lez_07

January 21, 2026

1 OS e File I/O

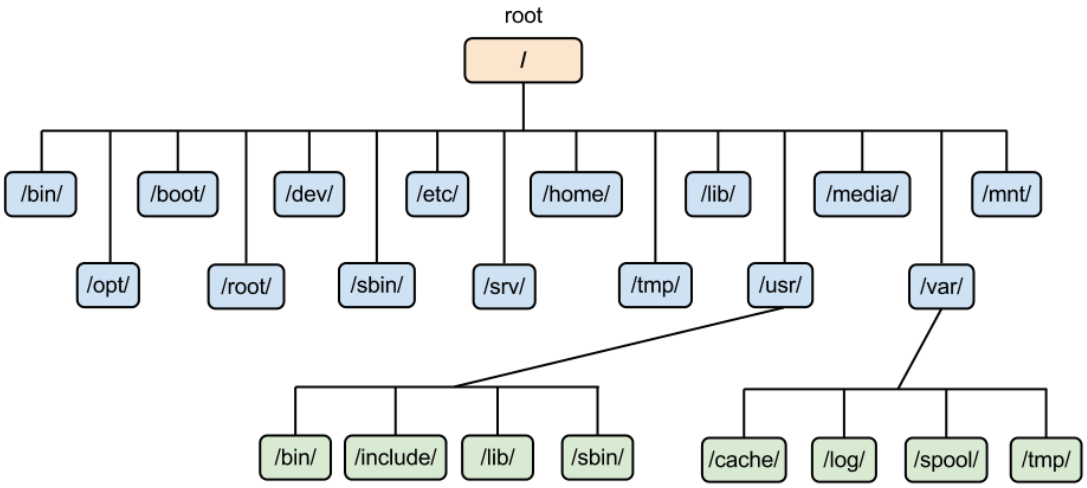
Nell'ambito dell'informatica, svolgono un ruolo fondamentale i file cosiddetti *Plain Text*. Si tratta di file di testo costituiti esclusivamente dai caratteri rappresentati con codifica ASCII, UTF-8, ecc. A differenza dei file testuali prodotti, ad esempio, con un *word processor* come *MS Word*, non contengono caratteri di formattazione. Questi file di testo costituiscono, per citare alcuni esempi, file di configurazione del sistema, file di log e, in ambito bioinformatico, file contenenti sequenze e/o annotazioni.

È chiaro dunque come sia utile accedere, leggere e modificare file testuali in maniera **programmatica**, utilizzando gli strumenti che ci vengono messi a disposizione dai linguaggi di programmazione. In questa lezione, vedremo come sia possibile svolgere queste operazioni in Python. I due elementi principali della lezione sono:

- **File I/O**: metodi e funzioni per l'accesso programmatico ai file testuali;
- **Il modulo OS**: una raccolta di metodi e funzioni per la gestione del **filesystem** del sistema operativo in uso.

1.1 Il filesystem nei sistemi UNIX

Prima di procedere, concediamoci un piccolo ripasso dell'organizzazione del **filesystem** nei sistemi *UNIX-like* e in particolare nei sistemi **GNU/Linux**.



Il filesystem Linux "*inizia*" dalla directory chiamata ***root***, indicata dal simbolo "/". La root è la cartella madre di tutte le altre directories. Tutti i file e tutte le cartelle del sistema sono contenute in root. Partendo dalla root, possiamo indicare la posizione delle altre directory utilizzando un ***path***, o *percorso*. Esistono due tipi di path:

- Il ***path assoluto*** inizia **SEMPRE** dalla root. Ad esempio, il path assoluto per la directory "*sbin/*", contenuta in "*usr/*", che a sua volta è contenuta in "/", può essere scritto: */usr/sbin/*. Il path assoluto ci consente di "localizzare" una directory a prescindere da dove ci troviamo all'interno del filesystem;
- Il ***path relativo*** descrive il percorso che bisogna seguire per arrivare al file o alla directory di interesse partendo dalla **posizione corrente**. Ai fini della nostra lezione e delle applicazioni che vedremo nel corso, la posizione corrente corrisponde alla directory nella quale si trova lo script in Python che vogliamo eseguire. Ricordiamo che la directory corrente si indica con il carattere "**.**", mentre per *salire di una posizione* nel filesystem, ossia raggiugnere la directory madre della directory corrente, si utilizza il carattere "**..**". **Esempio**: supponiamo che la posizione corrente sia */var/* e di voler raggiungere la posizione */usr/sbin/*. Il path relativo sarà dunque *./../usr/sbin/*. Traducendo: "Dalla directory corrente ("."), passa alla cartella madre ("..", ossia "/"), entra nella cartella *usr/* e poi nella cartella *sbin/*.

1.2 I/O per file di testo

Passiamo ora all'azione e vediamo in che modo Python ci consente di accedere, leggere e scrivere file di testo. Affinché un file di testo possa essere letto, creato o modificato è necessario che questo venga prima **aperto**. L'apertura dei file è svolta utilizzando la funzione ***open("path_del_file", "modalità")*** . Per iniziare, apriamo il file di testo denominato *text.txt*, situato nella cartella *data*:

[1]:
```
f = open('./data/text.txt', 'r')
print(f)
```

```
<_io.TextIOWrapper name='./data/text.txt' mode='r' encoding='UTF-8'>
```

Come possiamo vedere dall'output, la funzione *open()* crea un oggetto **TextIOWrapper** che contiene una sorta di collegamento al file. Questo oggetto sarà contenuto nella nostra variabile **f**. Il primo argomento è il *path* assoluto o relativo del file, mentre il secondo argomento è la modalità di apertura del file. Un file può essere aperto in 3 modalità diverse:

- **'r'** o *read*: il file viene aperto in modalità **lettura**. Un file aperto in questo modo può soltanto essere letto e non modificato. Costituisce la modalità più *sicura* e quella che viene utilizzata di default se nessun parametro viene specificato per la modalità. Se il file non esiste la funzione ritorna un errore;
- **'w'** o *write*: il file viene aperto in modalità **scrittura**. Se il file non esiste viene creato, se il file esiste invece, il suo contenuto viene **eliminato** e qualsiasi modifica successiva consiste sostanzialmente in una sovrascrittura. Dato che non è possibile recuperare successivamente il contenuto del file così sovrascritto è bene utilizzare l'apertura in scrittura con cautela;
- **'a'** o *append*: il file viene aperto in modalità lettura SENZA eliminare il contenuto del file. Le modifiche apportate al file verranno scritte al termine dell'informazione già presente. È quindi possibile *aggiungere* dell'informazione in fondo al file senza sovrascrivere nulla.

Una volta che abbiamo stabilito un collegamento al file, possiamo leggerlo riga per riga utilizzando il metodo ***read_line()***:

[2]:
```
line1 = f.readline()
line2 = f.readline()
print(f'######################Questa è la prima riga\n{line1}')
print(f'######################Questa è la seconda riga\n{line2}')
```

```
######################Questa è la prima riga
 Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus neque metus,
posuere non dolor sed, faucibus viverra tortor. Fusce vulputate gravida mauris
vel varius. Mauris tempor porttitor mattis. Aliquam facilisis scelerisque
varius. Duis maximus eros in mollis imperdiet. Donec erat massa, semper sed
facilisis sit amet, egestas a ipsum. Ut dolor urna, elementum non lacus eu,
congue accumsan sapien. Cras sed ante lacinia, euismod risus sed, ultricies
augue. Cras nibh purus, tempor in dignissim et, maximus at tellus. Aliquam risus
augue, luctus at dictum eget, imperdiet ac lorem.

######################Questa è la seconda riga
Proin egestas tellus sed ligula dapibus, id tempus augue scelerisque. Maecenas
scelerisque et arcu a accumsan. Proin sed consequat urna, sed rhoncus mauris.
Sed fringilla sem id elit egestas ornare. Maecenas condimentum, velit non
fermentum interdum, quam massa vestibulum quam, sed aliquam ante eros ac eros.
Mauris nec tincidunt lacus. Phasellus luctus libero eget posuere lobortis. Ut
vestibulum, metus sed porta tristique, quam est dapibus nisi, a dapibus libero
lectus eu mi. Proin consequat condimentum porta. Morbi et nisi ac nunc pharetra
condimentum. Aliquam euismod nisl id augue fringilla pulvinar.
```

Il metodo *readline()* legge una riga del file di testo ogni volta che viene chiamato. Dopo aver letto una riga la restituisce come stringa e ***sposta il puntatore*** alla riga successiva. Il processo si ripete finché non viene raggiunta la fine del file, denominata generalmente **EOF** (*End Of File*):

[4]:
```
for i in range(15):
    line_piece = f.readline()[0:3]
    print(line_piece)
    print(type(line_piece))
```

```
Ves
<class 'str'>
Viv
<class 'str'>
Dui
<class 'str'>

<class 'str'>

<class 'str'>
```

```
<class 'str'>

<class 'str'>

<class 'str'>

<class 'str'>

<class 'str'>

<class 'str'>

<class 'str'>

<class 'str'>

<class 'str'>

<class 'str'>
```

Notiamo come al termine del file, il metodo *readline()* non dia errore, ma si limiti a restituire delle stringhe vuote. Una volta terminate le operazioni sul file, questo deve essere chiuso. Per farlo utilizziamo il metodo ***close()***:

[39]:
```
f.close()
```

> **NOTA**: il linguaggio Python, come la maggior parte dei linguaggi interpretati, possiede un **garbage collector**, ossia una funzionalità che libera la memoria da variabili non utilizzate e, tra le altre cose, chiude i file non più necessari. La chiusura automatica dei file avviene generalmente una volta che lo script che li ha aperti termina l'esecuzione. Per script che vengono eseguiti in locale, che non aprono molti file e che terminano le operazioni rapidamente, dimenticarsi di chiudere un file non costituisce generalmente un problema. Pensiamo però alla situazione in cui uno script resta in esecuzione per lungo tempo, magari su di un server e magari lanciato da numerosi utenti contemporaneamente. Se ciascuno di questi utenti apre molti file e nello script non viene implementata la loro chiusura al termine delle operazioni, si capisce come sia semplice arrivare ad un sovraccarico della memoria di sistema, che comporta generalmente il crash dell'applicazione. È quindi importante che i file vengano SEMPRE chiusi una volta che non sono più necessari.

Un modo più utilizzato per *scorrere* un file aperto in lettura è quello di utilizzare un **ciclo *for***. Il *wrapper* dei file aperti in lettura si comporta infatti come un **iterabile** del quale ciascun elemento è rappresentato da una riga del file di testo. La sintassi generale è la seguente:

[6]:
```
f = open('./data/text.txt', 'r')
for line in f:
    print(len(line))
```

```
591
610
```

```
850
661
509
```

Per compattezza, è stata stampata la lunghezza di ogni riga invece della riga intera. Scorrere le righe di un file con un ciclo *for* ha un effetto simile a quello dell'utilizzo del metodo *readline()*. Se infatti proviamo a scorrere nuovamente il file, non otterremo nulla perché il **puntatore** è già stato spostato tra le righe successive del file fino ad arrivare alla fine:

[7]:
```
for line in f:
    print(line)
f.close()
```

Come volevasi dimostrare, non otteniamo nulla in output, poiché il file è già stato letto fino alla fine e il puntatore si trova ora alla fine del file. Per leggere nuovamente le righe, dovremmo chiudere il file e riaprirlo.

Vediamo ora un applicazione pratica di quanto appreso fin'ora, provando ad estrarre una sequenza amminoacidica da un file *FASTA*. La prima riga di un file FASTA costituisce **l'header**: questo presenta informazioni generali sulla sequenza contenuta nel file ed inizia sempre con il simbolo '>'. Dalla riga successiva a quella dell'header, troviamo invece la sequenza vera e propria, generalmente distribuita su più righe, ciascuna costituita da 60 nucleotidi/aminoacidi.

Quello che faremo è quindi aprire il file FASTA, ignorare la riga dell'header ricordandoci che inizia con il carattere "$>$" per poi inserire l'intera sequenza in una variabile di tipo stringa:

[9]:
```
#Apro il file FASTA

fasta_file = open('./data/lyz.fasta', 'r')

#Inizializzo una variabile "sequenza" come stringa vuota:
seq = ''

#Scorro le righe del file FASTA
for line in fasta_file:
    #Se la riga inizia con il simbolo '>', è l'header
    if line.startswith('>'):
        #stampo l'header
        print(f'Header found: {line}')
    else:
        #se la riga non inizia con '>' sarà parte della sequenza
        seq += line.strip()
print(seq)

#chiudo il file!
f.close()
```

```
Header found: >NP_000230.1 lysozyme C precursor [Homo sapiens]

MKALIVLGLVLLSVTVQGKVFERCELARTLKRLGMDGYRGISLANWMCLAKWESGYNTRATNYNAGDRSTDYGIFQINSR
```

```
YWCNDGKTPGAVNACHLSCSALLQDNIADAVACAKRVVRDPQGIRAWVAWRNRCQNRDVRQYVQGCGV
```

1.3 Il *Context Manager*

Ricordarsi ogni volta di chiudere i file aperti è un'operazione tediosa ed è facile dimenticarsene. Per questo motivo, non si utilizza quasi mai la sintassi che abbiamo utilizzato fin'ora per aprire un file. La gestione della chiusura del file viene affidata al **context manager**.

Come suggerisce il nome, il context manager è un costrutto sintattico che ci permette di utilizzare il file soltanto nel *contesto* in cui è necessario. Sarà poi il context manager ad occuparsi della sua chiusura quanto le operazioni sul file saranno concluse. Il context manager è riconoscibile dalle *keywords* ***with*** ed ***as*** e viene presentata di seguito:

[1]:
```
with open('./data/lyz.fasta', 'r') as f:
    print(f.readline())
    print(f.readline())
    print('ciao, il file è ancora aperto')
print('Il context manager ha chiuso il file')

try:
    print(f.readline())
except ValueError:
    print('Can\'t read line. The file has been closed')
```

```
>NP_000230.1 lysozyme C precursor [Homo sapiens]

MKALIVLGLVLLSVTVQGKVFERCELARTLKRLGMDGYRGISLANWMCLAKWESGYNTRATNYNAGDRST

ciao, il file è ancora aperto
Il context manager ha chiuso il file
Can't read line. The file has been closed
```

Come possiamo osservare dall'output, sebbene il file non sia stato chiuso "*manualmente*", il context manager ha provveduto alla sua chiusura al termine dell'esecuzione del blocco di codice.

1.4 Scrittura su file di testo

Come anticipato, le due modalità di apertura di un file in scrittura sono ***write*** (w) e ***append*** (a). La prima crea un nuovo file e lo apre in scrittura se questo non esiste. Se il file invece esiste già, cancella tutto il suo contenuto e ci permette di iniziare a riempirlo da capo. È quindi evidente come la modalità *write* vada usata con cautela.

La modalità *append*, invece, apre un file già esistente ed ci da la possibilità di iniziare a scrivere sulla prima riga dopo il contenuto già presente nel file.

Il metodo che ci permette di scrivere su un file di testo è ***write()*** e prende come argomento la stringa da inserire. Vediamo un semplice esempio del suo utilizo, nel quale leggiamo prima l'header da un file *fasta* già esistente per poi scriverlo su un nuovo file chiamato *lyz_header*.fasta. Successivamente riapriamo il nuovo file in append per aggiungere nuove righe successive all'header.

[3]:
```
#estraggo l'header dal file lyz.fasta
with open('./data/lyz.fasta', 'r') as f:
    header = f.readline()

#apro un nuovo file, lyz_header.fasta e inserisco al suo interno l'header +␣
 ↪un'altra stringa
filename = 'lyz_header.txt'
with open(f'./data/{filename}', 'w') as file:
    file.write(header)
    file.write('Ciao, ho scritto qui\n')

#riapro il file appena creato in append, e aggiungo due righe di testo
with open(f'./data/{filename}', 'a') as file:
    file.write('Ho scritto in append\n')
    file.write('Ciao, me ne vado \n')

#controllo il file
with open(f'./data/{filename}', 'r') as f:
    for line in f:
        print(line)
```

```
>NP_000230.1 lysozyme C precursor [Homo sapiens]

Ciao, ho scritto qui

Ho scritto in append

Ciao, me ne vado
```

1.5 Modulo OS

OS è un modulo python che ci consente, tra le altre cose, di interagire con il *filesystem* del nostro sistema operativo. Per brevità e praticità, in questa lezione affronteremo solo la creazione e la rimozione di directories. Per iniziare, importiamo il modulo OS e creiamo una cartella utilizzando la funzione *mkdir()*

[21]:
```
import os
os.mkdir('./cartella')
```

E verifichiamo che la cartella sia stata effettivamente creata utilizzando la funzione *listdir()*

[23]:
```
print(os.listdir('./data'))
```

```
['biostats.csv', 'MN450734.1.fasta', 'lyz.fasta', 'text.txt',
'MK404050.1.fasta', 'MK404051.1.fasta', 'lyz_header.txt', 'multi.fasta']
```

Come possiamo vedere dall'output, la funzione *listdir()* restituisce una **lista** contenente i nomi di tutti i file e tutte le cartelle presenti all'interno della nostra **directory corrente**. Ricordiamo che la directory corrente, di default, è la cartella contenente lo script Python che stiamo eseguendo.

Proviamo ora a creare di nuovo la directory *cartella* utilizzando nuovamente la funzione *mkdir()*

[24]:
```
os.mkdir('cartella')
```

```
---------------------------------------------------------------------------
FileExistsError                           Traceback (most recent call last)
/tmp/ipykernel_10588/398642766.py in <module>
----> 1 os.mkdir('cartella')

FileExistsError: [Errno 17] File exists: 'cartella'
```

Come possiamo vedere, cercare di creare una directory già esistente restituisce l'errore **FileExistsError**. In generale, quando vogliamo creare una nuova directory, dobbiamo prima assicurarsi che la cartella non esista già. Per farlo, possiamo utilizzare la funzione *os.path.isdir()*, che restituisce **True** se la cartella esiste, **False** se la cartella non esiste:

[26]:
```
if os.path.isdir('./cartella'):
    print('La cartella esiste già')
else:
    os.mkdir('./cartella')

try:
    os.mkdir('./cartella')
except FileExistsError:
    print('La cartella esiste')
```

```
La cartella esiste già
La cartella esiste
```

Per concludere, possiamo rimuovere una cartella **VUOTA** con la funzione *rmdir()*

[28]:
```
if os.path.isdir('./cartella'):
    os.rmdir('./cartella')
else:
    print('La cartella non esiste')
```

```
La cartella non esiste
```

2 Esercizi:

1) Leggere il file *text.txt* contenuto nella cartella *data/* e contare il numero di parole presenti nel file. Assumere come parole tutte le serie di caratteri separate da uno spazio;
2) Stabilire il contenuto percentuale di Alanina della sequenza contenuta nel file *data/lyz.fasta*;
3) Leggere il file *data/multi.fasta* e creare un dizionario così strutturato: {header1 : seq1, header2 : seq2, ..., headerN, seqN}
4) Utilizzando il dizionario così generato, creare per ciascuna sequenza un file *fasta*, con nome *header.fasta*. All'interno del file, il formato dovrà essere del tipo:

|> acc_number

ATCGATCGATCTACGCTACGCTA….

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

..

3 Soluzioni

3.0.1 Esercizio 1

[12]:
```
n_parole = 0
with open('./data/text.txt', 'r') as file:
    for line in file:
        n_parole += len(line.split(' '))
print(f'Il file contiene {n_parole} parole')
```

```
Il file contiene 477 parole
```

3.0.2 Esercizio 2

[32]:
```
with open('data/lyz.fasta', 'r') as f:
    seq = ''
    for line in f:
```

```
        if line.startswith('>'):
            pass
        else:
            seq += line.strip()
print(seq)
ala_percent = seq.count('A')
print(ala_percent)
```

```
MKALIVLGLVLLSVTVQGKVFERCELARTLKRLGMDGYRGISLANWMCLAKWESGYNTRATNYNAGDRSTDYGIFQINSR
YWCNDGKTPGAVNACHLSCSALLQDNIADAVACAKRVVRDPQGIRAWVAWRNRCQNRDVRQYVQGCGV
15
```

3.0.3 Esercizio 3

[35]:
```
seq_dict = {}
with open('data/multi.fasta', 'r') as f:
    for line in f:
        if line.startswith('>'):
            acc_no = line.split(' ')[0][1:]
            seq_dict[acc_no] = ''
        else:
            seq_dict[acc_no] += line.strip()
for accs in seq_dict:
    print(accs, seq_dict[accs], '\n')
```

```
MN450734.1 GCTATAGTTTCCATGGCATGTCGGCTACCGGGTAATGTGTCAACACCCGAAGAGCTTTGGAACCTATTG
CAAAGTGGTAAAGACACAACTACCGACGTTCCTAAAGATCGCTGGGATGCTGGGAAATTATACCATCCGGATCCTAGCGT
GGAAGGGAAGTCATACTGCAGCAGAGGGAGCTTCCTTGACTCTATCTACTCCTACGATGCGTCCTTCTTCGGCATTTCTC
CGCGTGAGGCACAAGCAATGGATCCCGCGCAGCACCTGATGCTGGAGTTGGTCTGGGAAGGATTTGAAAGAGCCGGCTAC
ACCAAGGACAAACTAAGCGGAAGTACTACGGGCGTTTTCGTTGGCGTTAGCAATAATGGAACTTCGACTGCTGTGCCACC
AGATCTCAAGGGTCACTCTATCACAGGGAGTGCCAGTGCGACAATATCTGGTCGCCTATCGTATACCTTCAATCTACAAG
GGCCATCCATGACAATTGATACAGCATGCTCGTCTTCCCTGGTGGCTACCCATTTGGCGTGTAATGCCCTGCGCCAGGGC
GAATGCAATATGGCGTTGGCCGGCGGCATCAGTCTTCTTCTCACTCCGGGAATACATATAGAGTTTAGCAGACTTCGCGG
TATTTCGCCTGATGGTCGGTGCAGAGCTTTCTCAGAGGACACAGAGGGAACAGGGTTTAGTGAGGGTGCCGCTATTGTAC
TTCTTAAACGTCTTTCGGATGCCGGGCCT

MK404051.1 GGGGATTTTCTTTTGGGTGCGAGTGCCGCAGGCATCCGGTGCCTCGTCTGAGCGCTCAGCCAGCCTTCT
GGCTTATCGCGAGGAGGGGCAATTTGGATGGTGGGGTTGTGCCGACTTTTTCGCGCCAGCGCTAGGCCCGATCTGGGTCT
CGCCAACAACAACACCACGACGCACATCCGCATTTCTCGCTTCATTGCGAGCGCGCTTTTCCACAATTTCATGATGCTAA
CACACTCCACAGGAAGCTGCCGAACTCGGCAAGGGTTCCTTCAAGTACGCATGGGTGCTCGACAAGTTGAAGGCCGAGCG
TGAGCGTGGTATCACCATTGACATTGCCCTGTGGAAGTTTGAAACTCCCAAGTACTACGTAACAGTTAGTAGGTGTTCAC
CTTTATTATGTTGTTACTTTCTCTGAGCGAACTCAACAACTGACAACTCGATTAGTCGACGCCCCCGGTCATCGTGATTT
CATCAAGAACATGATCACTGG

MK404050.1 TGTTCTGAGCTGCCTTCTGAGACGGCACTGACAGCGTGCAGGGTAACCAGATTGGTGCCGCCTTCTGGC
AGACCATCTCCGGCGAGCATGGCCTCGACGGCTCCGGTGTCTACAATGGCACCTCGGATCTCCAGCTCGAGCGCATGAAC
GTCTACTTCAACGAGGTGCGTCACATGACTCCATCGCTGCTTCAGGGCGCGCAGACTGACCGCAACAGGCGTCCGGCAAC
AAGTTCGTGCCCCGCGCCGTCCTCGTCGATCTCGAGCCCGGCACCATGGACGCTGTCCGCGCTGGACCTTTCGGACAGCT
```

```
CTTCCGCCCGGACAACTTCGTTTTCGGCCAA
```

3.0.4 Esercizio 4

[25]:
```
for accs in seq_dict:
    with open(f'./data/{accs}.fasta', 'w') as f:
        f.write(f'>{accs}\n{seq_dict[accs]}')
os.listdir('./data')
```

[25]:
```
['biostats.csv',
 'MN450734.1.fasta',
 'lyz.fasta',
 'text.txt',
 'MK404050.1.fasta',
 'MK404051.1.fasta',
 'multi.fasta']
```