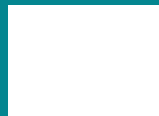




**University
of Ferrara**

Python

Eccezioni, files



Errori ed eccezioni

- Gli errori di sintassi vengono mostrati dal compilatore.
- Gli errori a tempo di esecuzione sono chiamati **Exceptions**, eccezioni.
- Le eccezioni non gestite generano un messaggio di errore durante l'esecuzione del programma e la sua conseguente terminazione.
- Ci sono molti tipi di eccezioni, vedi:

<https://docs.python.org/3/library/exceptions.html#builtin-exceptions>

per una lista completa di eccezioni:

- IndexError (indice fuori limite)
- TypeError (operazione applicata ad un oggetto del tipo sbagliato)
- ...

Gestione delle Eccezioni

- Abbiamo visto funzioni che possono trasformare un tipo in un altro tipo. Cosa succede se questa conversione non può essere fatta?
- La funzione solleverà un'eccezione → **ValueError** quando la funzione riceve un argomento del tipo corretto ma con un valore inappropriate
- Ad esempio, in `int('a')`, la funzione `int()` riceve un valore di tipo corretto (**str**), ma contenente un valore non corretto.
- Possiamo usare l'istruzione **try** per gestire situazioni erranee come questa.

L'istruzione try

```
try:
    # istruzione che può sollevare
    # un'eccezione
    x = int('a')
except ValueError:
    print("Wrong number format")
    x = 0
x = x + 2
```

L'istruzione try

Le istruzioni contenute nel blocco try vengono eseguite.

```
try:
    # istruzione per sollevare
    # un'eccezione
    x = int('a')
except ValueError:
    print("Wrong number format")
    x = 0
x = x + 2
```

L'istruzione try

```
try:
    # istruzione per
    # un'eccezione
    x = int('a')
except ValueError:
    print("Wrong number format")
    x = 0
x = x + 2
```

Se non ci sono errori,
l'esecuzione prosegue
dopo il blocco **except**
(operazione **x = x + 2**)

L'istruzione try

```
try:
    # istruzione per s
    # un'eccezione
    x = int('a')
except ValueError:
    print("Wrong number format")
    x = 0
x = x + 2
```

Se viene sollevata un'eccezione, l'esecuzione salta all'interno del corrispondente blocco **except** (se esiste) senza eseguire le altre istruzioni all'interno del blocco **try**.

L'istruzione try

```
try:
    # istruzione per
    # un'eccezione
    x = int('a')
except ValueError:
    print("Wrong number format")
    x = 0
x = x + 2
```

Dopo aver eseguito le istruzioni nel blocco **except**, l'esecuzione prosegue da dopo di esso.

L'istruzione try

- L'istruzione **try** può avere:

- **except** che cattura più di un'eccezione

```
except (TypeError, ValueError):
```

- Più di un **except**

```
except TypeError:
```

```
    pass
```

```
except ValueError:
```

```
    pass
```

- **except** può rinominare le eccezioni per poter lavorare su di esse

```
except TypeError as e:
```

L'istruzione try

- Le eccezioni che except può gestire sono quelle indicate nell'istruzione e nelle sue sottoclassi (come per Java).
 - L'ordine è importante!
- Se un'eccezione non può essere catturata, viene sollevata al di fuori del blocco try ed eventualmente può essere gestita da un blocco try esterno. In alternativa l'esecuzione verrà terminata restituendo l'eccezione non gestita.

L'istruzione try

- Dopo tutte le istruzioni **except** è possibile aggiungere un clausola **else** opzionale.
- E' utile nell'eventualità di codice che dev'essere eseguito se la clausola **try** non dovesse sollevare eccezioni.
- L'uso della clausola **else** è da preferire all'aggiunta di ulteriore codice alla clausola **try** perché evita di catturare accidentalmente un'eccezione che non è stata sollevata dal codice protetto dall'istruzione **try-except**.

L'istruzione try

```
try:
    # un'operazione che può generare un IOError
except IOError:
    # gestione dell'eccezione
else:
    # un'operazione che può generare un IOError
    # ma non vogliamo catturare l'eccezione
    # se venisse sollevata
finally:
    # qualcosa che vogliamo venga sempre eseguito
```

Try-except-else

```
try:
    f = open('f.txt', 'r')
except OSError:
    print('cannot open f.txt')
else:
    print('f.txt has', len(f.readlines()),
          'lines')
    f.close()
```

L'istruzione try

- **try** ha un'altra clausola opzionale: **finally**
- La clausola **finally** viene eseguita in entrambi i casi: sia che l'eccezione venga sollevata sia che non lo sia. Se nel blocco **try** o nell'**except** ci fosse un **return**, venisse generata una nuova eccezione, o altro, le istruzioni presenti nel blocco **finally** verrebbero eseguite prima del return
- Quando si verificasse un'eccezione nella clausola **try** ed essa non fosse gestita da una clausola **except** (o si fosse verificata in una clausola **except** o **else**), viene sollevata nuovamente dopo l'esecuzione della clausola **finally**.

Try-except-else-finally

```
try:
    result = x / y
except ZeroDivisionError:
    print("division by zero!")
else:
    print("result is", result)
finally:
    print("executing finally clause")
```

Indipendentemente dai valori assegnati a **x** e **y**, la stampa in **finally** verrà sempre eseguita.

Sollevare un'eccezione

- Con l'istruzione **raise** possiamo lanciare un'eccezione.
- Dopo raise, può essere indicata solo un'eccezione, semplicemente scrivendo il nome dell'eccezione verrà generata un'eccezione usando il costruttore senza argomenti.

```
raise TypeError    # equivale a sollevare TypeError()  
raise TypeError('message')
```


Sollevare un'eccezione

- La forma più semplice non è seguita da alcuna eccezione e viene utilizzata quando è stata sollevata un'eccezione e non intendiamo gestirla, ma semplicemente rilanciarla

```
try:  
    # qualcosa  
except TypeError:  
    print("TypeError occurred")  
    raise
```

- Se è presente un'istruzione finally, questa verrà eseguita prima di sollevare nuovamente l'eccezione

Definire una nuova eccezione

- Possiamo definire le nostre eccezioni.
- Esse devono derivare dalla classe **Exception** o da una sua sottoclasse.
- Solitamente, le eccezioni sono classi molto semplici, possono eventualmente avere alcuni argomenti per dare informazioni a proposito dell'errore stesso.

Definire una nuova eccezione

```
class MyError(Exception)
    pass
```

```
class MyMoreSpecificError(MyError)
    def __init__(self, why, mess):
        self.why = why
        self.mess = mess
```

```
class AnotherSpecificError(MyError)
    ...
```

L'istruzione **with**

- Python definisce un'istruzione molto utile che, in alcuni casi, semplifica l'uso di try-except-finally.
- Per esempio, si può scrivere

```
with expression:  
    # istruzione
```



```
expression  
# è possibile  
# un'inizializzazione  
try:  
    # istruzione  
finally:  
    # un'operazione  
    # di chiusura
```

L'istruzione **with**

- Quest'espressione può essere definita dal programmatore.
 - Le operazioni fatte durante l'inizializzazione e la chiusura devono essere definite seguendo specifiche convenzioni.
- La definizione è accoppiata a specifiche funzioni/operazioni.
 - L'esempio che segue mostra che per l'istruzione **expression** ci sono definizioni per le operazioni di **inizializzazione** e di **chiusura**.

```
with expression:  
    # istruzione
```

```
expression  
# è possibile una  
# inizializzazione  
try:  
    # istruzione  
finally:  
    # una qualche  
    # chiusura
```

L'istruzione **with**

- Nelle prossime slide vedremo dei possibili utilizzi, ad ogni modo, la definizione di questa istruzione non verrà descritta.
- Per chi fosse interessato può fare riferimento alla documentazione di Python:
 - https://docs.python.org/3/reference/compound_stmts.html#with
 - <https://docs.python.org/3/reference/datamodel.html#context-managers>
 - <https://www.python.org/dev/peps/pep-0343/>

File

- Per aprire un file si può usare la funzione `open()` che restituisce un oggetto di tipo **file**:

`open(filename, mode)`

- **mode** può essere:
 - **r** → lettura (questa è la modalità di default se l'argomento **mode** non è passato). **r+** apre in entrambi i modi lettura e scrittura. **rb** apre in lettura binaria. Se il file già esiste viene aperto con un puntatore all'inizio del file.
 - **w** → scrittura. **w+** apre in entrambi i modi lettura e scrittura. **wb** per file binari. Se il file già esiste, crea un file **nuovo vuoto**. Il puntatore viene posizionato all'inizio del file.
 - **a** → append. **a+** per entrambi i modi lettura e scrittura. **ab** per file binari. Il file non viene creato se già esiste. Il puntatore viene posizionato alla fine del file.

File

- Un oggetto file **f** ha 3 attributi:
 - **f.closed** → true se e solo se il file è chiuso
 - **f.mode** → modalità con cui il file è aperto
 - **f.name** → il nome del file
- Come sempre esiste un metodo per poter chiudere il file una volta che sono concluse le operazioni IO su di esso:

f.close()

Lettura di un file

- `f.read()` → legge l'intero file e restituisce il suo contenuto.
 - A questo metodo può essere passato un argomento per leggere un numero fissato di caratteri o byte a seconda del tipo di file che stiamo leggendo

```
f = open('text.txt')
```

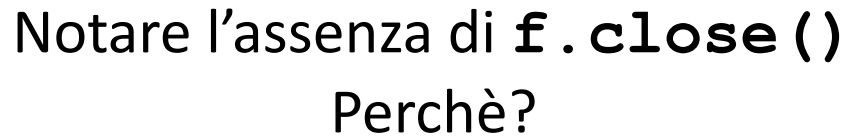
```
read_data = f.read()
```

```
f.close()
```

Lettura di un file

- `f.read()` → legge l'intero file e restituisce il suo contenuto.
 - In questo metodo può essere passato un argomento per leggere un numero fissato di caratteri o byte a seconda del tipo di file che stiamo leggendo

```
with open('workfile') as f:  
    read_data = f.read()
```



Notare l'assenza di `f.close()`
Perché?

L'istruzione **with**

- Python ha già definito il significato dell'istruzione **with** per la funzione `open()`

```
with open('workfile') as f:  
    read_data = f.read()
```



```
f = open('workfile')  
try:  
    read_data = f.read()  
finally:  
    f.close()
```

L'istruzione **with**

- Python ha già definito il significato dell'istruzione **with** per la funzione `open()`

```
with open('workfile') as f:  
    read_data = f.read()
```



```
f = open('workfile')  
try:  
    read_data = f.read()  
finally:  
    f.close()
```

Se abbiamo bisogno di una gestione più dettagliata di tutte le possibili eccezioni (file non trovato?), dobbiamo:

- usare direttamente try-except,
- circondare l'istruzione **with** con try-except, o
- ridefinire l'istruzione **with**.

Lettura di un file

- **f.readline()** → legge una singola riga (file di testo). Alla fine della stringa che viene restituita, viene inserito un carattere di nuova linea.
 - **f.readline()** restituisce una stringa vuota "" quando è stata raggiunta la fine del file, mentre una riga vuota è rappresentata da '\n', una stringa contenente un solo ritorno a capo.

```
line=f.readline()  
while line != '':  
    print(line, end='')  
    line=f.readline()  
f.close()
```

Una stringa viene aggiunta
dopo l'ultimo carattere, di
default un carattere di a capo.

Lettura di un file

- Lettura di un file ancora più semplice (e completa)

```
with open('workfile') as f:  
    for line in f:  
        print(line)
```

- Per raccogliere tutte le righe di un file:
 - `list(f)` → inizializza la lista inserendo le line del file
 - `f.readlines()` → come `list(f)`

Scrittura di un file

- Scrivere un file usando:

`f.write("Questa è una stringa")`

- Scritture multiple vengono concatenate.
- Per poter scrivere su un file è necessario aprirlo in modalità append o write.
- Il mode append aggiunge le stringhe alla fine del file.

Spostarsi in un file

- **`f.tell()`** → restituisce un numero intero che fornisce la posizione corrente del puntatore nel file rappresentata come numero di byte dall'inizio del file in modalità binaria e numero di caratteri in modalità testo.
- **`f.seek(offset, from_what)`** → sposta il puntatore di offset posizioni a partire da `from_what`. L'argomento `from_what` è facoltativo (il valore predefinito è 0) e impostato a 0 per l'inizio del file, 1 per la posizione corrente del puntatore e 2 per la fine del file.

Gestione dei file

- Python fornisce il modulo **os** che contiene metodi per la gestione del File System.
- **os.rename(filename, new_filename)** → rinomina un file.
- **os.remove(filename)** → rimuove un file
- **os.mkdir(dirname)** → crea una nuova cartella
- **os.chdir(dirname)** → per spostarsi nella cartella
- **os.getcwd()** → restituisce la cartella corrente di lavoro
os.rmdir(dirname) → rimuove la cartella passata. La cartella dev'essere vuota.

Salvataggio di oggetti su file

- Fino ad ora abbiamo visto file di testo. E se invece vogliamo salvare lo stato di un oggetto?
- In Java abbiamo la serializzazione, e vorremmo qualcosa di simile anche in Python.
- Per fare questo, Python fornisce un modulo integrato: **Pickle**.
- Questo consente facilmente di salvare e recuperare oggetti su/da file.

Pickle

- Per usare Pickle dobbiamo importarlo come `pickle`.
- Per salvare un oggetto usiamo `pickle.dump(object_name, file)`
- Per recuperare un oggetto usiamo:
`object_name = pickle.load(file)`

Esercizio 7

- Scrivere un programma che :
 - Definisca la classe **ToSave** contenente una stringa, un int, un double ed un booleano, il cui costruttore istanzia le variabili.
 - Prende una stringa, un int, un float ed un Boolean in ingresso (da tastiera).
 - Scriva su un file di testo i valori presi come input.
 - Legga dal file precedentemente creato questi valori e li salvi creando un istanza della classe **ToSave**
 - Si usi pickle per scrivere la classe sul file e la si recuperi salvandola su una nuova variabile **tsp**.
 - Si stampi l'istanza di ToSave così creata, con l'istruzione **print(tsp)**. Per poterla stampare ridefinire il metodo **__str__()** all'interno della classe che restituisce la sua rappresentazione come stringa.