Python: Input/Output and Formatting

Rights & Credits

Il notebook è stato creato da Agostino Migliore; parte di esso è stata ispirata o basata su notebooks di:

- Simone Campagna (CINECA)
- Mirko Cestari (CINECA)
- Nicola Spallanzani (CNR-NANO)

Input/Output (I/O)

Blocco with

L'istruzione with consente di dotare l'esecuzione di un blocco di istruzioni con i metodi di un gestore di contesto (context manager). In altre parole, `with` consente di sincronizzare opportunamente un set di operazioni con l'apertura e chiusura di file, l'allocazione e il rilascio di risorse. In particolare, le operazioni di lettura e scrittura si effettuano all'interno di un blocco `with`, il quale assicura pure che i files siano chiusi alla fine delle operazioni che li riguardano. Possiamo comprendere meglio quanto sopra col semplice esempio seguente:

In [70]:
```
with open("a.txt", 'w') as f:
    f.write("Salve a tutti.\n")
    f.write("Questa è una lezione di Python,")
```

Tale blocco `with` definisce il seguente set di operazioni: con il file di nome `a.text` aperto (cioè, mentre il file `a.text` è aperto) in modalità di scrittura (`'w'` o, equivalentemente, si poteva scrivere `"w"`), avendogli attribuito lo pseudonimo (alias) `f`, effettuare su `f` la scrittura di .... Descriviamo dettagliatamente il blocco di istruzioni.

- `with` permette di mantenere il file aperto tanto quanto è necessario per le operazioni di scrittura e di richiuderlo alla fine.

- La funzione intrinseca open() apre il file e crea un corrispondente *oggetto file* (file object). Tale funzione ha la sintassi `open(nome_file, mode='r', ...)` dove il secondo parametro rappresenta la modalità di accesso al file. Il valore di default è il carattere `"r"`, che comanda di aprire il file in modalità di lettura.

- Scegliendo la modalità di apertura del file 'w', richiediamo di aprire il file nella modalità che consente la scrittura. Infatti, la lettera *w* deriva dall'inglese *writing*. In realtà, la

documentazione di Python per `w` recita: "open for writing, truncating the file first". Questo significa che il file viene troncato nel renderlo aperto in modalità scrittura. In generale, il *troncamento* consiste nel rimuovere contenuti da un file per ridurne le dimensioni. In particolare, per un file di testo, ciò comporterà la rimozione del contenuto attuale, quindi dando luogo alla sua riscrittura secondo le istruzioni successive nel blocco `with`.

- La parola chiave as consente di assegnare il nome (alias) `f` all'oggetto file, in modo tale da abbreviare la scrittura delle istruzioni successive. Abbiamo già usato tale parola chiave allo stesso modo nell'importare moduli;

- write() è il metodo di scrittura sul file. Esso riceve come argomento la stringa da scrivere.

I/O col blocco with

Creiamo un file `a.txt`, scrivendovi qualcosa, e poi lo riscriviamo usando le istruzioni di sopra. Per creare il file da qui, ci serviremo del comando echo, derivato da Unix/Linux.
La linea di comando

`!echo stringa_fornita_dall'utente`

stampa sullo standard output la stringa fornita dall'utente in tale linea di comando:

In [11]:
```
!echo corso di metodi di calcolo per la chimica
```

```
corso di metodi di calcolo per la chimica
```

Per immettere tale contenuto nel file `a.txt`, invece che scriverlo nello standard output attualmente in uso (lo jupyter notebook), dobbiamo ridirigere la scrittura verso il file. Ciò viene fatto con `>>`. Infatti, eseguendo

In [13]:
```
 !echo corso di metodi di calcolo per la chimica >> a.txt
```

vediamo che abbiamo creato un file col nome `a.txt` e col contenuto di cui sopra nel direttorio corrente, cioè quello che contiene questo notebook:

In [15]:
```
!dir *txt
```

```
 Il volume nell'unità C è OS
 Numero di serie del volume: 26B8-B399

 Directory di C:\Users\Agostino\Desktop\New_Life\insegnamento\insegnamento_2023_2024
\metodi_di_calcolo_2023_2024\Agostino_Migliore_MCC_lectures_2024

18/03/2024  14:44                44 a.txt
               1 File             44 byte
               0 Directory  113,271,209,984 byte disponibili
```

In [17]:
```
!more a.txt
```

```
corso di metodi di calcolo per la chimica
```

Sopra abbiamo usato un altro comando ereditato da Unix/Linux, more, per riportare sullo standard output il contenuto del file.
A questo punto, usiamo il blocco `with` di sopra,

In [71]:
```
with open("a.txt", 'w') as f:
    f.write("Salve a tutti.\n")
    f.write("Questa è una lezione di Python,")
```

e vediamo come è cambiato `a.txt` :

In [20]:
```
!more a.txt
```

```
Salve a tutti.
Questa è una lezione di Python.
```

Il file è stato riscritto perché esisteva già. Se non fosse già esistito, sarebbe stato creato all'atto dell'apertura. A questo punto, se vogliamo riaprire il file senza cancellarne il contenuto, per aggiungere altro testo, possiamo procedere come segue:

In [72]:
```
with open("a.txt", 'a') as f:
    f.write(" che fa parte \ndel corso di metodi di calcolo per la chimica.")
```

Sopra abbiamo usato la modalità di apertura `'a'`, che sta per `append` (aggiungi).
Vediamo che adesso il file ha il seguente contenuto:

In [118…
```
!more a.txt
```

```
Salve a tutti.
Questa è una lezione di Python, che fa parte
del corso di metodi di calcolo per la chimica.
```

Se vogliamo leggere il contenuto del file e farlo stampare usando una sintassi tipica del linugaggio Python, possiamo, per esempio, procedere come segue

In [74]:
```
with open("a.txt", 'r') as f:
    x = f.read()
    print(x)
```

```
Salve a tutti.
Questa è una lezione di Python, che fa parte
del corso di metodi di calcolo per la chimica.
```

oppure

In [75]:
```
with open("a.txt", 'r') as f:
    for line in f:
        print(line)
```

```
Salve a tutti.

Questa è una lezione di Python, che fa parte

del corso di metodi di calcolo per la chimica.
```

o ancora

In [78]:
```
with open("a.txt", 'r') as f:
    for line in f:
        print(line,end='')
```

```
Salve a tutti.
Questa è una lezione di Python, che fa parte
del corso di metodi di calcolo per la chimica.
```

Altri modi:

In [53]:
```
with open("a.txt", 'r') as f:
    for line in f:
        print(line.strip())
```

```
Salve a tutti.
Questa è una lezione di Python, che fa parte
del corso di metodi di calcolo per la chimica.
```

In [54]:
```
with open("a.txt", 'r') as f:
    x = f.read().splitlines()
    for n in x:
        print(n)
```

```
Salve a tutti.
Questa è una lezione di Python, che fa parte
del corso di metodi di calcolo per la chimica.
```

In [62]:
```
with open("a.txt", 'r') as f:
    x = f.read().split()
    print(x)
```

```
['Salve', 'a', 'tutti.', 'Questa', 'è', 'una', 'lezione', 'di', 'Python,', 'che', 'f
a', 'parte', 'del', 'corso', 'di', 'metodi', 'di', 'calcolo', 'per', 'la', 'chimic
a.']
```

In [79]:
```
with open("a.txt", 'r') as f:
    x = f.read().split()
    for n in x:
        print(n)
```

```
Salve
a
tutti.
Questa
è
una
lezione
di
Python,
che
fa
parte
del
corso
di
metodi
di
calcolo
per
la
chimica.
```

Il metodo readlines() consente di leggere tutte le righe assieme:

In [80]:
```
with open("a.txt", 'r') as f:
    x = f.readlines()
    print(x)
```

```
['Salve a tutti.\n', 'Questa è una lezione di Python, che fa parte \n', 'del corso d
i metodi di calcolo per la chimica.']
```

Possiamo anche procedere come segue a partire dal codice di sopra:

In [58]:
```
with open("a.txt", 'r') as f:
    x = f.readlines()
    for n in x:
        print(n,end='')
```

```
Salve a tutti.
Questa è una lezione di Python, che fa parte
del corso di metodi di calcolo per la chimica.
```

Esercizio

A partire dal programma `trova_primi_lc.py` creato precedentemente per individuare i numeri primi in una lista di numeri opportunamente definita, costruire un nuovo programma `trova_primi_f.py` che legga i numeri da vagliare da un file di input `numeri.txt`. Si ricordi che, se leggiamo per esempio il numero 5 in un file di testo, esso in realtà è la stringa "5" e bisogna usare `int()` per convertirlo in un numero intero.

Formatting

Nozioni preliminari su str() e repr() (facoltativo)

Prima di tutto, discutiamo due funzioni fondamentali nella creazione di stringhe. Esse sono `str()` e `repr()`.
La funzione str() converte un oggetto in una stringa, ovvero restituisce una versione in forma di stringa dell'oggetto fornito come suo argomento. Per esempio,

In [14]:
```
str(3.0), str('studio'), str()
```

Out[14]:
```
('3.0', 'studio', '')
```

Quando un oggetto Python viene stampato, in realtà esso viene previamente trasformato in una stringa. Infatti, dato `x`, `print(str(x))` e `print(x)` danno lo stesso risultato:

In [15]:
```
print(4, str(4))
```

```
4 4
```

L'azione di `str()` su un oggetto si può anche vedere nell'ottica dell'applicazione di un metodo, `__str__()`, associato con l'oggetto:

In [23]:
```
3.0.__str__()
```

Out[23]:
```
'3.0'
```

La funzione repr() fornisce una stringa che contiene una rappresentazione stampabile di un oggetto. Tutti gli oggetti hanno un metodo `__repr__()`, anche qualora non abbiano un metodo `__str__()`.
Confrontiamo le azioni di `str()` e `repr()` su qualche oggetto per capire la loro differenza.

In [27]:
```
x = 3.0; y = 'studio'
```

In [28]:
```
str(x), str(y), repr(x), repr(y)
```

Out[28]:
```
('3.0', 'studio', '3.0', "'studio'")
```

Quindi,

In [31]:
```
print(str(x), str(y), repr(x), repr(y))
```

```
3.0 studio 3.0 'studio'
```

Vediamo che `repr()` fornisce una rappresentazione più "intrinseca" degli oggetti rispetto a `str()`. Dal momento che l'oggetto di nome `y` è definito come una stringa, `str(y)` non ha bisogno di convertire `y` in una stringa, così restituendo lo stesso oggetto. Infatti,

In [33]:
```
str(y) is y
```

Out[33]: True

e `print` stampa il contenuto della stringa, cioè i suoi caratteri. Si noti, tuttavia, che l'oggetto stringa (completo) è fatto dai caratteri e dalle virgolette che li racchiudono. `repr()` fornisce una rappresentazione sotto forma di stringa (quindi aggiungendo virgolette) dell'oggetto stringa `y` nella sua interezza, quindi comprendendo le virgolette che gli conferiscono la struttura di stringa: `repr` : `y = 'studio'` → `"'studio'"`. Infatti, stampando `repr(y)` con `print`, ci ritroviamo il reale oggetto `y`, con la sua natura di stringa.

Si noti che la funzione `str()` è costruita in modo tale che, quando un oggetto non ha un metodo `__str__()`, essa invoca, in realtà, `repr()` e ne restituisce il risultato. Questo è il caso di `3.0`. Come potete vedere eseguendo il comando `help(3.0)`, nella lista di metodi associati con il float `3.0` c'è `__repr__()` mentre non c'è `__str__()`. Lo stesso vale per i contenitori intrinseci lista, tupla, dizionario e set.
Comprendiamo adesso che, quando scriviamo un'espressione in una cella dello jupyter notebook e la eseguiamo, il risultato viene stampato usando `repr()` e non `str()`. Per esempio,

In [54]:
```
y = "studio"
y
```

Out[54]: 'studio'

e quindi `y` equivale a `print(repr(y))`.

Si noti, infine, quanto segue.

In [60]:
```
print(repr(2+4), repr(6), repr('2'+'4'), repr("6"))
```

```
6 6 '24' '6'
```

Dal momento che la natura degli oggetti è preservata da `repr()`, `repr('2'+'4')` fornisce la rappresentazione sotto forma di stringa della concatenazione delle stringhe `2` e `4`, quindi `print` stampa la stringa `24` risultante dalla concatenazione. Invece, la corretta natura del risultato viene persa stampando `str('2'+'4') == str('24')`, in quanto

In [62]:
```
print(str('2'+'4'))
```

```
24
```

fornisce l'intero `24`. Così, è utile usare `repr()` invece di `str()` in contesti in cui è bene evitare ambiguità che possono scaturire dalla rappresentazione degli oggetti.

Formattazione di stringhe (spesso usate in I/O)

Formattazione vecchio stile

Supponiamo di volere stampare una frase che descrive la somma di due numeri interi, presentando i due numeri forniti in input e il risultato della loro somma. Il vecchio stile di formattazione ereditato dal linguaggio C procede come segue:

In [65]:
```
a, b = 5, 10
print("Il risultato di %d + %d è %d." % (a, b, a+b))
```

```
Il risultato di 5 + 10 è 15.
```

Nella frase formattata di sopra, %d è un segnaposto (campo) per numeri interi. I campi contrassegnati da `%d` all'interno della stringa indicano le posizioni in cui inserire i valori numerici raggruppati nella tupla dopo `%`.

La corrispondenza dei valori ai vari segnaposti e rispettivi specificatori di formato può anche essere effettuata per nome, il che significa che il set di valori dovrà essere fornito nella forma di un dizionario:

In [19]:
```
dct = {'a': 15, 'b': 5, 'c': 10}
print("Il risultato di %(a)d - %(b)d è %(c)d." % dct)
```

```
Il risultato di 15 - 5 è 10.
```

Consideriamo un altro esempio:

In [11]:
```
a, b = 0.1, 2
print("Il risultato di %.3f + %03d è %+0.3f" % (a, b, a + b))
```

```
Il risultato di 0.100 + 002 è +2.100
```

Nel segnaposto `%.3f`, che si può equivalentemente scrivere come `%0.3f`, la `f` indica di scrivere un float e `.3` di usare tre cifre decimali. `%03d` indica di scrivere l'intero con almeno 3 cifre. Dal momento che `b` consta di una sola cifra, sono messi due zeri prima. Se `b` contenesse tre o più cifre, il numero verrebbe scritto per come è, senza premettere alcuno zero. In `%+0.3f` il segno `+` forza ad aggiungere tale segno nello scrivere il numero positivo che rappresenta il risultato dell'operazione di somma.
Un altro segnaposto con uno specificatore di formato per stringhe è `%s`. Esso può essere usato per qualsiasi valore (oggetto) in ingresso, dal momento che usa `str()` per convertirlo comunque in stringa. Così, per esempio,

In [15]:
```
a, b = 10, "ciao"
"Hai detto %s volte %s!" % (a, b)
```

Out[15]:
```
'Hai detto 10 volte ciao!'
```

e quindi

In [16]:
```
print("Hai detto %s volte %s!" % (a, b))
```

```
Hai detto 10 volte ciao!
```

Con `%r` viene usata la funzione `repr()` invece di `str()`, per cui compare il virgolettato:

In [18]:
```
a, b = 10, "ciao"
print("Hai detto %s volte %r!" % (a, b))
```

```
Hai detto 10 volte 'ciao'!
```

Formattazione nuovo stile

Python fornisce il metodo intrinseco format() per la formattazione di stringhe. La stringa da rappresentare può contenere testo e campi (segnaposti) per la sostituzione di valori delimitati da parentesi graffe. `format()` consente di effettuare complicate sostituzioni e formattazioni di variabili.
Si consideri il semplice esempio seguente:

In [4]:
```
"Il mio nome è {0} {1}.".format("James", "Bond")
```

Out[4]:
```
'Il mio nome è James Bond.'
```

ovvero

In [5]:
```
print("Il mio nome è {0} {1}.".format("James", "Bond"))
```

```
Il mio nome è James Bond.
```

I numeri (*numeri di posizione*) `0` e `1` nei segnaposti sono le posizioni degli argomenti passati come input al metodo `format()`. Di conseguenza, nel creare la stringa, il campo `{0}` dentro la stringa va sostituito con `James` e il campo `{1}` con `Bond`.

In [3]:
```
print("Il mio nome è {1} {0}.".format("James", "Bond"))
```

```
Il mio nome è Bond James.
```

Se non avessimo fornito i numeri dentro le parentesi graffe, i campi sarebbero stati riempiti secondo l'ordine degli argomenti passati a `format()`:

In [6]:
```
print("Il mio nome è {} {}.".format("James", "Bond"))
```

```
Il mio nome è James Bond.
```

Si possono fornire gli argomenti a `format` anche *per nome* (ovvero *keyword*). In tal caso, le parentesi graffe conterranno le keywords. Nel caso di sopra possiamo, per esempio, procedere come segue:

In [7]:
```
"Il mio nome è {nome} {cognome}.".format(nome="James", cognome="Bond")
```

Out[7]:
```
'Il mio nome è James Bond.'
```

È pure possibile accedere agli elementi di una lista e inserirli in una stringa opportunamente formattata. Per esempio:

In [8]:
```
L = [1, 7, 4, 3, 9, 12, 19, 21, 34]
"Il risultato di {a[1]} + {a[3]} + {a[4]} è {a[6]}".format(a=L)
```

Out[8]:
```
'Il risultato di 7 + 3 + 9 è 19'
```

Si può procedere analogamente con un dizionario:

In [2]:
```
d = {'a': 0, 'b':1, 'c':2}
print("{D[c]} - {D[b]} = {D[b]}".format(D=d))
```

```
2 - 1 = 1
```

Gli specificatori di formato usati nel vecchio stile di formattazione si possono inserire anche nei campi definiti come nel nuovo stile. In tal caso, i numeri di posizione nei segnaposto sono seguiti da `:` e dallo specificatore di formato:

In [20]:
```
print("Il risultato di {1:.2f}/{0:.2f} è {2:.2f}.".format(0.25, 2.0, 8))
```

```
Il risultato di 2.00/0.25 è 8.00.
```

Tuttavia, se si voglino usare gli specificatori di formato `r` ed `s`, bisogna farli precedere dal punto esclamativo piuttosto che dai due punti:

In [28]:
```
print("{0!r} è un libro di {1!s}.".format("I Promessi Sposi", "Manzoni"))
```

```
'I Promessi Sposi' è un libro di Manzoni.
```

Interpolazione di stringhe: stringhe f (a partire da Python 3.6)

Il metodo delle stringhe f (in inglese, f-strings) condivide aspetti del metodo che usa `format` e l'assegnazione dei suoi argomenti per nome, ma l'assegnazione viene scissa dalla formattazione, che viene quindi semplificata. Confrontiamo i due modi.

Modo di sopra:

In [6]:
```
print("Il nome del ricercato è {cognome}, {nome} {cognome}!".format(nome="James", c
```

```
Il nome del ricercato è Bond, James Bond!
```

Modo alternativo recente:

In [7]:
```
nome = "James"
cognome = "Bond"
print(f"Il nome del ricercato è {cognome}, {nome} {cognome}!")
```

```
Il nome del ricercato è Bond, James Bond!
```

Bisogna pure considerare che, in generale, sono stati già assegnati valori alle variabili alla fine di qualche elaborazione e ora si vogliono semplicemente stampare in una frase leggibile, per cui il risultato di questo approccio è una semplificazione della parte di codice che detta la formattazione.
L'esempio di sopra è stato usato per illustrare il metodo di formattazione in questione, ma

era possibile scrivere direttamente la stringa dal momento che i valori sostituiti erano anch'essi delle stringhe. Tuttavia, come appena detto, bisogna immaginare un contesto in cui tali valori sono il risultato dell'applicazione di un programma. Per esempio, un codice viene usato per consultare un archivio e trovare nome e cognome di un ricercato, che vengono forniti all'utente tramite una frase, da formattare opportunamente, che appare sul terminale.
Facciamo un altro esempio utile per mostrare un altro aspetto utile della formattazione:

In [8]:
```
a = 1; b = 2
print(f"Il risultato di {a}/{b} è {a/b}.")
```

```
Il risultato di 1/2 è 0.5.
```

Come vedete, si possono anche eseguire operazioni tra i valori direttamente all'interno dei segnaposto. Si noti inoltre che, mentre il modo di procedere di sopra riflette il fatto che i valori stampati sono, in genere, l'esito di un'elaborazione precedente, tecnicamente funziona anche quanto segue:

In [9]:
```
print(f"Il risultato di {1}/{2} è {1/2}.")
```

```
Il risultato di 1/2 è 0.5.
```