Laboratorio di programmazione

Python

A.A. 2020-2021

Lezione 6

Input/Output da file

Fino ad ora abbiamo sempre impostato a mano le variabili da usare come **input** nei nostri codici (metodo *hardcoded*) oppure abbiamo chiesto, in modo interattivo, all'utente di fornire dei valori con la funzione *built-in* `input()`.

Abbiamo sempre visualizzato il nostro **output** a schermo, tramite la funzione `print()`.

In un caso realistico, però, molto probabilmente vorremo salvare i risultati per poterli rivedere/rianalizzare/rimaneggiare in seguito (magari con un nuovo programma!).

Inoltre, immaginate di voler girare diverse volte il vostro programma cambiando il valore di alcune variabili: aprire ogni volta il codice e cercare a mano i valori da cambiare è molto scomodo.

Per questo motivo, è molto pratico imparare a gestire l'Input/Output (I/O) del nostro proramma tramite la **lettura e scrittura di file**.

Lavorare con i file

Lavorare con i file è come lavorare con un quaderno:

1. Per poterci lavorare sopra devo aprirlo
2. Posso leggere il contenuto o scriverci sopra
    - posso leggerlo per intero e in ordine
    - posso "saltare" direttamente a quello che mi interessa
3. Finito di lavorare va richiuso

Aprire un file

Per prima cosa dobbiamo aprire il file, per poterci lavorare.

Per aprire un file, Python mette a disposizione la funzione *built-in* `open()`:

Aprire un file

Per prima cosa dobbiamo aprire il file, per poterci lavorare.

Per aprire un file, Python mette a disposizione la funzione *built-in* `open()`:

In [3]:
```
file_divina = open('divina_commedia.txt', 'w')  # Apro un file in scrittura
```

Aprire un file

Per prima cosa dobbiamo aprire il file, per poterci lavorare.

Per aprire un file, Python mette a disposizione la funzione *built-in* `open()`:

In [3]:
```
file_divina = open('divina_commedia.txt', 'w')  # Apro un file in scrittura
```

Questa funziona vuole due argomenti in ingresso:

- il nome del file (o il suo *file path*)
- una stringa che indica in che modalità abbiamo aperto il file (*default* `'r'`)

e ritorna un *file object* con una serie di metodi che ci permettono di maneggiarlo.

Tabella delle modalità

- `'r'`: modalità di lettura (*default*)
- `'w'`: modalità di scrittura
- `'x'`: modalità di creazione esclusiva: fallisce se il file esiste già
- `'a'`: modalità di scrittura: se il file esiste già, appende il contenuto alla fine
- `'b'`: modalità binaria
- `'t'`: modalità testo (*default*)
- `'+'`: consente l'update del file (lettura e scrittura)

Scrivere un file

Ora che abbiamo aperto il nostro file, proviamo a scriverci sopra, tramite il metodo `write()`

Scrivere un file

Ora che abbiamo aperto il nostro file, proviamo a scriverci sopra, tramite il metodo `write()`

In [4]:

```
divina = '''Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,
ché la diritta via era smarrita.

Ahi quanto a dir qual era è cosa dura,
esta selva selvaggia e aspra e forte,
che nel pensier rinova la paura!'''

for riga in divina.split('\n'):      # Ciclo sulle righe che compongono la stringa multilinea
    file_divina.write(f'{riga}\n')   # Scrivo la riga nel file (ricordandomi di andare a capo con \n
```

Una volta finito di scrivere, possiamo chiudere il file:

Una volta finito di scrivere, possiamo chiudere il file:

In [5]:
```
file_divina.close()   # Chiudo il file
```

Una volta finito di scrivere, possiamo chiudere il file:

In [5]:
```
file_divina.close()   # Chiudo il file
```

Andando ad aprire `divina_commedia.txt` con un editor di testo, vediamo il contenuto:

```
Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,
ché la diritta via era smarrita.

Ahi quanto a dir qual era è cosa dura,
esta selva selvaggia e aspra e forte,
che nel pensier rinova la paura!
```

Leggere un file

Ora che abbiamo scritto un file, proviamo ad aprirlo per leggerlo.

Proviamo ad usare il metodo `read()`:

Leggere un file

Ora che abbiamo scritto un file, proviamo ad aprirlo per leggerlo.

Proviamo ad usare il metodo `read()`:

In [6]:
```
file_divina = open('divina_commedia.txt', 'r')   # Apro un file in lettura
contenuto = file_divina.read()                   # Leggo l'intero contenuto del file
file_divina.close()                              # Chiudo il file
print(contenuto)
print(type(contenuto))
```

```
Nel mezzo del cammin di nostra vita
mi ritrovai per una selva oscura,
ché la diritta via era smarrita.

Ahi quanto a dir qual era è cosa dura,
esta selva selvaggia e aspra e forte,
che nel pensier rinova la paura!

<class 'str'>
```

Il metodo `read()` ci consente di leggere il contenuto del dile nella sua totalità.

In questo modo il contenuto del nostro file è una stringa multilinea che possiamo maneggiare con i metodi del suo tipo. Sarebbe comodo, però, poter maneggiare le singole righe che compongono il contenuto del file.

Possiamo ottenere la lista delle righe che compongono il file con il metodo: `readlines()`:

Il metodo `read()` ci consente di leggere il contenuto del dile nella sua totalità.

In questo modo il contenuto del nostro file è una stringa multilinea che possiamo maneggiare con i metodi del suo tipo. Sarebbe comodo, però, poter maneggiare le singole righe che compongono il contenuto del file.

Possiamo ottenere la lista delle righe che compongono il file con il metodo: `readlines()`:

In [7]:
```
file_divina = open('divina_commedia.txt', 'r')   # Apro un file in lettura
contenuto = file_divina.readlines()              # Leggo la lista di righe
file_divina.close()                              # Chudo il file
print(contenuto)
print(type(contenuto))
```

```
['Nel mezzo del cammin di nostra vita\n', 'mi ritrovai per una selva oscura,\n',
'ché la diritta via era smarrita.\n', '\n', 'Ahi quanto a dir qual era è cosa dur
a,\n', 'esta selva selvaggia e aspra e forte,\n', 'che nel pensier rinova la paur
a!\n']
<class 'list'>
```

O, se necessario, leggere e maneggiare una riga alla volta, con il metodo `readline()`:

O, se necessario, leggere e maneggiare una riga alla volta, con il metodo `readline()`:

In [8]:
```
file_divina = open('divina_commedia.txt', 'r')   # Apro un file in lettura
while True:                                      # Apro un ciclo "infinito"
    verso = file_divina.readline()               # Leggo una riga
    if len(verso) == 0:                          # Se trovo una riga vuota, esco dal ciclo
        break
    print(verso)
file_divina.close()                              # Chiudo il file
```

```
Nel mezzo del cammin di nostra vita

mi ritrovai per una selva oscura,

ché la diritta via era smarrita.


Ahi quanto a dir qual era è cosa dura,

esta selva selvaggia e aspra e forte,

che nel pensier rinova la paura!
```

Lavorare con file binari

Non tutti i tipi di file contengono del semplice testo. Possiamo trovarci a lavorare con **file binari** (immagini, video, zip, eseguibili, ...)

Python ci permette di lavorare con in file binari allo stesso modo con cui lavoriamo coi file di testo: dobbiamo solo ricordare che maneggiamo **byte** e non stringhe.

Proviamo a scrivere un file binario col metodo `write()`:

Lavorare con file binari

Non tutti i tipi di file contengono del semplice testo. Possiamo trovarci a lavorare con **file binari** (immagini, video, zip, eseguibili, ...)

Python ci permette di lavorare con in file binari allo stesso modo con cui lavoriamo coi file di testo: dobbiamo solo ricordare che maneggiamo **byte** e non stringhe.

Proviamo a scrivere un file binario col metodo `write()`:

In [14]:
```
soluzione = [3, 6, 2, 7, 1, 4, 0]
bin_file = open('regine.bin', 'wb')      # Apro un file binario in scrittura
byte_soluzione = bytearray(soluzione)    # Converto la soluzione in byte
bin_file.write(byte_soluzione)           # Scrivo la soluzione convertita in byte nel file
bin_file.close()                         # Chiudo il file
```

Lavorare con file binari

Non tutti i tipi di file contengono del semplice testo. Possiamo trovarci a lavorare con **file binari** (immagini, video, zip, eseguibili, ...)

Python ci permette di lavorare con in file binari allo stesso modo con cui lavoriamo coi file di testo: dobbiamo solo ricordare che maneggiamo **byte** e non stringhe.

Proviamo a scrivere un file binario col metodo `write()`:

In [14]:
```
soluzione = [3, 6, 2, 7, 1, 4, 0]
bin_file = open('regine.bin', 'wb')      # Apro un file binario in scrittura
byte_soluzione = bytearray(soluzione)    # Converto la soluzione in byte
bin_file.write(byte_soluzione)           # Scrivo la soluzione convertita in byte nel file
bin_file.close()                         # Chiudo il file
```

E a leggerne il contenuto col metodo `read()`:

Lavorare con file binari

Non tutti i tipi di file contengono del semplice testo. Possiamo trovarci a lavorare con **file binari** (immagini, video, zip, eseguibili, ...)

Python ci permette di lavorare con in file binari allo stesso modo con cui lavoriamo coi file di testo: dobbiamo solo ricordare che maneggiamo **byte** e non stringhe.

Proviamo a scrivere un file binario col metodo `write()`:

In [14]:
```
soluzione = [3, 6, 2, 7, 1, 4, 0]
bin_file = open('regine.bin', 'wb')      # Apro un file binario in scrittura
byte_soluzione = bytearray(soluzione)    # Converto la soluzione in byte
bin_file.write(byte_soluzione)           # Scrivo la soluzione convertita in byte nel file
bin_file.close()                         # Chiudo il file
```

E a leggerne il contenuto col metodo `read()`:

In [15]:
```
bin_file = open('regine.bin', 'rb')    # Apro un file binario in lettura
contenuto = bin_file.read()            # Leggo l'intero contenuto del file
bin_file.close()                       # Chiudo il file
print(contenuto)
print(type(contenuto))
```

```
b'\x03\x06\x02\x07\x01\x04\x00'
<class 'bytes'>
```

Esercizi

1. Leggete il file `divina_commedia_extended.txt` e scrivete un programma per contare:
    - il numero di versi nel file
    - il numero di parole nel file
    - il numero di caratteri diversi nel file
2. Leggete il file `rubrica.txt`, stampatene il contenuto e create un dizionario che abbia come chiavi i nomi delle colonne e come valori associati la lista dei valori della colonna

> ***suggerimento***
>
> *La tab in Python viene interpretata con il carattere* `\t`

Argomenti da linea di comando

Un programma serve per risolvere un problema.

Più il programma è scritto e progettato in modo da generalizzare il problema, più sarà in grado di risolvere problemi *dello stesso tipo* che hanno parametri diversi.

Finora, per cambiare i parametri del nostro programma generico (es. la dimensione della scacchiera nel problema delle regine, oppure il nome del file di output, ...) abbiamo modificato ogni volta a mano i valori delle variabili nel sorgente (*hard-coded*) o utilizzato la funzione *input* per passare dei valori in modo *interattivo*.

Un altro modo per *passare* valori specifici ad un programma generico, è specificare all'interprete all'avvio del programma una lista di parametri che il corpo centrale (*main*) dello *Script* attende ed è in grado di leggere.
In questo caso i parametri sono *passati* al programma come **argomenti da linea di comando** contestualmente alla chiamata del programma:

Argomenti da linea di comando

Un programma serve per risolvere un problema.

Più il programma è scritto e progettato in modo da generalizzare il problema, più sarà in grado di risolvere problemi *dello stesso tipo* che hanno parametri diversi.

Finora, per cambiare i parametri del nostro programma generico (es. la dimensione della scacchiera nel problema delle regine, oppure il nome del file di output, ...) abbiamo modificato ogni volta a mano i valori delle variabili nel sorgente (*hard-coded*) o utilizzato la funzione *input* per passare dei valori in modo *interattivo*.

Un altro modo per *passare* valori specifici ad un programma generico, è specificare all'interprete all'avvio del programma una lista di parametri che il corpo centrale (*main*) dello *Script* attende ed è in grado di leggere.
In questo caso i parametri sono *passati* al programma come **argomenti da linea di comando** contestualmente alla chiamata del programma:

```
$ python3 programma.py  argomento1  argomento2
```

Le quantità 'argomento1, argomento2, ecc..', assieme alla chiamata del programma, saranno salvate nella lista argv all'interno del namespace sys e potete visualizzarle ed utilizzarle con

```
import sys
..
print (sys.argv)

..
valore = sys.argv[1]
```

Le quantità 'argomento1, argomento2, ecc..', assieme alla chiamata del programma, saranno salvate nella lista argv all'interno del namespace sys e potete visualizzarle ed utilizzarle con

```
import sys
..
print (sys.argv)

..
valore = sys.argv[1]
```

Nel programma sarà possibile utilizzare i valori passati come argomenti, prendendoli dalla lista `argv` **nello stesso ordine con cui sono passati nella riga di comando**

> ***Attenzione**: il primo argomento non è in posizione `0` ma in posizone `1`: prima ci sarà il nome dello script Python e poi cominceranno i parametri*

Esercizi

Modificate il programma delle regine per utilizzare `sys.argv` per *passare* da linea di comando (nell'ordine)

- il numero di soluzioni da cercare
- la dimensione della scacchiera
- il nome del *file* dove salvare i risultati

Argparse

Passare al programma i valori specifici del problema *runtime* è molto comodo ma il metodo della lista `sys.argv` è molto fragile:

- cosa succede se l'ordine degli argomenti è sbagliato?
- come possiamo ricordare dopo mesi (o giorni) qual'era l'ordine corretto degli argomenti
- come ricordiamo che effetto ha ogni argomento nel programma?

Argparse

Passare al programma i valori specifici del problema *runtime* è molto comodo ma il metodo della lista `sys.argv` è molto fragile:

- cosa succede se l'ordine degli argomenti è sbagliato?
- come possiamo ricordare dopo mesi (o giorni) qual'era l'ordine corretto degli argomenti
- come ricordiamo che effetto ha ogni argomento nel programma?

Per semplificare il modo in cui si passano gli argomenti *runtime* ci sono diversi modi, uno di questi è la libreria **ArgumentParser** nel *namespace* `argparse`

```
import argparse

parser =  argparse.ArgumentParser()  # inizializza parser
```

Gli argomenti attesi con la loro descrizione vengono aggiunti come *parametri* da passare al programma

```
parser.add_argument('-n','--nome_esteso', help='Descrizione del parametro')
parser.add_argument('-b','--boolean_value', action='store_true', help="imposta il valore 'True' se trova il parametro")
parser.add_argument('-d','--con_default',  default='riferimento', help="Parametro che ha già un valore di default se non viene fornito l'argomento")

args = parser.parse_args() # fa il parsing degli argomenti da linea di comando
```

`ArgumentParser` produrrà inoltre un parametro `-h` che permette di visualizzare le descrizioni (*help*) dei parametri

Parametri da linea di comando

Il comando da shell:

```
python programma.py  -h
```

produrrà:

```
usage: programma.py [-h] [-n NOME_ESTESO] [-b] [-d CON_DEFAULT]

optional arguments:
  -h, --help            show this help message and exit
  -n NOME_ESTESO, --nome_esteso NOME_ESTESO
                        Descrizione del parametro
  -b, --boolean_value   imposta il valore 'True' se trova il parametro
  -d CON_DEFAULT, --con_default CON_DEFAULT
                        Parametro che ha già un valore di default se non viene
                        fornito l'argomento
```

mentre i valori, una volta fatto il parsing degli argomenti, saranno disponibili con il loro nome esteso come *attributi* dell'oggetto `args`

```
args = parser.parse_args() # fa il parsing degli argomenti da linea di comando
..
print(args.nome_esteso) # stampa a video il valore assegnato a nome_esteso
```

Esercizi

modificate il programma per la soluzione delle 8 regine per utilizzare argparse per

- specificare il numero di soluzioni da cercare (int)
- specificare la dimensione della scacchiera (int)
- specificare il nome del *file* dove salvare i risultati (string)
- selezionare se i risultadi devono essere unici o possono essere ripetuti (bool: True/False)

Considerazioni

ArgumentParser è molto comodo per gestire le *opzioni* di un programma e per preparare un *help* che permetta di ricordare **cosa fanno le opzioni di un programma**

L'*help* creato automaticamente, puo' anche essere *personalizzato* con delle opzioni in fase di creazione del *parser*

```
parser = argparse.ArgumentParser(prog="NomeProgramma.py",
                                 description="Cosa fa il programma quando utilizzato",
                                 epilog="Altre informazioni come Autore o data")

args = parser.parse_args()
```

In questo caso l'*help* :

```
python programma.py  -h
```

produrrà

```
usage: NomeProgramma.py [-h] [-n NOME_ESTESO] [-b] [-d CON_DEFAULT]

Cosa fa il programma quando utilizzato

optional arguments:
  -h, --help            show this help message and exit
  -n NOME_ESTESO, --nome_esteso NOME_ESTESO
                        Descrizione del parametro
  -b, --boolean_value   imposta il valore 'True' se trova il parametro
  -d CON_DEFAULT, --con_default CON_DEFAULT
                        Parametro che ha già un valore di default se non viene
                        fornito l'argomento

Altre informazioni come Autore o data
```