - United States:303824646
 - . . .

Un record per linea, campi non delimitati

- Cosa succede se non abbiamo un delimitatore di campo?

China 1330044605

India 1147995898

United States 303824646

. . .

- Alcuni nomi di nazioni sono composti da **più di una parola**, non possiamo utilizzare lo **spazio** come delimitatore (i nomi con più parole sarebbero divisi male)
- Possiamo usare `rsplit()` in questo caso?

```
inputString = "United States 303824646"  
result = inputString.rsplit(" ", 1)  
print(result)
```

Un record per linea, campi non delimitati

- Un altro approccio potrebbe essere di leggere l'intera riga con `readline()` e poi ricercare la prima cifra numerica

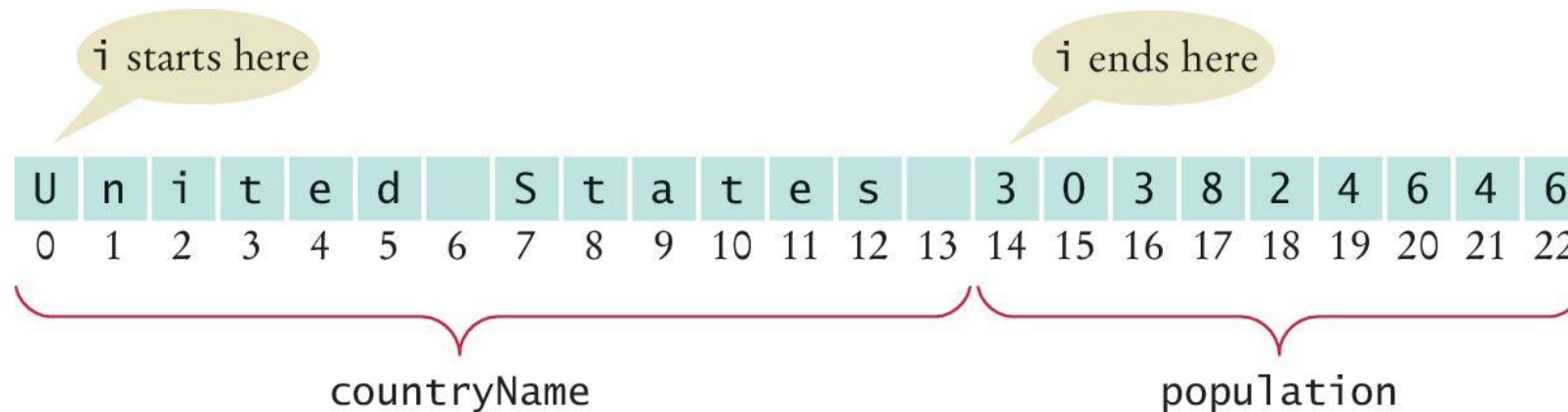
```
i = 0
while not line[i].isdigit() :
    i = i + 1
# 'i' punta al primo carattere numerico
```

- A questo punto si può estrarre la nazione e la popolazione estraendo delle porzioni (slice)

```
countryName = line[0 : i - 1]
population = int(line[i : ])
```

Un record per linea, campi non delimitati

- Trovare la prima cifra numerica e dividere la stringa



Nota

- I formati di record non delimitati possono portare ad ambiguità di lettura, oltre che difficoltà di gestione (ogni formato è un caso particolare)
- Fortunatamente questi formati non sono quasi mai utilizzati

Un record su più linee, un campo per linea

- I record possono essere organizzati in diversi modi, alcuni più facili di altri da gestire
- Un possibile formato prevede di memorizzare ciascun campo su una linea separata del file. In questo modo tutti i campi di un determinato record saranno su linee consecutive:

China

1330044605

India

1147995898

United States

303824646

. . .

Un record su più linee, un campo per linea

- Leggere i dati in questo formato è semplice
- Poiché ciascun record contiene due campi, leggiamo due linee dal file ad ogni nuovo record

```
line = infile.readline()
while line != "" :
    countryName = line.rstrip()
    line = infile.readline()
    population = int(line)
    Process data record
    line = infile.readline()
```


Leggere l'intero file – come stringa «gigante»

- Il metodo `read()` senza alcun parametro leggerà il file intero come un'unica stringa, la cui lunghezza è pari alla dimensione del file

```
contents = infile.read()  
# the whole file is read in a single (huge!) string
```

- Se vogliamo dividere la stringa gigante nelle singole righe, useremo `split`

```
lines = contents.split('\n')
```

- Oppure `splitlines`

```
lines = contents.splitlines()
```

Leggere l'intero file – come lista di stringhe

- Il metodo `readlines()` legge l'intero file come una lista di stringhe (una stringa per ogni linea del file)

```
lines = infile.readlines()
```

```
# scorciatoia (non molto leggibile...)  
lines = list(infile)
```

```
# equivalente a:  
lines = []  
for line in infile:  
    lines.append(line)
```

Operazioni sui file

Operazione

Spiegazione

`f = open(nomefile, modalità)`

Apre il file specificato dalla stringa *nomefile*. Il parametro *modalità* indica se il file va aperto in lettura ("r") o in scrittura ("w"). Viene restituito un oggetto di tipo file.

`f.close()`

Chiude un file aperto in precedenza. Una volta chiuso, il file non può essere utilizzato, finché non viene riaperto.

`string = f.readline()`

Legge la successiva riga di testo dal file e la restituisce sotto forma di stringa. Se è stata raggiunta la fine del file, viene restituita una stringa vuota, "".

`string = f.read(num)`

`string = f.read()`

Legge i successivi *num* caratteri dal file e li restituisce sotto forma di stringa, restituendo, invece, una stringa vuota se è stata raggiunta la fine del file. Se non viene fornito alcun argomento, viene letto l'intero contenuto del file, restituito come singola stringa.

`f.write(stringa)`

Scrive la *stringa* in un file aperto in scrittura..

Operazioni sui file

Method	What It Does	Method	What It Does
<code>.read(size=-1)</code>	This reads from the file based on the number of <code>size</code> bytes. If no argument is passed or <code>None</code> or <code>-1</code> is passed, then the entire file is read.	<code>.write(string)</code>	This writes the string to the file.
<code>.readline(size=-1)</code>	This reads at most <code>size</code> number of characters from the line. This continues to the end of the line and then wraps back around. If no argument is passed or <code>None</code> or <code>-1</code> is passed, then the entire line (or rest of the line) is read.	<code>.writelines(seq)</code>	This writes the sequence to the file. No line endings are appended to each sequence item. It's up to you to add the appropriate line ending(s).
<code>.readlines()</code>	This reads the remaining lines from the file object and returns them as a list.		

<https://realpython.com/read-write-files-python/>

Esempio

- Aprire il file `items.py`

Esercizio

- Leggere un file “estremi.dat” contenente coppie di numeri interi (x, y) , una coppia per riga e separate da uno spazio, e creare un secondo file “differenze.dat” che contenga le differenze $x - y$, una per riga.
- Esempio:

```
23 32
2 11
19 6
23 5
3 2
...
```

estremi.dat

```
-9
-9
13
18
1
...
```

differenze.dat

Soluzione (usando for)

```
infile = open('estremi.dat', 'r')
outfile = open('differenze.dat', 'w')

for line in infile:
    numeri = line.split()
    diff = int(numeri[0]) - int(numeri[1])
    outfile.write( f'{diff}\n' )

infile.close()
outfile.close()
```



Soluzione (usando readline)

```
infile = open('estremi.dat', 'r')
outfile = open('differenze.dat', 'w')

line = infile.readline()
while line!='':
    numeri = line.split()
    diff = int(numeri[0])-int(numeri[1])
    outfile.write( f'{diff}\n' )
    line = infile.readline()

infile.close()
outfile.close()
```



Un esempio di elaborazione di file di testo

PAGINA 440

Specifiche del problema

- Leggere due file che contengono dati sulle nazioni del mondo: `worldpop.txt` e `worldarea.txt`

```
Afghanistan 647500  
Akrotiri 123  
Albania 28748  
Algeria 2381740  
American Samoa 199  
• • •
```

`worldarea.txt`

Assumiamo che l'ordine delle nazioni sia lo stesso nei due file (alfabetico)

```
Afghanistan 32738376  
Akrotiri 15700  
Albania 3619778  
Algeria 33769669  
American Samoa 57496  
• • •
```

`worldpop.txt`

Specifiche del problema (2)

- Leggere due file che contengono dati sulle nazioni del mondo: `worldpop.txt` e `worldarea.txt`
- Creare un file `world_pop_density.txt` che contenga i nomi delle nazioni e la relativa densità di popolazione, dove i nomi delle nazioni siano allineati a sinistra e i numeri siano allineati a destra

```
Afghanistan      50.56
Akrotiri         127.64
Albania          125.91
Algeria          14.18
American Samoa  288.92
. . .
```

Sei passi per elaborare file di testo

- Capire il tipo di elaborazione necessaria
 - Elaborare i dati “al volo” (mentre li leggiamo)?
 - Memorizzare tutti i dati e poi elaborarli?
- Determinare quali file debbano essere scritti e letti
- Definire un metodo per ottenere i nomi dei file
- Scegliere se iterare direttamente sul file (`for ..in file`) o leggere le singole linee, parole o caratteri
 - Se i dati dei record sono su tutti su una linea, conviene iterare per linee
- Se l'input è organizzato per linee, estrarre i dati richiesti da ciascuna linea
 - Analizzare la stringa e gestire spazi vuoti, delimitatori, separatori, ...
- Usare funzioni per raggruppare ed isolare attività comuni

Sei passi per elaborare file di testo

- Capire il tipo di elaborazione necessaria
 - Elaborare i dati “al volo” (mentre li leggiamo)?
 - Memorizzare tutti i dati e poi elaborarli?
- Determinare quali file debbano essere scritti e letti
- Definire un metodo per ottenere i nomi dei file
- Scegliere se iterare direttamente sul file (`for ..in file`) o leggere le singole linee, parole o caratteri
 - Se i dati dei record sono su tutti su una linea, conviene iterare per linee
- Se l’input è organizzato per linee, e per linea
 - Analizzare la stringa e gestire spazi vuoti
- Usare funzioni per raggruppare ed

Ricordarsi, che in generale, la risposta giusta a queste domande è:

«dipende»

La difficoltà è capire da cosa dipende e che scelte prendere nel caso specifico

Passo 1: Capire il problema

- Finché ci sono linee da leggere
 - Leggi una linea da ciascuno dei due file
 - Estrai il nome della nazione
 - *population* = numero che segue la nazione, nel primo file
 - *area* = numero che segue la nazione, nel secondo file
 - If *area* != 0
 - $density = population / area$
 - Stampa nome nazione e *density*

```
Afghanistan      50.56
Akrotiri         127.64
Albania          125.91
Algria           14.18
American Samoa  288.92
. . .
```

Passo 2: Definire i file

- Definire quali file devo leggere o scrivere
- Ci sono due file di input:
 - worldpop.txt
 - worldarea.txt
- C'è un file di output:
 - world_pop_density.txt

Passo 3: Ottenere i nomi dei file

- Scegliere un meccanismo per acquisire i nomi dei file
- Ci sono generalmente tre opzioni:
 - Definire il nome del file come costante («hardcoded»)
 - Chiedere all'utente
 - Usare un *argomento sulla linea di comando* (argomento avanzato, non verrà trattato)
- In questo esempio useremo nomi *hardcoded*
 - Definiamo 3 costanti corrispondenti ai nomi dei 3 file

Passo 4: Iterare sul file o leggere le linee?

- Scegliere tra l'iterazione sull'oggetto file oppure la lettura di ciascuna linea separatamente
- In generale, se i dati di un record sono sulla stessa linea, conviene iterare sul file (`for line in infile:`)
- Se i dati sono su più linee, leggiamo una linea per volta (`infile.readline()`)
- In questo esempio conviene leggere una linea per volta, poiché i dati devono essere letti da due file diversi

Passo 5: Estrarre i dati

- Estrarre i dati dalle stringhe lette dal file, separando i diversi campi esistenti
 - Usare `split`, `rsplit`, porzioni di stringhe, ecc., per isolare ed estrarre i dati

Passo 6: Dividere in funzioni

- Usare le funzioni per isolare passaggi standard
- Trovare i task ripetitivi e sviluppare delle funzioni per gestirli

Esempio

- Aprire il file `population.py`

Gestione delle Eccezioni



7.5

Gestione delle eccezioni

- Vi sono due aspetti da gestire a proposito degli errori del programma a run-time:
 - Rilevare gli errori
 - Gestire gli errori
- La funzione **open** può **rilevare** il tentativo di leggere da un file non esistente
 - La funzione open **non** è in grado di **gestire** l'errore
 - Ci sono diverse azioni possibili, la funzione **non può sapere** quale sia la scelta corretta in questo programma
 - La funzione allora **riferisce l'errore ad un'altra parte del programma** che lo sappia gestire
- **La gestione delle eccezioni** fornisce un meccanismo flessibile per passare il controllo **dall'errore** ad un **gestore** che lo sappia trattare

Gestione delle eccezioni: quadro generale

RILEVARE ERRORI (RAISE)

- Il programma deve **controllare** se sono verificate tutte le condizioni per permettere la normale prosecuzione del lavoro
- Altrimenti il programma **solleva un'eccezione**
- Istruzione: **raise**
- Vi sono diversi **tipi** di eccezioni (ValueError, IOError, ...) in funzione della causa
- All'eccezione può essere assegnato un **messaggio** descrittivo del problema

GESTIRE ERRORI (TRY...EXCEPT)

- Se usiamo delle funzioni che possono generare eccezioni, dobbiamo definire del codice per gestirle
- Il codice «controllato» viene inserito in un blocco **try**
- Il codice «gestore» è inserito in un blocco **except**
- Se l'eccezione non viene gestita, ciò causerà **l'interruzione** del programma

Gestire le eccezioni

- Tutte le eccezioni dovrebbero essere gestita da qualche parte del programma
- Si tratta di un problema molto complesso
 - Occorre gestire ogni possibile eccezione e reagire a ciascuna nel modo corretto
 - Non è detto che tutti gli errori siano recuperabili
- Per gestire gli errori non recuperabili:
 - Per semplicità: terminare il programma
 - Per maggior usabilità: chiedere all'utente di correggere l'errore

Gestire le eccezioni: `try-except`

- Le eccezioni vengono gestite attraverso l'istruzione `try/except`
- Posizionare l'istruzione nella parte di programma che sa come gestire una particolare eccezione
- Il blocco `try` contiene una o più istruzioni che **potrebbero causare** un'eccezione (del tipo che stiamo cercando di gestire)
- Una o più clausole `except` contengono il **gestore** per ciascun tipo di eccezione

Sintassi: try-except

Sintassi

```
try :  
    enunciati  
    . . .  
except TipoDiEccezione1 :  
    enunciati  
    . . .  
except TipoDiEccezione2 as nomeVariabile :  
    enunciati  
    . . .
```

Esempio

Quando viene sollevata un'eccezione di tipo IOError, l'esecuzione riprende da qui.

Qui possono comparire ulteriori clausole except; le eccezioni più specifiche vanno elencate prima di quelle più generiche.

```
try :  
    infile = open("input.txt", "r")  
  
    line = infile.readline()  
    process(line)  
  
except IOError :  
    print("Could not open input file.")  
  
except Exception as exceptObj :  
    print("Error:", str(exceptObj))
```

Questa funzione può sollevare un'eccezione di tipo IOError.

Questo è l'oggetto di tipo eccezione che è stato sollevato.

try-except: Esempio

```
try :  
    filename = input("Enter filename: ")  
    infile = open(filename, "r")  
    line = infile.readline()  
    value = int(line)  
    . . .  
except IOError :  
    print("Error: file not found.")  
except ValueError as exception :  
    print("Error:", str(exception))
```

`open()` può generare una
eccezione `IOError`

`int()` può generare una
eccezione `ValueError`

L'esecuzione arriva qui se il file non
si può aprire

L'esecuzione arriva qui se la
stringa non si può convertire in
`int`

***Se una di queste due eccezioni viene
sollevata, le restanti istruzioni nel
blocco try vengono saltate***

Esempio

- Se viene sollevata un'eccezione `IOError`, si eseguirà la clausola `except` relativa all'eccezione `IOError`
- Se viene generata un'eccezione `ValueError`, si eseguirà la clausola `except` relativa all'eccezione `IOError`
- Ogni altro tipo di eccezione non sarà gestito da nessuno dei due blocchi `except`

Messaggi in output (1)

- Per ottenere il messaggio contenuto nell'eccezione, dobbiamo accedere all'*oggetto eccezione* corrispondente
- L'oggetto corrispondente all'eccezione si può assegnare con la sintassi **as**:

```
except ValueError as exception :
```

- Quando si esegue il gestore di ValueError, la variabile **exception** è impostata all'oggetto eccezione che è stato creato dalla **raise**

Messaggi in output (2)

- Nel codice del gestore, possiamo estrarre il messaggio usando `str(exception)` (convertiamo l'eccezione in stringa)
- Quando si esegue il corpo del gestore di errori, si può stampare il messaggio che è fornito all'interno dell'eccezione:

```
except ValueError as exception :  
    print("Error:", str(exception))
```

- Per esempio, se la stringa passata alla funzione `int()` fosse "35x2", allora il messaggio compreso nell'eccezione sarebbe:
`invalid literal for int() with base 10: '35x2'`

Suggerimento

- Lanciare eccezioni il prima possibile
 - Quando una funzione rileva un problema che non sa risolvere, è meglio generare un'eccezione rispetto a cercare di «riparare» con una correzione imperfetta
- Catturare le eccezioni il più tardi possibile
 - Al contrario, una funzione dovrebbe catturare le eccezioni solo se effettivamente è in grado di rimediare al problema
 - Altrimenti, il rimedio più semplice è lasciare che l'eccezione si «propaghi» alla funzione chiamante, finché non verrà (sperabilmente) catturata da un gestore competente

La clausola `finally`

- La clausola `finally` si usa quando occorre compiere alcune azioni «conclusive», indipendentemente dal fatto che si siano verificate eccezioni oppure no
- Ecco una tipica situazione:
 - È importante ricordare di **chiudere sempre un file di output** anche nel caso si verificano eccezioni (per garantire che tutto l'output sia scritto sul file)
 - Possiamo inserire la chiamata a `close()` in una clausola `finally` :

```
outfile = open(filename, "w")
try :
    writeData(outfile)
finally :
    outfile.close()
```


Sintassi: la clausola `finally`

Sintassi

```
try :  
    enunciati  
    . . .  
finally :  
    enunciati
```

Esempio

Questa sezione di codice
può sollevare eccezioni.

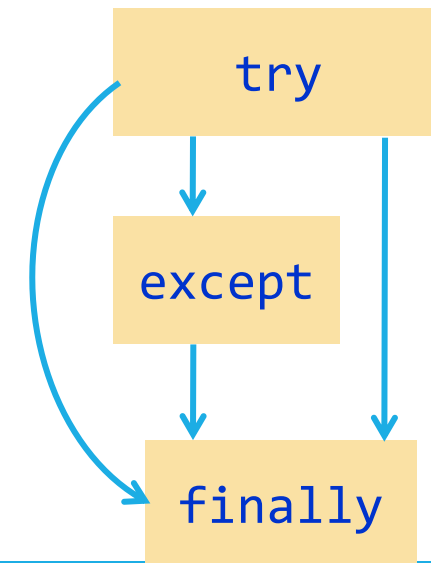
Questo codice viene sempre
eseguito, anche se nel blocco
`try` è stata sollevata
un'eccezione.

```
outfile = open(filename, "w")  
try :  
    writeData(outfile)  
    . . .  
finally :  
    outfile.close()  
    . . .
```

Il file deve essere aperto al di fuori del blocco `try`, altrimenti, se questo viene interrotto per qualche motivo, la clausola `finally` potrebbe tentare di chiudere un file che non è stato aperto.

💡 Suggerimento

- Non usare `except` e `finally` nello stesso blocco `try`
 - La clausola `finally` viene eseguita quando il blocco `try` termina, in tre modi possibili:
 - 1. Dopo il completamento dell'ultima istruzione del blocco `try`
 - 2. Dopo il completamento dell'ultima istruzione di una clausola `except`, se il blocco `try` ha incontrato un'eccezione
 - 3. Nel caso in cui si sia generata un'eccezione nel blocco `try`, ma questa non sia stata gestita da alcun blocco `except`

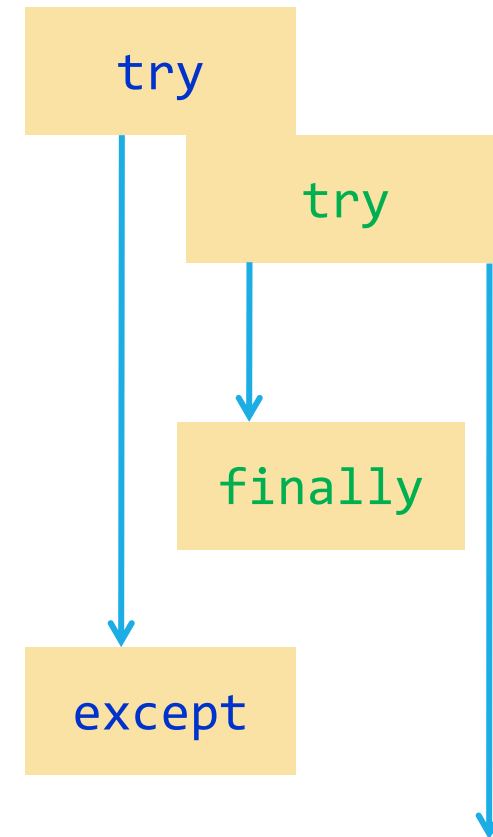


💡 Suggerimento (2)

Argomento
avanzato

- È meglio usare due blocchi try (annidati) per gestire il controllo del flusso

```
try :  
    outfile = open(filename, "w")  
    try :  
        # Write output to outfile  
    finally :  
        out.close() # Close resources  
except IOError :  
    # Handle exception
```



L'istruzione `with`

- Visto che il blocco `try/finally` è così comune per l'apertura e chiusura del file, Python offre una scorciatoia speciale:

```
with open(filename, "w") as outfile :  
    Write output to outfile
```

- Questa istruzione `with` apre il file con il nome specificato, assegna l'oggetto file alla variabile `outfile`, e chiude l'oggetto file quando si raggiunge la fine del blocco o quando si genera un'eccezione

Rilevare gli errori

- Cosa fare se qualcuno cerca di prelevare troppi soldi da un conto corrente?
- Si può “sollevare” un’eccezione
- Quando si solleva un’eccezione, l’esecuzione non continua con le istruzioni successive
 - Si trasferisce al gestore dell’eccezione

Usiamo l’istruzione `raise` per segnalare un’eccezione

```
if amount > balance :  
    raise ValueError("Amount exceeds balance")
```

Fonte dei messaggi di output

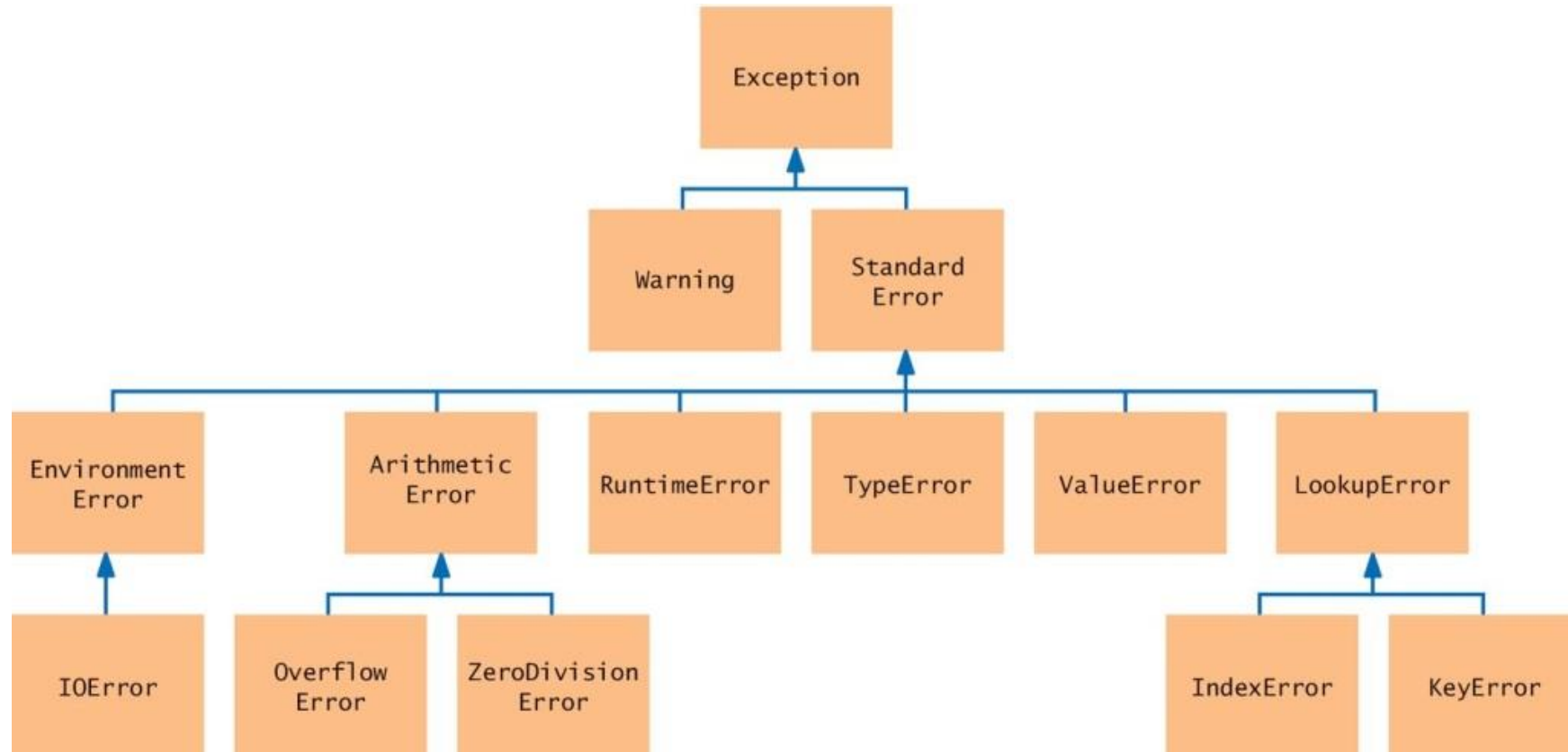
- Quando si solleva un'eccezione, si può fornire una stringa con un proprio messaggio. Ad esempio, chiamando:

```
raise ValueError("Amount exceeds balance")
```

- Il messaggio dell'eccezione "Amount exceeds balance", viene fornito all'atto della creazione dell'eccezione

Classi di eccezioni (una piccola parte)

- Ricercare l'eccezione più adeguata da generare



Sintassi: sollevare un'eccezione

Sintassi

```
raise oggettoEccezione
```

Esempio

Viene costruito un nuovo oggetto di tipo eccezione, per poi sollevarlo.

```
if amount > balance :  
    raise ValueError("Amount exceeds balance")  
balance = balance - amount
```

Questo messaggio fornisce informazioni dettagliate sull'eccezione.

Quando viene sollevata l'eccezione, questa riga non viene eseguita.

Gestire errori di acquisizione



7.6

Gestire errori di acquisizione

- Esempio: applicazione di lettura file
- Obiettivo: Leggere un file contenente i valori dei dati
 - La prima linea è il *conteggio* del numero dei dati presenti
 - Le linee restanti contengono i dati veri e propri
- Rischi:
 - Il file potrebbe **non esistere**
 - La funzione `open()` solleva un'eccezione se il file non esiste
 - Il file potrebbe avere dati nel **formato errato**
 - Se ci sono meno dati rispetto al previsto, o quando la prima riga non contiene il conteggio, il programma genererà un'eccezione di tipo `ValueError`
 - Infine, se ci sono più dati del previsto, dovrà essere sollevata un'eccezione di tipo `RuntimeError`

```
3
1.45
-2.1
0.05
```

Gestire errori di acquisizione: main()

- Scheletro della funzione, con la gestione di tutte le eccezioni

```
done = False
while not done :
    try:
        # Prompt user for file name
        data = readFile(filename) # May raise exceptions
        # Process data
        done = true;
    except IOError:
        print("File not found.")
    except ValueError :
        print("File contents invalid.")
    except RuntimeError as error:
        print("Error:", str(error))
```

Gestire errori di acquisizione: readFile()

- Crea l'oggetto file e chiama la funzione readData()
- Nessuna gestione delle eccezioni (non vi sono clausole except)
- La clausola `finally` chiude il file in tutti i casi (eccezione o no)

```
def readFile(filename) :  
    inFile = open(filename, "r") # May throw exceptions  
    try  
        return readData(inFile)  
    finally  
        inFile.close()
```

Gestire errori di acquisizione: readData()

- No gestione delle eccezioni (non vi sono clausole except o try)
- Se si verifica un'eccezione ValueError, esce
- Può generare un'eccezione RuntimeError

```
def readData(inFile) :  
    line = inFile.readline()  
    numberOfValues = int(line) # May raise a ValueError exception.  
    data = []  
    for i in range(numberOfValues) :  
        line = inFile.readline()  
        value = float(line) # May raise a ValueError exception.  
        data.append(value)  
    # Make sure there are no more values in the file.  
    line = inFile.readline()  
    # Extra data in file  
    if line != "" :  
        raise RuntimeError("End of file expected.")  
    return data
```

Uno scenario possibile

- `main` chiama `readFile`
 - `readFile` chiama `readData`
 - `readData` chiama `int`
 - L'input non contiene un numero intero, `int` solleva un'eccezione `ValueError`
 - `readData` non ha clausole `except`: **termina** immediatamente
 - `readFile` non ha clausole `except`: **termina** immediatamente dopo l'esecuzione della clausola `finally` e la **chiusura** del file
- La clausola `except IOError` viene **saltata**
- La clausola `except ValueError` viene **eseguita**

Example Code

- Aprire il file analyzedata.py

Sommario

Sommario: Input/Output su file

- Quando si apre un file, si specifica il nome del file che è memorizzato su disco, e la modalità con cui aprirlo
- Ricordare di chiudere tutti i file al termine della loro elaborazione
- Usare il metodo `readline()` per ottenere linee di testo da un file
- Scrivere su file usando il metodo `write()`

Sommario: Elaborare file di testo

- Si può iterare su un oggetto di tipo file per leggere le linee di testo del file
- Usare il metodo `rstrip()` per rimuovere il carattere di 'a capo' da una linea di testo
- Usare il metodo `split()` per dividere una stringa nelle parole componenti
- Leggere uno o più caratteri usando il metodo `read()`

Sommario: Eccezioni (1)

- Per segnalare una condizione eccezionale, usare l'istruzione `raise` per 'sollevare' (generare) un oggetto di tipo eccezione
- Quando si solleva un'eccezione, l'elaborazione prosegue nel gestore dell'eccezione
- Inserire le istruzioni che possono causare eccezioni in un blocco `try`, ed il gestore in una clausola `except`
- Quando si entra in un blocco `try`, è garantito che le istruzioni della clausola `finally` verranno eseguite, che si verifichino eccezioni o no

Sommario: Eccezioni (2)

- Sollevare un'eccezione appena viene rivelato un problema
 - Gestirla solo quando il problema può essere risolto
- Nel progettare un programma, chiedersi quali tipi di eccezioni potranno verificarsi
- Per ciascuna eccezione, decidere quale parte del programma potrà avere la competenza per poterla gestire