



## **Corso di Apprendistato Python**

Mod. Python  
Docente:  
Tonino Petrulli



- Generatori, iteratori e chiusure;
- Lavorare con il file system, l'albero delle directory e i file;
- Moduli selezionati della libreria standard Python (*os*, *datetime*, *time* e *calendar*).

## Generators - where to find them

**Generatore:** a cosa associate questa parola? Forse si riferisce a qualche dispositivo elettronico. O forse si riferisce a una macchina pesante e seria progettata per produrre energia, elettrica o di altro tipo.

Un generatore Python è **un pezzo di codice specializzato in grado di produrre una serie di valori e di controllare il processo di iterazione**. Per questo motivo i generatori sono spesso chiamati **iteratori** e, sebbene alcuni possano trovare una distinzione molto sottile tra questi due elementi, noi li tratteremo come un tutt'uno.

Forse non ve ne rendete conto, ma avete già incontrato i generatori molte, molte volte. Date un'occhiata a questo semplicissimo snippet:

```
for i in range(5):  
    print(i)
```

La funzione `range()` è, infatti, un generatore, che è (di nuovo) un iteratore.

Qual è la differenza?

Una funzione restituisce un valore ben definito, che può essere il risultato di una valutazione più o meno complessa, ad esempio di un polinomio, e viene invocata una volta sola.

Un generatore **restituisce una serie di valori** e in genere viene invocato (implicitamente) più di una volta.

Nell'esempio, il generatore `range()` viene invocato sei volte, fornendo cinque valori successivi da zero a quattro e segnalando infine che la serie è completa.

Il processo sopra descritto è completamente trasparente. Facciamo un po' di chiarezza. Mostriamo il **protocollo dell'iteratore**.

## **Generatori - dove trovarli: continua**

Il **protocollo dell'iteratore** è un modo in cui un oggetto deve comportarsi per conformarsi alle regole imposte dal **contesto delle istruzioni** for e in. Un oggetto conforme al protocollo degli iteratori è chiamato **iteratore**.

Un iteratore deve fornire due metodi:

`__iter__()` che dovrebbe **restituire l'oggetto stesso** e che viene invocato una sola volta (è necessario affinché Python avvii con successo l'iterazione)

`__next__()`, che ha lo scopo di **restituire il valore successivo** (primo, secondo e così via) della serie desiderata; sarà invocato dalle istruzioni for/in per passare all'iterazione successiva; se non ci sono più valori da fornire, il metodo deve **sollevare l'eccezione** StopIteration.

Sembra strano? Niente affatto. Guardate l'esempio

class Fib:

```
def __init__(self, nn):
```

```
    print("__init__")
```

```
    self.__n = nn
```

```
    self.__i = 0
```

```
    self.__p1 = self.__p2 = 1
```

```
def __iter__(self):
```

```
    print("__iter__")
```

```
    return self
```

```
def __next__(self):
```

```
    print("__next__")
```

```
    self.__i += 1
```

```
    if self.__i > self.__n:
```

```
        raise StopIteration
```

```
    if self.__i in [1, 2]:
```

```
        return 1
```

```
    ret = self.__p1 + self.__p2
```

```
    self.__p1, self.__p2 = self.__p2, ret
```

```
    return ret
```

```
for i in Fib(10):
```

```
    print(i)
```

Abbiamo costruito una classe in grado di iterare i primi  $n$  valori (dove  $n$  è un parametro del costruttore) dei numeri di Fibonacci.

Ricordiamo che i numeri di Fibonacci ( $Fib_i$ ) sono definiti come segue:

$$Fib_1 = 1$$

$$Fib_2 = 1$$

$$Fib_i = Fib_{i-1} + Fib_{i-2}$$

In altre parole:

- i primi due numeri di Fibonacci sono uguali a 1;
- qualsiasi altro numero di Fibonacci è la somma dei due precedenti (ad esempio,  $Fib_3 = 2$ ,  $Fib_4 = 3$ ,  $Fib_5 = 5$ , e così via)

Immergiamoci nel codice:

- righe da 2 a 6: il costruttore della classe stampa un messaggio (lo useremo per tracciare il comportamento della classe), prepara alcune variabili (`__n` per memorizzare il limite della serie, `__i` per tracciare il numero di Fibonacci corrente da fornire e `__p1` insieme a `__p2` per salvare i due numeri precedenti);
- righe da 8 a 10: il metodo `__iter__` è obbligato a restituire l'oggetto iteratore stesso; il suo scopo può essere un po' ambiguo, ma non c'è alcun mistero; provate a immaginare un oggetto che non è un iteratore (per esempio, è una collezione di alcune entità), ma uno dei suoi componenti è un iteratore in grado di scansionare la collezione; il metodo `__iter__` dovrebbe **estrarre l'iteratore e affidargli l'esecuzione del protocollo di iterazione**; come potete vedere, il metodo inizia la sua azione stampando un messaggio;
- righe da 12 a 21: il metodo `__next__` è responsabile della creazione della sequenza; è un po' prolisso, ma questo dovrebbe renderlo più leggibile; prima stampa un messaggio, poi aggiorna il numero di valori desiderati e, se raggiunge la fine della sequenza, il metodo interrompe l'iterazione sollevando l'eccezione `StopIteration`; il resto del codice è semplice e riflette esattamente la definizione che abbiamo mostrato in precedenza;
- Le righe 24 e 25 utilizzano l'iteratore.

Il codice produce il seguente risultato:

\_\_init\_\_

\_\_iter\_\_

\_\_prossimo\_\_

1

\_\_prossimo\_\_

1

\_\_prossimo\_\_

2

\_\_prossimo\_\_

3

\_\_prossimo\_\_

5

\_\_prossimo\_\_

8

\_\_prossimo\_\_

13

\_\_prossimo\_\_

21

\_\_prossimo\_\_

34

\_\_prossimo\_\_

55

\_\_prossimo\_\_



Guarda:

- l'oggetto iteratore viene istanziato per primo;
- Successivamente, Python invoca il metodo `__iter__` per avere accesso all'iteratore vero e proprio;
- il metodo `__next__` viene invocato undici volte: le prime dieci volte producono valori utili, mentre l'undicesima termina l'iterazione.

### **Generatori - dove trovarli: continua**

L'esempio precedente mostra una soluzione in cui l'**oggetto iteratore fa parte di una classe più complessa**.

Il codice non è molto sofisticato, ma presenta il concetto in modo chiaro.

Date un'occhiata al codice

```
class Fib:
    def __init__(self, nn):
        self.__n = nn
        self.__i = 0
        self.__p1 = self.__p2 = 1
    def __iter__(self):
        print("Fib iter")
        return self
    def __next__(self):
        self.__i += 1
        if self.__i > self.__n:
            raise StopIteration
        if self.__i in [1, 2]:
            return 1
        ret = self.__p1 + self.__p2
        self.__p1, self.__p2 = self.__p2, ret
        return ret
```

```
class Class:
    def __init__(self, n):
        self.__iter = Fib(n)

    def __iter__(self):
        print("Class iter")
        return self.__iter;
```

```
object = Class(8)
```

```
for i in object:
    print(i)
```

Abbiamo inserito l'iteratore Fib in un'altra classe (possiamo dire che lo abbiamo composto nella classe Class). Viene istanziato insieme all'oggetto Class.

L'oggetto della classe può essere usato come iteratore quando (e solo quando) risponde positivamente all'invocazione `__iter__` - questa classe può farlo e, se viene invocata in questo modo, fornisce un oggetto in grado di obbedire al protocollo di iterazione.

Per questo motivo, l'output del codice è lo stesso di prima, anche se l'oggetto della classe Fib non viene utilizzato esplicitamente nel contesto del ciclo for.

## The yield statement

Il protocollo degli iteratori non è particolarmente difficile da capire e da usare, ma è anche indiscutibile che **sia piuttosto scomodo**.

Il principale disagio che comporta è **la necessità di salvare lo stato dell'iterazione tra le successive invocazioni di `__iter__`**.

Ad esempio, l'iteratore Fib è costretto a memorizzare con precisione il punto in cui è stata interrotta l'ultima invocazione (cioè il numero valutato e i valori dei due elementi precedenti). Questo rende il codice più grande e meno comprensibile.

Per questo motivo Python offre un modo molto più efficace, comodo ed elegante di scrivere iteratori.

Il concetto si basa fondamentalmente su un meccanismo molto specifico e potente fornito dalla parola chiave `yield`.

Si può pensare alla parola chiave `yield` come a un fratello più intelligente dell'istruzione `return`, con una differenza essenziale.

Date un'occhiata a questa funzione:

```
def fun(n):
```

```
    for i in range(n):
```

```
        return i
```

Sembra strano, non è vero? È chiaro che il ciclo for non ha la possibilità di terminare la sua prima esecuzione, poiché il ritorno lo interromperà irrevocabilmente.

Inoltre, richiamare la funzione non cambierà nulla: il ciclo for partirà da zero e verrà interrotto immediatamente.

Possiamo dire che una funzione di questo tipo non è in grado di salvare e ripristinare il proprio stato tra invocazioni successive.

Ciò significa anche che una funzione come questa **non può essere utilizzata come generatore**.

Abbiamo sostituito esattamente una parola nel codice: riuscite a vederla?

```
def fun(n):
```

```
    for i in range(n):
```

```
        yield i
```

Abbiamo aggiunto yield al posto di return. Questa piccola modifica **trasforma la funzione in un generatore** e l'esecuzione dell'istruzione yield ha effetti molto interessanti.

Innanzitutto, fornisce il valore dell'espressione specificata dopo la parola chiave yield, proprio come return, ma non perde lo stato della funzione.

Tutti i valori delle variabili sono congelati e attendono la prossima invocazione, quando l'esecuzione viene ripresa (non ripresa da zero, come dopo il ritorno).

C'è una limitazione importante: una **funzione** di questo tipo **non deve essere invocata esplicitamente**, perché di fatto non è più una funzione, ma **un oggetto generatore**.

L'invocazione **restituirà l'identificatore dell'oggetto**, non la serie che ci aspettiamo dal generatore.

Per le stesse ragioni, la funzione precedente (quella con l'istruzione return) può essere invocata solo in modo esplicito e non deve essere usata come generatore.

## **Come costruire un generatore**

Vi mostriamo il nuovo generatore in azione.

Ecco come possiamo usarlo:

```
def fun(n):  
    for i in range(n):  
        yield i
```

```
for v in fun(5):  
    print(v)
```

Riesci a indovinare l'uscita?

0

1

2

3

4



## Come costruire il proprio generatore

E se si avesse bisogno di un **generatore per produrre le prime  $n$  potenze di 2**?

Niente di più facile. Guardate il codice qui sotto:

```
def powers_of_2(n):  
    power = 1  
    for i in range(n):  
        yield power  
        power *= 2
```

```
for v in powers_of_2(8):  
    print(v)
```

Riuscite a indovinare l'output? Copiate il codice nell'editor ed eseguitelo per verificare le vostre ipotesi.

## Comprensione delle liste

I generatori possono essere utilizzati anche all'interno di **comprensioni di liste**, come in questo caso:

```
def powers_of_2(n):  
    power = 1  
    for i in range(n):  
        yield power  
        power *= 2  
t = [x for x in powers_of_2(5)]  
print(t)
```

Eseguire l'esempio e verificare l'output.

### La funzione list()

La funzione list() può trasformare una serie di invocazioni successive del generatore in **un vero e proprio elenco**:

```
def powers_of_2(n):  
    power = 1  
    for i in range(n):  
        yield power  
        power *= 2  
t = list(powers_of_2(3))  
print(t)
```

Di nuovo, provate a prevedere l'output ed eseguite il codice per verificare le vostre previsioni.

## **L'operatore in**

Inoltre, il contesto creato dall'operatore in consente di utilizzare anche un generatore.

L'esempio mostra come farlo:

```
def powers_of_2(n):  
    power = 1  
    for i in range(n):  
        yield power  
        power *= 2
```

```
for i in range(20):  
    if i in powers_of_2(4):  
        print(i)
```

Qual è l'output del codice? Eseguire il programma e verificare.

## Il generatore di numeri di Fibonacci

Vediamo ora un **generatore di numeri di Fibonacci** e assicuriamoci che sia molto migliore della versione oggettiva basata sull'implementazione del protocollo dell'iteratore diretto.

Eccola qui:

```
def fibonacci(n):  
    p = pp = 1  
    for i in range(n):  
        if i in [0, 1]:  
            yield 1  
        else:  
            n = p + pp  
            pp, p = p, n  
            yield n
```

```
fibs = list(fibonacci(10))  
print(fibs)
```

Indovinate l'output (un elenco) prodotto dal generatore ed eseguite il codice per verificare se avete ragione.

## Ulteriori informazioni sulle list comprehensions

Dovreste essere in grado di ricordare le regole che governano la creazione e l'uso di un fenomeno Python molto speciale, chiamato **comprensione delle liste: un modo semplice e di grande effetto per creare liste e il loro contenuto.**

```
list_1 = []  
for ex in range(6):  
    list_1.append(10 ** ex)  
list_2 = [10 ** ex for ex in range(6)]  
print(list_1)  
print(list_2)
```

All'interno del codice sono presenti due parti, che creano entrambe un elenco contenente alcune delle prime potenze naturali di dieci.

Il primo utilizza un modo routinario di utilizzare il ciclo for, mentre il secondo fa uso della comprensione delle liste e costruisce l'elenco in situ, senza bisogno di un ciclo o di altro codice esteso.

Sembra che l'elenco venga creato all'interno di se stesso - non è vero, naturalmente, perché Python deve eseguire quasi le stesse operazioni del primo frammento, ma è indiscutibile che il secondo formalismo è semplicemente più elegante e consente al lettore di evitare qualsiasi dettaglio non necessario.

L'esempio produce due righe identiche contenenti il seguente testo:

```
[1, 10, 100, 1000, 10000, 100000]  
[1, 10, 100, 1000, 10000, 100000]
```

## Ulteriori informazioni sulle list comprehensions

C'è una sintassi molto interessante che vogliamo mostrarvi ora. La sua utilizzabilità non è limitata alla comprensione di liste, ma dobbiamo ammettere che le comprensioni sono l'ambiente ideale per essa.

Si tratta di un'**espressione condizionale: un modo per selezionare uno dei due diversi valori in base al risultato di un'espressione booleana.**

`expression_one if condition else expression_two`

A prima vista può sembrare un po' sorprendente, ma bisogna tenere presente che **non si tratta di un'istruzione condizionale**. Inoltre, non è affatto un'istruzione. È un operatore.

Il valore fornito è uguale a `espressione_uno` quando la condizione è vera e a `espressione_due` in caso contrario.

Un buon esempio vi dirà di più. Guardate il codice

```
the_list = []
```

```
for x in range(10):
```

```
    the_list.append(1 if x % 2 == 0 else 0)
```

```
print(the_list)
```

Il codice riempie un elenco con 1 e 0: se l'indice di un particolare elemento è dispari, l'elemento viene impostato a 0, altrimenti a 1.

Semplice? Forse non a prima vista. Elegante? Indiscutibilmente.

È possibile utilizzare lo stesso trucco all'interno di una **list comprehension**? Sì, è possibile.

## List comprehensions vs. generators

Ora guardate il codice qui sotto e vedete se riuscite a trovare il dettaglio che trasforma la comprensione di un elenco in un generatore:

```
the_list = [1 if x % 2 == 0 else 0 for x in range(10)]  
the_generator = (1 if x % 2 == 0 else 0 for x in range(10))
```

```
for v in the_list:  
    print(v, end=" ")  
print()
```

```
for v in the_generator:  
    print(v, end=" ")  
print()
```

Sono le **parentesi**. Le parentesi fanno una comprensione, le parentesi fanno un generatore.

Il codice, tuttavia, quando viene eseguito, produce due righe identiche:

```
1 0 1 0 1 0 1 0 1 0  
1 0 1 0 1 0 1 0 1 0
```

Come si può sapere che il secondo assegnamento crea un generatore e non una lista?

Possiamo mostrarvi una prova. Applicate la funzione `len()` a entrambe le entità.

`len(the_list)` sarà valutato 10. Chiaro e prevedibile. `len(the_generator)` solleverà un'eccezione e verrà visualizzato il seguente messaggio:

```
TypeError: object of type 'generator' has no len()
```

Naturalmente, non è necessario salvare né l'elenco né il generatore: è possibile crearli esattamente nel punto in cui servono, proprio come qui:

```
for v in [1 if x % 2 == 0 else 0 for x in range(10)]:  
    print(v, end=" ")  
print()
```

```
for v in (1 if x % 2 == 0 else 0 for x in range(10)):  
    print(v, end=" ")  
print()
```

Nota: lo stesso aspetto dell'output non significa che i due cicli funzionino allo stesso modo. Nel primo ciclo, l'elenco viene creato (e iterato) come un insieme - esiste effettivamente quando il ciclo viene eseguito.

Nel secondo ciclo, non c'è alcun elenco: ci sono solo i valori successivi prodotti dal generatore, uno per uno.

Eseguite i vostri esperimenti.



## La funzione lambda

La funzione lambda è un concetto mutuato dalla matematica, più precisamente da una parte chiamata *calcolo lambda*, ma questi due fenomeni non sono la stessa cosa.

I matematici utilizzano *il calcolo lambda* in molti sistemi formali legati alla logica, alla ricorsione o alla dimostrabilità di teoremi. I programmatori usano la funzione lambda per semplificare il codice, per renderlo più chiaro e comprensibile.

Una funzione lambda è una funzione senza nome (si può anche chiamare **funzione anonima**). Naturalmente, una simile affermazione solleva immediatamente la domanda: come si fa a usare qualcosa che non può essere identificato?

Fortunatamente non è un problema, perché si può dare un nome a una funzione di questo tipo se è davvero necessario, ma, in effetti, in molti casi la funzione lambda può esistere e funzionare rimanendo completamente in incognito.

La dichiarazione della funzione lambda non assomiglia in alcun modo a una normale dichiarazione di funzione: vedetela voi stessi:

parametri lambda: espressione

Una clausola di questo tipo **restituisce il valore dell'espressione tenendo conto del valore attuale dell'argomento lambda corrente**.

Come al solito, un esempio sarà utile. Il nostro esempio usa tre funzioni lambda, ma dà loro un nome. Osservatelo attentamente:

```
two = lambda: 2
sqr = lambda x: x * x
pwr = lambda x, y: x ** y
```

```
for a in range(-2, 3):
    print(sqr(a), end=" ")
    print(pwr(a, two()))
```

Analizziamolo:

- la prima lambda è una **funzione anonima priva di parametri** che restituisce sempre 2. Poiché **l'abbiamo assegnata a una variabile di nome** due, possiamo dire che la funzione non è più anonima e possiamo usare il nome per richiamarla.
- la seconda è una **funzione anonima a un parametro** che restituisce il valore del suo argomento al quadrato. Anche questa funzione è stata chiamata così.
- la terza lambda **prende due parametri** e restituisce il valore del primo elevato alla potenza del secondo. Il nome della variabile che contiene il lambda parla da sé. Non usiamo pow per evitare confusione con la funzione incorporata che ha lo stesso nome e lo stesso scopo.

Il programma produce il seguente output:

```
4 4
1 1
0 0
1 1
4 4
```

Questo esempio è abbastanza chiaro da mostrare come vengono dichiarate le lambda e come si comportano, ma non dice nulla sul perché siano necessarie e a cosa servano, dato che possono essere tutte sostituite con funzioni Python di routine.

Dove sta il beneficio?

## Come usare le lambda e per cosa?

La parte più interessante dell'uso delle lambda si presenta quando è possibile utilizzarle nella loro forma pura, **come parti anonime di codice destinate a valutare un risultato.**

Immaginiamo di aver bisogno di una funzione (la chiameremo `print_function`) che stampi i valori di una data (altra) funzione per un insieme di argomenti selezionati.

Vogliamo che `print_function` sia universale: dovrebbe accettare un insieme di argomenti inseriti in un elenco e una funzione da valutare, entrambi come argomenti.

Guardate l'esempio

```
def print_function(args, fun):
    for x in args:
        print('f(', x,')=', fun(x), sep='')
def poly(x):
    return 2 * x**2 - 4 * x + 2
print_function([x for x in range(-2, 3)], poly)
```

Ecco come abbiamo implementato l'idea.

Analizziamola. La funzione `print_function()` accetta due parametri:

- il primo, un elenco di argomenti per i quali si desidera stampare i risultati;
- la seconda, una funzione che deve essere invocata tante volte quanto il numero di valori raccolti nel primo parametro.

Nota: abbiamo definito anche una funzione denominata `poly()` - questa è la funzione di cui stamperemo i valori. Il calcolo che la funzione esegue non è molto sofisticato: è il polinomio (da cui il nome) di un modulo:

$$f(x) = 2x^2 - 4x + 2$$

Il nome della funzione viene passato alla funzione `print_function()` insieme a un insieme di cinque argomenti diversi, costruiti con una clausola di comprensione della lista.

Il codice stampa le seguenti righe:

`f(-2)=18`

`f(-1)=8`

`f(0)=2`

`f(1)=0`

`f(2)=2`

Possiamo evitare di definire la funzione `poly()`, visto che non la useremo più di una volta? Sì, è possibile: questo è il vantaggio di una `lambda`.

Guardate l'esempio qui sotto. Riuscite a notare la differenza?

Look at the example below. Can you see the difference?

```
def print_function(args, fun):  
    for x in args:  
        print('f(', x, ')=', fun(x), sep='')
```

```
print_function([x for x in range(-2, 3)], lambda x: 2 * x**2 - 4 * x + 2)
```

La `print_function()` è rimasta esattamente la stessa, ma non c'è più la funzione `poly()`. Non ne abbiamo più bisogno, poiché il polinomio è ora direttamente all'interno dell'invocazione di `print_function()` sotto forma di una `lambda` definita nel modo seguente:

```
lambda x: 2 * x**2 - 4 * x + 2
```

Il codice è diventato più breve, più chiaro e più leggibile.

Mostriamo un altro punto in cui le `lambda` possono essere utili. Inizieremo con una descrizione di `map()`, una funzione integrata di Python. Il suo nome non è troppo descrittivo, la sua idea è semplice e la funzione stessa è davvero utilizzabile.

## Lambda e la funzione map()

Nel più semplice dei casi possibili, la funzione map():

map(funzione, elenco)

riceve due argomenti:

- una funzione;
- un elenco.

La descrizione precedente è estremamente semplificata, in quanto:

- il secondo argomento di map() può essere una qualsiasi entità che può essere iterata (ad esempio, una tupla o un semplice generatore)
- map() può accettare più di due argomenti.

La **funzione** map() **applica la funzione passata dal primo argomento a tutti gli elementi del secondo argomento e restituisce un iteratore che consegna tutti i risultati successivi della funzione.**

È possibile utilizzare l'iteratore risultante in un ciclo o convertirlo in un elenco utilizzando la funzione list().

Si può vedere un ruolo per qualsiasi lambda qui?

Guardate il codice: abbiamo usato due lambda.

```
list_1 = [x for x in range(5)]
list_2 = list(map(lambda x: 2 ** x, list_1))
print(list_2)
for x in map(lambda x: x * x, list_2):
    print(x, end=' ')
print()
```

Questo è l'intrigo:

- costruire l'elenco\_1 con valori da 0 a 4;
- Successivamente, si usa map insieme alla prima lambda per creare una nuova lista in cui tutti gli elementi sono stati valutati come 2 elevato alla potenza presa dall'elemento corrispondente di lista\_1;
- viene stampato l'elenco\_2;
- nel passo successivo, utilizziamo di nuovo la funzione map() per utilizzare il generatore che restituisce e per stampare direttamente tutti i valori che fornisce; come si può vedere, qui abbiamo impegnato il secondo lambda - che si limita a squadrare ogni elemento della lista\_2.

Provate a immaginare lo stesso codice senza lambda. Sarebbe migliore? È improbabile.



## Lambda e la funzione filter()

Un'altra funzione Python che può essere notevolmente abbellita dall'applicazione di un lambda è filter(). Si aspetta lo stesso tipo di argomenti di map(), ma fa qualcosa di diverso: **filtra il suo secondo argomento facendosi guidare dalle indicazioni che provengono dalla funzione specificata come primo argomento** (la funzione viene invocata per ogni elemento dell'elenco, proprio come in map()). Gli elementi che restituiscono True dalla funzione **passano il filtro**, mentre gli altri vengono scartati. L'esempio mostra la funzione filter() in azione.

```
from random import seed, randint
seed()
data = [randint(-10,10) for x in range(5)]
filtered = list(filter(lambda x: x > 0 and x % 2 == 0, data))

print(data)
print(filtered)
```

Nota: abbiamo utilizzato il modulo random per inizializzare il generatore di numeri casuali (da non confondere con i generatori di cui abbiamo appena parlato) con la funzione seed() e per produrre cinque valori interi casuali da -10 a 10 utilizzando la funzione randint().

L'elenco viene quindi filtrato e vengono accettati solo i numeri pari e maggiori di zero.

Naturalmente, non è detto che otterrete gli stessi risultati, ma i nostri risultati sono stati questi:

```
[6, 3, 3, 2, -7]
```

```
[6, 2]
```

## Un breve sguardo alle chiusure

Cominciamo con una definizione: la **chiusura** è una tecnica che permette di conservare i valori nonostante il **contesto in cui sono stati creati non esista più**. Intricato? Un po'.

Analizziamo un semplice esempio:

```
def outer(par):  
    loc = par
```

```
var = 1  
outer(var)
```

```
print(par)  
print(loc)
```

L'esempio è ovviamente errato.

Le ultime due righe causeranno un'eccezione `NameError`: né `par` né `loc` sono accessibili al di fuori della funzione. Entrambe le variabili esistono quando e solo quando viene eseguita la funzione `outer()`.

Guardate l'esempio

```
def outer(par):  
    loc = par  
    def inner():  
        return loc  
    return inner  
var = 1  
fun = outer(var)  
print(fun())
```

Abbiamo modificato il codice in modo significativo.

C'è un elemento nuovo: una funzione (chiamata `inner`) all'interno di un'altra funzione (chiamata `outer`).

Come funziona? Come qualsiasi altra funzione, tranne per il fatto che `inner()` può essere invocata solo dall'interno di `outer()`. Possiamo dire che `inner()` è lo strumento privato di `outer()`: nessun'altra parte del codice può accedervi.

Guardate con attenzione:

la funzione `inner()` restituisce il valore della variabile accessibile all'interno del suo ambito, poiché `inner()` può utilizzare qualsiasi entità a disposizione di `outer()`

la funzione `outer()` restituisce la funzione `inner()` stessa; più precisamente, restituisce una copia della funzione `inner()`, quella che è stata congelata al momento dell'invocazione di `outer()`; la funzione congelata contiene il suo ambiente completo, compreso lo stato di tutte le variabili locali, il che significa anche che il valore di `loc` viene mantenuto con successo, sebbene `outer()` abbia cessato di esistere molto tempo fa.

In effetti, il codice è pienamente valido e viene emesso:

1

La funzione restituita durante l'invocazione di `outer()` è una **chiusura**.

### **Breve sguardo alle chiusure: continua**

**Una chiusura deve essere invocata esattamente nello stesso modo in cui è stata dichiarata.**

Nell'esempio seguente:

```
def outer(par):  
    loc = par  
  
    def inner():  
        restituire loc  
    restituire l'interno
```

```
var = 1  
fun = outer(var)  
print(fun())
```

la funzione inner() è priva di parametri, quindi dobbiamo invocarla senza argomenti.

Ora guardate il codice

```
def make_closure(par):
```

```
    loc = par
```

```
    def power(p):
```

```
        return p ** loc
```

```
    return power
```

```
fsqr = make_closure(2)
```

```
fcub = make_closure(3)
```

```
for i in range(5):
```

```
    print(i, fsqr(i), fcub(i))
```

È possibile **dichiarare una chiusura dotata di un numero arbitrario di parametri**, ad esempio uno, proprio come la funzione `power()`.

Ciò significa che la chiusura non solo fa uso dell'ambiente congelato, ma può anche **modificare il proprio comportamento utilizzando valori presi dall'esterno**.

Questo esempio mostra un'altra circostanza interessante: si possono **creare tutte le chiusure che si vogliono, usando lo stesso pezzo di codice**. Questo viene fatto con una funzione chiamata `make_closure()`. Nota:

- la prima chiusura ottenuta da `make_closure()` definisce uno strumento che squaderna il suo argomento;
- il secondo è stato concepito per ridurre al cubo l'argomentazione.

Per questo motivo il codice produce il seguente risultato:

0 0 0

1 1 1

2 4 8

3 9 27

4 16 64

Eseguite i vostri test.

## Punti di forza

1. Un **iteratore** è un oggetto di una classe che fornisce almeno **due** metodi (senza contare il costruttore):

`__iter__()` viene invocato una volta quando l'iteratore viene creato e restituisce l'oggetto dell'iteratore **stesso**;

`__next__()` viene invocato per fornire il **valore dell'iterazione successiva** e solleva l'eccezione `StopIteration` quando l'iterazione **giunge al termine**.

2. L'istruzione `yield` può essere utilizzata solo all'interno di funzioni. L'istruzione `yield` sospende l'esecuzione della funzione e fa sì che la funzione restituisca come risultato l'argomento di `yield`. Una funzione di questo tipo non può essere invocata in modo regolare: il suo unico scopo è quello di essere usata come **generatore** (cioè in un contesto che richiede una serie di valori, come un ciclo `for`).

3. Un'**espressione condizionale** è un'espressione costruita utilizzando l'operatore `if-else`. Ad esempio:

```
print(Vero se 0 >= 0 altrimenti Falso)
```

Vero.

4. La **comprensione di una lista** diventa un **generatore** quando è usata all'interno di **parentesi** (se è usata tra parentesi, produce una lista regolare). Ad esempio:

```
per x in (el * 2 per el in range(5)):
```

```
    stampa(x)
```

02468.



5. Una **funzione lambda** è uno strumento per creare **funzioni anonime**. Ad esempio:

```
def foo(x, f):  
    restituire f(x)
```

```
print(pippo(9, lambda x: x ** 0,5))
```

3.0.

6. La funzione `map(fun, list)` crea una **copia** dell'argomento lista e applica la funzione `fun` a tutti i suoi elementi, restituendo un **generatore** che fornisce il nuovo contenuto della lista elemento per elemento. Ad esempio:

```
short_list = ['mitone', 'pitone', 'cadde', 'su', 'il', 'pavimento'].  
new_list = list(map(lambda s: s.title(), short_list))  
print(new_list)
```

```
['Mython', 'Python', 'Fell', 'On', 'The', 'Floor'].
```

7. La funzione `filter(fun, list)` crea una **copia** di questi elementi dell'elenco, che fanno sì che la funzione `fun` restituisca `True`. Il risultato della funzione è un **generatore** che fornisce il nuovo contenuto dell'elenco elemento per elemento. Ad esempio:

```
short_list = [1, "Python", -1, "Monty"]
new_list = list(filter(lambda s: isinstance(s, str), short_list))
print(new_list)
```

`['Python', 'Monty']`.

8. Una chiusura è una tecnica che consente di **memorizzare i valori** nonostante il **contesto** in cui sono stati creati **non esista più**. Ad esempio:

```
def tag(tg):
    tg2 = tg
    tg2 = tg[0] + '/' + tg[1:]

    def inner(str):
        return tg + str + tg2
    return inner
b_tag = tag('<b>')
print(b_tag('Monty Python'))
```

outputs `<b>Monty Python</b>`

### Esercizio 1

Qual è l'output previsto del seguente codice?

```
class Vowels:
    def __init__(self):
        self.vow = "aeiouy " # Yes, we know that y is not always considered a vowel.
        self.pos = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self.pos == len(self.vow):
            raise StopIteration
        self.pos += 1
        return self.vow[self.pos - 1]

vowels = Vowels()
for v in vowels:
    print(v, end=' ')
```

Soluzione:

a e i o u y

## Esercizio 2

Scrivete una funzione **lambda**, impostando il bit meno significativo del suo argomento intero, e applicatela alla funzione `map()` per produrre la stringa `1 3 3 5` sulla console.

```
any_list = [1, 2, 3, 4]
even_list = # Complete the line here.
print(even_list)
```

Soluzione

```
list(map(lambda n: n | 1, any_list))
```

### Esercizio 3

Qual è l'output previsto del seguente codice?

```
def replace_spaces(replacement='*'):
    def new_replacement(text):
        return text.replace(' ', replacement)
    return new_replacement
stars = replace_spaces()
print(stars("And Now for Something Completely Different"))
```

Soluzione:

And\*Now\*for\*Something\*Completely\*Different

### Nota

La [PEP 8](#), la Guida allo stile del codice Python, raccomanda di **non assegnare le lambda alle variabili, ma di definirle come funzioni.**

Ciò significa che è meglio usare una dichiarazione def ed evitare di usare una dichiarazione di assegnazione che lega un'espressione lambda a un identificatore. Analizzate il codice sottostante:

```
# Recommended:
def f(x): return 3*x
# Not recommended:
f = lambda x: 3*x
```

Il collegamento di lambda a identificatori generalmente duplica la funzionalità dell'istruzione def. L'uso delle dichiarazioni def, invece, genera più righe di codice.

È importante capire che spesso la realtà ama disegnare i propri scenari, che non seguono necessariamente le convenzioni o le raccomandazioni formali. La decisione di seguirle o meno dipenderà da molti fattori: le vostre preferenze, le altre convenzioni adottate, le linee guida interne dell'azienda, la compatibilità con il codice esistente, ecc. Tenetene conto.

## Accesso ai file dal codice Python

Uno dei problemi più comuni nel lavoro dello sviluppatore è l'**elaborazione dei dati memorizzati nei file**, che di solito sono memorizzati fisicamente su dispositivi di archiviazione - dischi rigidi, ottici, di rete o a stato solido.

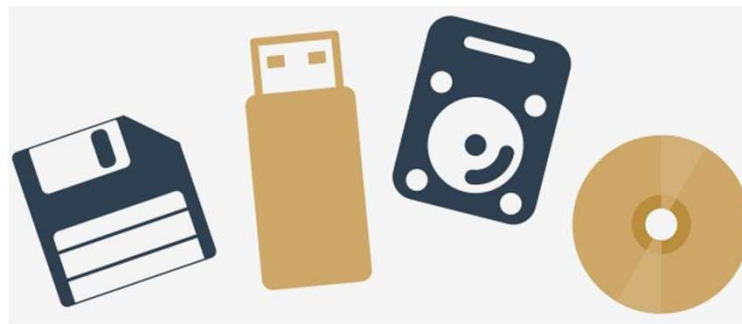
È facile immaginare un programma che ordina 20 numeri, ed è altrettanto facile immaginare che l'utente di questo programma inserisca questi venti numeri direttamente dalla tastiera.

È molto più difficile immaginare lo stesso compito quando ci sono 20.000 numeri da ordinare e non c'è un solo utente in grado di inserire questi numeri senza commettere errori.

È molto più semplice immaginare che questi numeri siano memorizzati nel file del disco che viene letto dal programma. Il programma ordina i numeri e non li invia allo schermo, ma crea un nuovo file e vi salva la sequenza ordinata di numeri.

Se vogliamo implementare un semplice database, l'unico modo per memorizzare le informazioni tra un'esecuzione e l'altra del programma è salvarle in un file (o più file, se il database è più complesso).

In linea di principio, qualsiasi problema di programmazione non semplice si basa sull'uso di file, sia che si tratti di elaborare immagini (memorizzate in file), moltiplicare matrici (memorizzate in file) o calcolare salari e tasse (leggendo dati memorizzati in file).



Vi chiederete perché abbiamo aspettato fino ad ora per mostrarvi questi problemi. La risposta è molto semplice: il modo in cui Python accede ai file e li elabora è implementato utilizzando un insieme coerente di oggetti. Non c'è momento migliore per parlarne.

## Nomi dei file

I diversi sistemi operativi possono trattare i file in modo diverso. Ad esempio, Windows utilizza una convenzione di denominazione diversa da quella adottata nei sistemi Unix/Linux.

Se utilizziamo la nozione di nome di file canonico (un nome che definisce in modo univoco la posizione del file, indipendentemente dal suo livello nell'albero delle directory), possiamo capire che questi nomi hanno un aspetto diverso in Windows e in Unix/Linux:

### Windows

```
C:\directory\file
```

### Linux

```
/directory/files
```



Come si può notare, i sistemi derivati da Unix/Linux non utilizzano la lettera dell'unità disco (ad esempio, C:) e tutte le directory crescono da una directory principale chiamata /, mentre i sistemi Windows riconoscono la directory principale come \.

Inoltre, i nomi dei file dei sistemi Unix/Linux sono sensibili alle maiuscole e alle minuscole. I sistemi Windows memorizzano il caso delle lettere utilizzate nel nome del file, ma non fanno alcuna distinzione tra i casi.

Ciò significa che queste due stringhe: ThisIsTheNameOfTheFile e thisisthenameofthefile descrivono due file diversi nei sistemi Unix/Linux, ma sono lo stesso nome per un solo file nei sistemi Windows.

La differenza principale e più evidente è che bisogna utilizzare **due diversi separatori per i nomi delle directory**: \ in Windows e / in Unix/Linux.

Questa differenza non è molto importante per l'utente normale, ma è **molto importante quando si scrivono programmi in Python**.

Per capirne il motivo, provate a ricordare il ruolo molto specifico svolto dal simbolo \ all'interno delle stringhe Python.

### **Nomi dei file: continua**

Supponiamo di essere interessati a un particolare file situato nella directory `dir` e denominato `file`.

Si supponga inoltre di voler assegnare una stringa contenente il nome del file.

Nei sistemi Unix/Linux, l'aspetto può essere il seguente:

```
nome = "/dir/file«
```

Ma se si cerca di codificarlo per il sistema Windows:

```
nome = "profilo"
```

si avrà una spiacevole sorpresa: o Python genererà un errore, o l'esecuzione del programma si comporterà in modo strano, come se il nome del file fosse stato distorto in qualche modo.

In realtà, non è affatto strano, ma piuttosto ovvio e naturale. Python usa la `\` come carattere di escape (come `\n`).

Ciò significa che i nomi dei file di Windows devono essere scritti come segue:

```
nome = "profilo"
```

Fortunatamente, esiste anche un'altra soluzione. Python è abbastanza intelligente da essere in grado di convertire gli slash in backslash ogni volta che scopre che è richiesto dal sistema operativo.

Ciò significa che qualsiasi incarico di seguito riportato:

```
nome = "/dir/file"
```

```
nome = "c:/dir/file"
```

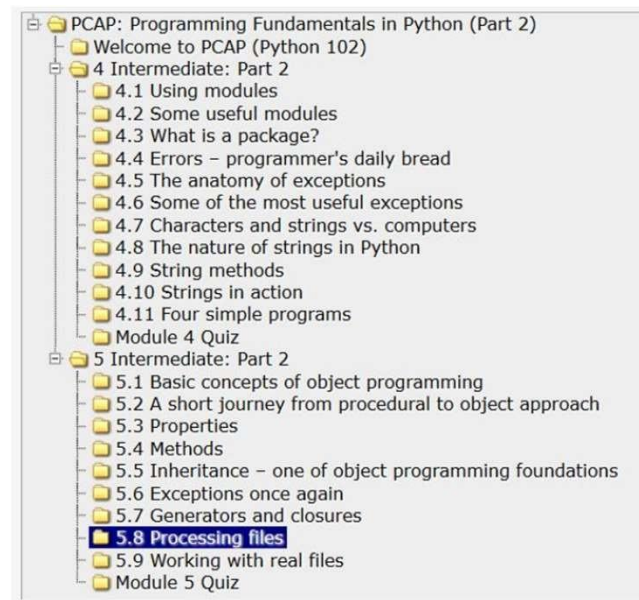
funzionerà anche con Windows.

Qualsiasi programma scritto in Python (e non solo in Python, perché questa convenzione si applica praticamente a tutti i linguaggi di programmazione) non comunica direttamente con i file, ma attraverso alcune entità astratte che vengono denominate in modo diverso nei vari linguaggi o ambienti: i termini più usati sono **handle** o **stream** (qui li useremo come sinonimi).

Il programmatore, disponendo di un insieme più o meno ricco di funzioni/metodi, è in grado di eseguire determinate operazioni sul flusso, che influiscono sui file reali, utilizzando meccanismi contenuti nel kernel del sistema operativo.

In questo modo, è possibile implementare il processo di accesso a qualsiasi file, anche se il nome del file è sconosciuto al momento della scrittura del programma.

Le operazioni eseguite con il flusso astratto riflettono le attività relative al file fisico.



Per collegare (bind) lo stream con il file, è necessario eseguire un'operazione esplicita.

L'operazione di collegamento del flusso con un file viene chiamata **apertura del file**, mentre la disconnessione di questo collegamento viene chiamata **chiusura del file**.

Di conseguenza, la conclusione è che la prima operazione eseguita sullo stream è sempre open e l'ultima è close.

Il programma, in effetti, è libero di manipolare lo stream tra questi due eventi e di gestire il file associato.

Questa libertà è limitata, ovviamente, dalle caratteristiche fisiche del file e dal modo in cui il file è stato aperto.

Ripetiamo che l'apertura del flusso può fallire e ciò può accadere per diversi motivi: il più comune è la mancanza di un file con il nome specificato.

Può anche accadere che il file fisico esista, ma che il programma non sia autorizzato ad aprirlo. C'è anche il rischio che il programma abbia aperto troppi flussi e il sistema operativo specifico potrebbe non consentire l'apertura simultanea di più di n file (ad esempio, 200).

Un programma ben scritto dovrebbe rilevare queste aperture fallite e reagire di conseguenza.

## Flussi di file

L'apertura del flusso non è solo associata al file, ma deve anche dichiarare il modo in cui il flusso verrà elaborato. Questa dichiarazione è chiamata **modalità di apertura**.

Se l'apertura ha successo, il **programma potrà eseguire solo le operazioni coerenti con la modalità di apertura dichiarata**.

Le operazioni di base eseguite sul flusso sono due:

- **lettura** dal flusso: le porzioni di dati vengono recuperate dal file e collocate in un'area di memoria gestita dal programma (ad esempio, una variabile);
- **scrivere** sul flusso: le porzioni di dati dalla memoria (ad esempio, una variabile) vengono trasferite al file.

Esistono tre modalità di base per l'apertura del flusso:

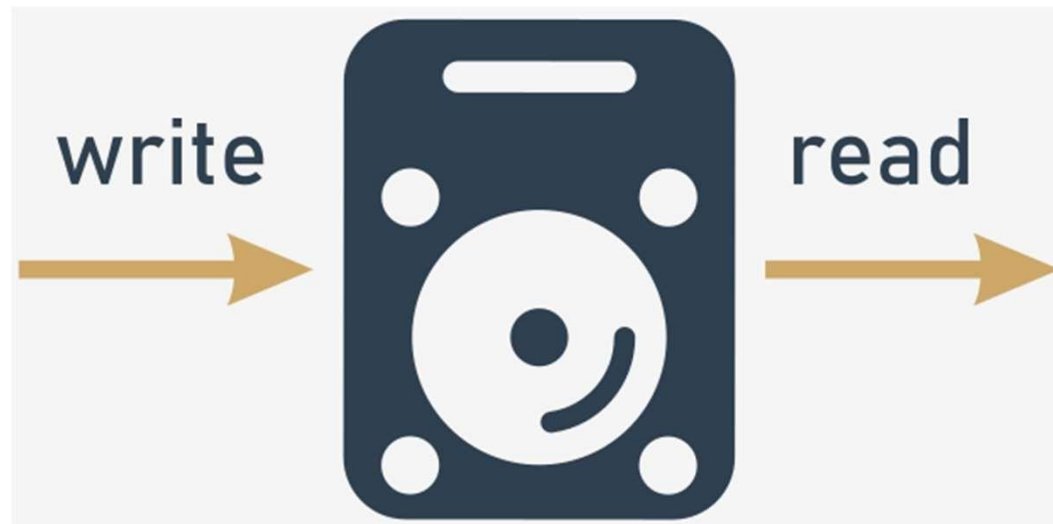
- **modalità di lettura**: un flusso aperto in questa modalità consente **solo operazioni di lettura**; il tentativo di scrittura sul flusso causerà un'eccezione (l'eccezione è denominata `UnsupportedOperation`, che eredita `OSError` e `ValueError` e proviene dal modulo `io`);
- **modalità di scrittura**: un flusso aperto in questa modalità consente **solo operazioni di scrittura**; il tentativo di leggere il flusso causerà l'eccezione di cui sopra;
- **modalità di aggiornamento**: un flusso aperto in questa modalità consente **sia la scrittura che la lettura**.

Prima di parlare di come manipolare i flussi, è doveroso fornire alcune spiegazioni. **Il flusso si comporta quasi come un registratore.**

Quando si legge qualcosa da uno stream, una testina virtuale si sposta sullo stream in base al numero di byte trasferiti dallo stream.

Quando si scrive qualcosa nello stream, la stessa testina si muove lungo lo stream registrando i dati dalla memoria.

Ogni volta che parliamo di lettura e scrittura sullo stream, provate a immaginare questa analogia. I libri di programmazione si riferiscono a questo meccanismo come alla **posizione corrente del file**, e anche noi useremo questo termine.



È necessario ora mostrare l'oggetto responsabile della rappresentazione dei flussi nei programmi.

## File handles

Python presuppone che **ogni file sia nascosto dietro un oggetto di classe adeguata**.

Naturalmente, è difficile non chiedersi come interpretare la parola "*adeguato*".

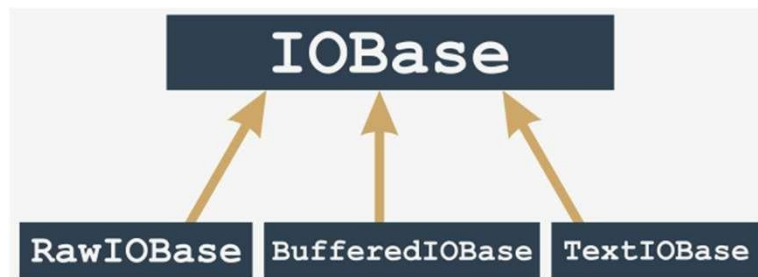
I file possono essere elaborati in molti modi diversi, alcuni dei quali dipendono dal contenuto del file, altri dalle intenzioni del programmatore.

In ogni caso, file diversi possono richiedere serie di operazioni diverse e comportarsi in modi diversi.

Un oggetto di una classe adeguata viene **creato all'apertura del file e annichilito al momento della chiusura**.

Tra questi due eventi, è possibile utilizzare l'oggetto per specificare le operazioni da eseguire su un determinato flusso. Le operazioni consentite sono imposte dal **modo in cui è stato aperto il file**.

In generale, l'oggetto proviene da una delle classi mostrate qui:



Nota: non si utilizzano mai i costruttori per dare vita a questi oggetti. L'unico modo per **ottenerli è invocare la funzione** `open()`.

La funzione analizza gli argomenti forniti e crea automaticamente l'oggetto richiesto.

Se si vuole **eliminare l'oggetto, si invoca il metodo** `close()`.

L'invocazione interrompe la connessione all'oggetto e al file e rimuove l'oggetto.

Per i nostri scopi, ci occuperemo solo dei flussi rappresentati dagli oggetti `BufferedIOBase` e `TextIOBase`. Il perché lo capirete presto.

In base al tipo di contenuto del flusso, **tutti i flussi sono suddivisi in flussi di testo e flussi binari.**

I flussi di testo sono strutturati in righe, cioè contengono caratteri tipografici (lettere, cifre, punteggiatura, ecc.) disposti in file (righe), come si vede a occhio nudo quando si osserva il contenuto del file nell'editor.

Questo file viene scritto (o letto) principalmente carattere per carattere, o riga per riga.

I flussi binari non contengono testo, ma una sequenza di byte di qualsiasi valore. Questa sequenza può essere, ad esempio, un programma eseguibile, un'immagine, un clip audio o video, un file di database, ecc.

Poiché questi file non contengono righe, le letture e le scritture riguardano porzioni di dati di qualsiasi dimensione. Quindi i dati vengono letti/scritti byte per byte, o blocco per blocco, dove la dimensione del blocco di solito va da uno a un valore scelto arbitrariamente.

A questo punto si presenta un problema sottile. Nei sistemi Unix/Linux, la fine della riga è contrassegnata da un singolo carattere chiamato LF (codice ASCII 10), che nei programmi Python viene indicato come `\n`.

Altri sistemi operativi, in particolare quelli derivati dal preistorico sistema CP/M (che si applica anche ai sistemi della famiglia Windows), utilizzano una convenzione diversa: la fine della riga è contrassegnata da una coppia di caratteri, CR e LF (codici ASCII 13 e 10) che possono essere codificati come `\r\n`.





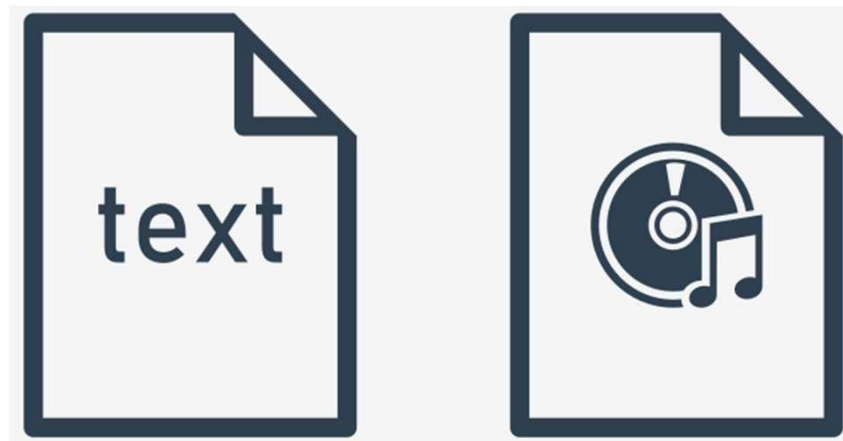
Questa ambiguità può causare diverse conseguenze spiacevoli.

Se si crea un programma responsabile dell'elaborazione di un file di testo e questo è scritto per Windows, è possibile riconoscere le estremità delle righe trovando i caratteri `\r\n`, ma lo stesso programma eseguito in un ambiente Unix/Linux sarà completamente inutile, e viceversa: il programma scritto per i sistemi Unix/Linux potrebbe essere inutile in Windows.

Tali caratteristiche indesiderate del programma, che impediscono o ostacolano l'uso del programma in ambienti diversi, sono chiamate **non portabilità**.

Allo stesso modo, la caratteristica del programma che consente l'esecuzione in ambienti diversi è chiamata **portabilità**. Un programma dotato di tale caratteristica è chiamato **programma portatile**.

Dato che i problemi di portabilità erano (e sono tuttora) molto seri, si è deciso di risolvere definitivamente il problema in un modo che non coinvolgesse l'attenzione dello sviluppatore.



È stato fatto a livello di classi, che sono responsabili della lettura e della scrittura di caratteri da e verso lo stream. Funziona nel modo seguente:

- quando il flusso è aperto e si sa che i dati del file associato saranno elaborati come testo (o non c'è alcun avviso), viene **commutato in modalità testo**;
- Durante la lettura/scrittura di righe da/verso il file associato, non avviene nulla di particolare in ambiente Unix, ma quando le stesse operazioni vengono eseguite in ambiente Windows, si verifica un processo chiamato **traduzione dei caratteri newline**: quando si legge una riga dal file, ogni coppia di caratteri `\r\n` viene sostituita con un singolo carattere `\n`, e viceversa; durante le operazioni di scrittura, ogni carattere `\n` viene sostituito con una coppia di caratteri `\r\n`;
- il meccanismo è completamente **trasparente** al programma, che può essere scritto come se fosse destinato esclusivamente all'elaborazione di file di testo Unix/Linux; anche il codice sorgente eseguito in ambiente Windows funzionerà correttamente;
- quando il flusso è aperto e si consiglia di farlo, il suo contenuto viene preso così com'è, **senza alcuna conversione** - nessun byte viene aggiunto o omesso.

## Apertura dei flussi

L'**apertura del flusso** viene eseguita da una funzione che può essere invocata nel modo seguente:

```
stream = open(file, mode = 'r', encoding = None)
```

Analizziamolo:

- il nome della funzione (open) parla da sé; se l'apertura ha successo, la funzione restituisce un oggetto stream; altrimenti, viene sollevata un'eccezione (ad esempio, FileNotFoundError **se il file che si intende leggere non esiste**);
- il primo parametro della funzione (file) specifica il nome del file da associare allo stream;
- il secondo parametro (mode) specifica la modalità di apertura utilizzata per lo stream; si tratta di una stringa riempita con una sequenza di caratteri, ognuno dei quali ha un significato particolare (maggiori dettagli a breve);
- il terzo parametro (codifica) specifica il tipo di codifica (ad esempio, UTF-8 quando si lavora con file di testo)
- l'apertura deve essere la prima operazione eseguita sul flusso.

Nota: gli argomenti mode e encoding possono essere omessi; in tal caso vengono assunti i loro valori predefiniti. La modalità di apertura predefinita è la lettura in modalità testo, mentre la codifica predefinita dipende dalla piattaforma utilizzata.

Vi presentiamo ora le modalità di apertura più importanti e utili. Pronti?

## Apertura dei flussi: modalità

r modalità aperta: lettura

- il flusso verrà aperto in **modalità di lettura**;
- il file associato allo stream **deve esistere** e deve essere leggibile, altrimenti la funzione open() solleva un'eccezione.

w modalità aperta: scrittura

- il flusso verrà aperto in **modalità di scrittura**;
- **non è necessario che** il file associato allo stream **esista**; se non esiste, verrà creato; se esiste, verrà troncato alla lunghezza di zero (cancellato); se la creazione non è possibile (ad esempio, a causa dei permessi del sistema), la funzione open() solleva un'eccezione.

a modalità aperta: append

- lo stream verrà aperto in **modalità append**;
- **non è necessario che** il file associato al flusso **esista**; se non esiste, verrà creato; se esiste, la testina di registrazione virtuale verrà impostata alla fine del file (il contenuto precedente del file rimane intatto).

r+ modalità aperta: lettura e aggiornamento

- il flusso verrà aperto in **modalità di lettura e aggiornamento**;
- il file associato allo stream **deve esistere e deve essere scrivibile**, altrimenti la funzione open() solleva un'eccezione;
- per lo stream sono consentite sia operazioni di lettura che di scrittura.

w+ modalità aperta: scrittura e aggiornamento

- il flusso verrà aperto in modalità di **scrittura e aggiornamento**;
- **non è necessario che** il file associato al flusso **esista**; se non esiste, verrà creato; il contenuto precedente del file rimane inalterato;
- per lo stream sono consentite sia operazioni di lettura che di scrittura.

## Selezione delle modalità testo e binarie

Se alla fine della stringa di modalità c'è una lettera b, significa che lo stream deve essere aperto in **modalità binaria**.

Se la stringa di modalità termina con una lettera t, il flusso viene aperto in **modalità testo**.

La modalità testo è il comportamento predefinito assunto quando non viene utilizzato alcuno specificatore di modalità binaria/testo.

Infine, l'apertura riuscita del file imposterà la posizione corrente del file (la testina virtuale di lettura/scrittura) prima del primo byte del file **se la modalità non è a** e dopo l'ultimo byte del file **se la modalità è impostata su a**.

| Modali<br>tà<br>testo | Modalità<br>binaria | Descrizio<br>ne          |
|-----------------------|---------------------|--------------------------|
| rt                    | rb                  | leggere                  |
| wt                    | wb                  | scrivere                 |
| a                     | ab                  | append                   |
| r+t                   | r+b                 | leggere e<br>aggiornare  |
| w+t                   | w+b                 | scrivere e<br>aggiornare |

## EXTRA

È anche possibile aprire un file per la sua creazione esclusiva. A tale scopo, si può utilizzare la modalità x open. Se il file esiste già, la funzione open() solleverà un'eccezione.

## Aprire il flusso per la prima volta

Immaginiamo di voler sviluppare un programma che legga il contenuto del file di testo denominato:  
C:\Users\User\Desktop\file.txt.

Come aprire il file per la lettura? Ecco il frammento di codice pertinente:

```
try:
```

```
    stream = open("C:\Users\User\Desktop\file.txt", "rt")
```

```
    # Processing goes here.
```

```
    stream.close()
```

```
except Exception as exc:
```

```
    print("Cannot open the file:", exc)
```

Cosa sta succedendo qui?

- apriamo il blocco try-except, poiché vogliamo gestire gli errori di runtime in modo soft;
- utilizziamo la funzione open() per cercare di aprire il file specificato (notare il modo in cui abbiamo specificato il nome del file)
- La modalità di apertura è definita come testo da leggere (poiché **il testo è l'impostazione predefinita**, si può saltare la t nella stringa della modalità).
- In caso di successo, otteniamo un oggetto dalla funzione open() e lo assegniamo alla variabile stream;
- se open() fallisce, gestiamo l'eccezione stampando informazioni complete sull'errore (è sicuramente utile sapere cosa è successo esattamente)

## Flussi pre-apertura

Abbiamo detto in precedenza che qualsiasi operazione sullo stream deve essere preceduta dall'invocazione della funzione `open()`. Esistono tre eccezioni ben definite alla regola.

All'avvio del programma, i tre flussi sono già aperti e non richiedono alcuna preparazione aggiuntiva. Inoltre, il programma può utilizzare questi flussi in modo esplicito, se si ha cura di importare il modulo `sys`:

```
import sys
```

perché è lì che si trova la dichiarazione dei tre flussi.

I nomi di questi flussi sono: `sys.stdin`, `sys.stdout` e `sys.stderr`.

Analizziamoli:

### **sys.stdin**

- `stdin` (come *input standard*)
- il flusso `stdin` è normalmente associato alla tastiera, pre-aperto per la lettura e considerato come la fonte primaria di dati per i programmi in esecuzione;
- la nota funzione `input()` legge i dati da `stdin` per impostazione predefinita.

### **sys.stdout**

- `stdout` (come *output standard*)
- il flusso `stdout` è normalmente associato allo schermo, pre-aperto per la scrittura, considerato come l'obiettivo principale per l'output di dati da parte del programma in esecuzione;
- la nota funzione `print()` invia i dati al flusso `stdout`.



## **sys.stderr**

- `stderr` (come *output di errore standard*)
- il flusso `stderr` è normalmente associato allo schermo, pre-aperto per la scrittura, considerato come il luogo principale in cui il programma in esecuzione deve inviare informazioni sugli errori incontrati durante il suo lavoro;
- non abbiamo presentato alcun metodo per inviare i dati a questo flusso (lo faremo presto, lo promettiamo)
- la separazione dello `stdout` (risultati utili prodotti dal programma) dallo `stderr` (messaggi di errore, innegabilmente utili ma che non forniscono risultati) dà la possibilità di reindirizzare questi due tipi di informazioni verso i diversi target. Una trattazione più approfondita di questo argomento esula dagli scopi del nostro corso. Il manuale del sistema operativo fornirà maggiori informazioni su questi argomenti.

## Flussi di chiusura

L'ultima operazione eseguita su un flusso (questo non include i flussi stdin, stdout e stderr che non lo richiedono) deve essere la **chiusura**.

Questa azione viene eseguita da un metodo invocato dall'oggetto open stream: `stream.close()`.

- il nome della funzione è sicuramente auto-commentante: `close()`
- la funzione non si aspetta alcun argomento; il flusso non deve essere aperto.
- la funzione non restituisce nulla ma solleva l'eccezione `IOException` in caso di errore;
- la maggior parte degli sviluppatori crede che la funzione `close()` di abbia sempre successo e che quindi non sia necessario verificare se ha svolto correttamente il suo compito.

Questa convinzione è giustificata solo in parte. Se lo stream è stato aperto per la scrittura e poi sono state eseguite una serie di operazioni di scrittura, può accadere che i dati inviati allo stream non siano ancora stati trasferiti al dispositivo fisico (a causa di un meccanismo chiamato **caching** o **buffering**).

Poiché la chiusura dello stream forza il flush dei buffer, è possibile che il flush fallisca e che quindi fallisca anche `close()`.

Abbiamo già parlato dei guasti causati dalle funzioni che operano con i flussi, ma non abbiamo menzionato come identificare esattamente la causa del guasto.

La possibilità di effettuare una diagnosi esiste ed è fornita da un componente di eccezione dei flussi di cui stiamo per parlarvi.

## Diagnosticare i problemi dello stream

L'oggetto IOError è dotato di una proprietà denominata errno (il nome deriva dal *numero dell'errore*) e vi si può accedere come segue:

try:

```
# Some stream operations.
```

except IOError as exc:

```
    print(exc.errno)
```

Il valore dell'attributo errno può essere confrontato con una delle costanti simboliche predefinite definite nel modulo errno.

Vediamo alcune **costanti** selezionate **utili per rilevare gli errori di flusso**:

- errno.EACCES → Autorizzazione negata  
L'errore si verifica quando si tenta, ad esempio, di aprire in scrittura un file con l'attributo di *sola lettura*.
- errno.EBADF → Numero di file errato  
L'errore si verifica quando si cerca, ad esempio, di operare con un flusso non aperto.
- errno.EEXIST → Il file esiste  
L'errore si verifica quando si cerca, ad esempio, di rinominare un file con il suo nome precedente.
- errno.EFBIG → File troppo grande  
L'errore si verifica quando si cerca di creare un file di dimensioni superiori al massimo consentito dal sistema operativo.
- errno.EISDIR → È una directory  
L'errore si verifica quando si cerca di trattare il nome di una directory come il nome di un normale file.

- `errno.EMFILE` → Troppi file aperti  
L'errore si verifica quando si tenta di aprire contemporaneamente un numero di flussi superiore a quello accettabile per il sistema operativo.
- `errno.ENOENT` → No such file or directory  
L'errore si verifica quando si cerca di accedere a un file/directory inesistente.
- `errno.ENOSPC` → Non c'è spazio libero sul dispositivo  
L'errore si verifica quando non c'è spazio libero sul supporto.

L'elenco completo è molto più lungo (comprende anche alcuni codici di errore non correlati all'elaborazione del flusso).

## Diagnosi dei problemi del flusso: continua

Se siete programmatori molto attenti, potreste sentire la necessità di utilizzare una sequenza di istruzioni simile a questa

```
import errno
try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # Actual processing goes here.
    s.close()
except Exception as exc:
    if exc.errno == errno.ENOENT:
        print("The file doesn't exist.")
    elif exc.errno == errno.EMFILE:
        print("You've opened too many files.")
    else:
        print("The error number is:", exc.errno)
```

Fortunatamente, esiste una funzione che può **semplificare** notevolmente **il codice di gestione degli errori**.

Il suo nome è `strerror()`, proviene dal modulo `os` e **si aspetta un solo argomento: un numero di errore**.

Il suo ruolo è semplice: si fornisce un numero di errore e si ottiene una stringa che descrive il significato dell'errore.

Nota: se si passa un codice di errore inesistente (un numero che non è legato ad alcun errore reale), la funzione solleverà l'eccezione `ValueError`.

Ora possiamo semplificare il nostro codice nel modo seguente:

```
from os import strerror
try:
    s = open("c:/users/user/Desktop/file.txt", "rt")
    # Actual processing goes here.
    s.close()
except Exception as exc:
    print("The file could not be opened:", strerror(exc.errno))
```

## Punti di forza

1. Un file deve essere **aperto** prima di poter essere elaborato da un programma e deve essere **chiuso** al termine dell'elaborazione.

L'apertura del file lo associa al **flusso**, che è una rappresentazione astratta dei dati fisici memorizzati sul supporto.

Il modo in cui il flusso viene elaborato è chiamato **modalità di apertura**. Esistono **tre** modalità di apertura:

- **modalità di lettura** - sono consentite solo operazioni di lettura;
- **modalità di scrittura** - sono consentite solo operazioni di scrittura;
- **modalità di aggiornamento** - sono consentite sia le scritture che le letture.

2. A seconda del contenuto fisico del file, è possibile utilizzare diverse classi Python per elaborare i file. In generale, `BufferedIOBase` è in grado di elaborare qualsiasi file, mentre `TextIOBase` è una classe specializzata nell'elaborazione di file di testo (cioè file contenenti testi visibili all'uomo e suddivisi in righe mediante marcatori di nuova riga). Pertanto, i flussi possono essere suddivisi in **binari** e **testuali**.

3. Per aprire un file si utilizza la seguente sintassi della funzione

```
open():  
open(nome_file, mode=open_mode, encoding=text_encoding)
```

L'invocazione crea un oggetto stream e lo associa al file denominato `nome_file`, utilizzando l'`open_mode` specificato e impostando la codifica\_testo specificata, oppure **solleva un'eccezione in caso di errore**.

4. All'avvio del programma sono già aperti tre flussi **predefiniti**:

- `sys.stdin` - input standard;
- `sys.stdout` - output standard;
- `sys.stderr` - output di errore standard.

5. L'oggetto eccezione `IOError`, creato quando qualsiasi operazione sui file fallisce (comprese le operazioni di apertura), contiene una proprietà denominata `errno`, che contiene il codice di completamento dell'azione fallita. Utilizzare questo valore per diagnosticare il problema.



### **Esercizio 1**

Come si codifica il valore dell'argomento mode di una funzione open() se si vuole creare un nuovo file di testo per riempirlo solo con un articolo?

Soluzione

"wt" o "w"

### **Esercizio 2**

Qual è il significato del valore rappresentato da errno.EACCES?

Soluzione

**Permission denied:** you're not allowed to access the file's contents.

### Esercizio 3

Qual è l'output previsto del seguente codice, supponendo che il file chiamato *file* non esista?

```
import errno
try:
    stream = open("file", "rb")
    print("exists")
    stream.close()
except IOError as error:
    if error.errno == errno.ENOENT:
        print("absent")
    else:
        print("unknown")
```

Soluzione

absent

## Elaborazione di file di testo

In questa lezione prepareremo un semplice file di testo con un contenuto breve e semplice.

Verranno mostrate alcune tecniche di base che possono essere utilizzate per **leggere i contenuti dei file** ed elaborarli.

L'elaborazione sarà molto semplice: si copierà il contenuto del file nella console e si conteranno tutti i caratteri letti dal programma.

Ma ricordate: il nostro concetto di file di testo è molto rigido. Nel nostro senso, è un file di testo semplice - può contenere solo testo, senza alcuna decorazione aggiuntiva (formattazione, caratteri diversi, ecc.).

Per questo motivo si dovrebbe evitare di creare il file utilizzando un elaboratore di testo avanzato come MS Word, LibreOffice Writer o simili. Utilizzate le basi del vostro sistema operativo: Blocco note, vim, gedit, ecc.

Se i file di testo contengono caratteri nazionali non coperti dalla codifica ASCII standard, potrebbe essere necessario un ulteriore passaggio. L'invocazione della funzione `open()` può richiedere un argomento che indichi una specifica codifica del testo.

Ad esempio, se si utilizza un sistema operativo Unix/Linux configurato per l'utilizzo di UTF-8 come impostazione di sistema, la funzione `open()` può avere il seguente aspetto:

```
# Apertura di tzop.txt in modalità lettura, restituendolo come oggetto file:  
stream = open('file.txt', 'rt', encoding='utf-8')  
print(stream.read()) # stampa del contenuto del file
```

dove l'argomento codifica deve essere impostato su un valore che è una stringa che rappresenta la corretta codifica del testo (UTF-8, in questo caso).

### **Elaborazione di file di testo: continua**

La lettura del contenuto di un file di testo può essere eseguita con diversi metodi, nessuno dei quali è migliore o peggiore di un altro. Sta a voi decidere quale di questi metodi preferite e vi piace.

Alcuni di essi saranno a volte più maneggevoli, altre volte più problematici. Siate flessibili. Non abbiate paura di cambiare le vostre preferenze.

Il metodo più elementare è quello offerto dalla funzione `read()`, che abbiamo visto in azione nella lezione precedente.

Se applicata a un file di testo, la funzione è in grado di:

- legge un numero desiderato di caratteri (anche uno solo) dal file e li restituisce come stringa;
- legge tutti i contenuti del file e li restituisce come stringa;
- se non c'è più nulla da leggere (la testina di lettura virtuale raggiunge la fine del file), la funzione restituisce una stringa vuota.

Inizieremo con la variante più semplice e utilizzeremo un file chiamato `text.txt`. Il file ha il seguente contenuto:  
Il bello è meglio del brutto.

L'esplicito è meglio dell'implicito.

Semplice è meglio di complesso.

Complesso è meglio di complicato.

Ora guardiamo il codice

```
from os import strerror
```

```
try:
```

```
    cnt = 0
```

```
    s = open('text.txt', "rt")
```

```
    ch = s.read(1)
```

```
    while ch != "":
```

```
        print(ch, end="")
```

```
        cnt += 1
```

```
        ch = s.read(1)
```

```
    s.close()
```

```
    print("\n\nCharacters in file:", cnt)
```

```
except IOError as e:
```

```
    print("I/O error occurred: ", strerror(e.errno))
```

La routine è piuttosto semplice:

- utilizzare il meccanismo try-except e aprire il file con il nome predeterminato (text.txt nel nostro caso)
- provare a leggere il primo carattere dal file (ch = s.read(1))
- se il risultato è positivo (è dimostrato dal risultato positivo del controllo della condizione while), si emette il carattere (notare l'argomento end= - è importante! Non si vuole saltare a una nuova riga dopo ogni carattere);
- aggiornare anche il contatore (cnt);
- cerca di leggere il carattere successivo e il processo si ripete.

Se siete assolutamente sicuri che la lunghezza del file è sicura e potete leggere l'intero file in memoria in una sola volta, potete farlo: la funzione read(), invocata senza argomenti o con un argomento che valuta None, farà il lavoro per voi.

Ricordate che la **lettura di un file lungo un terabyte con questo metodo può danneggiare il sistema operativo**. Non aspettatevi miracoli: la memoria del computer non è estensibile.

Guardate il codice

```
try:
    cnt = 0
    s = open('text.txt', "rt")
    content = s.read()
    for ch in content:
        print(ch, end="")
        cnt += 1
    s.close()
    print("\n\nCharacters in file:", cnt)
except IOError as e:
    print("I/O error occurred: ", strerr(e.errno))
```

Analizziamolo:

- aprire il file come in precedenza;
- leggere il suo contenuto con un'unica invocazione della funzione read();
- Successivamente, elaboriamo il testo, iterandolo con un normale ciclo for e aggiornando il valore del contatore a ogni giro del ciclo;

Il risultato sarà esattamente lo stesso di prima.

### Elaborazione di file di testo: readline()

Se si vuole trattare il contenuto del file **come un insieme di righe** e non come un gruppo di caratteri, il metodo `readline()` aiuta a farlo. Il metodo tenta di **leggere una riga completa di testo dal file** e la restituisce come stringa in caso di successo. Altrimenti, restituisce una stringa vuota. Questo apre nuove opportunità: ora è possibile contare facilmente anche le righe, non solo i caratteri. Utilizziamolo. Guardate il codice

```
from os import strerror
try:
    ccnt = lcnt = 0
    s = open('text.txt', 'rt')
    line = s.readline()
    while line != "":
        lcnt += 1
        for ch in line:
            print(ch, end="")
            ccnt += 1
        line = s.readline()
    s.close()
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:    ", lcnt)
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

Come si può notare, l'idea generale è esattamente la stessa di entrambi gli esempi precedenti.



### **Elaborazione di file di testo: readlines()**

Un altro metodo, che tratta il file di testo come un insieme di righe e non di caratteri, è `readlines()`.

Il metodo `readlines()`, se invocato senza argomenti, cerca di **leggere tutto il contenuto del file e restituisce un elenco di stringhe, un elemento per ogni riga del file**.

Se non si è sicuri che la dimensione del file sia sufficientemente piccola e non si vuole testare il sistema operativo, si può convincere il metodo `readlines()` a leggere non più di un numero specifico di byte in una volta sola (il valore restituito rimane lo stesso: è un elenco di stringhe).

Provate a sperimentare il seguente esempio di codice per capire come funziona il metodo `readlines()`:

```
s = open("testo.txt")
print(s.readlines(20))
print(s.readlines(20))
print(s.readlines(20))
print(s.readlines(20))
s.close()
```

### **La dimensione massima accettata del buffer di input viene passata al metodo come argomento.**

Ci si può aspettare che `readlines()` sia in grado di elaborare il contenuto di un file in modo più efficace di `readline()`, in quanto potrebbe essere necessario invocarla meno volte.

Nota: quando non c'è nulla da leggere dal file, il metodo restituisce un elenco vuoto. Utilizzarlo per rilevare la fine del file.

Per quanto riguarda le dimensioni del buffer, si può prevedere che aumentarle possa migliorare le prestazioni dell'input, ma non esiste una regola d'oro: cercate di trovare da soli i valori ottimali.

Guardate il codice. Lo abbiamo modificato per mostrare come utilizzare readlines().

```
from os import strerror
```

```
try:
```

```
    ccnt = lcnt = 0
```

```
    s = open('text.txt', 'rt')
```

```
    lines = s.readlines(20)
```

```
    while len(lines) != 0:
```

```
        for line in lines:
```

```
            lcnt += 1
```

```
            for ch in line:
```

```
                print(ch, end="")
```

```
                ccnt += 1
```

```
    lines = s.readlines(10)
```

```
    s.close()
```

```
    print("\n\nCharacters in file:", ccnt)
```

```
    print("Lines in file:    ", lcnt)
```

```
except IOError as e:
```

```
    print("I/O error occurred:", strerror(e.errno))
```

Abbiamo deciso di utilizzare un buffer di 15 byte. Non è una raccomandazione.

Abbiamo usato questo valore per evitare che la prima invocazione di `readlines()` consumi l'intero file.

Vogliamo che il metodo sia costretto a lavorare di più e a dimostrare le sue capacità.

**Nel codice** sono presenti **due cicli annidati**: quello esterno utilizza il risultato di `readlines()` per iterare, mentre quello interno stampa le righe carattere per carattere.

### **Elaborazione di file di testo: continua**

L'ultimo esempio che vogliamo presentare mostra una caratteristica molto interessante dell'oggetto restituito dalla funzione `open()` in modalità testo.

**L'oggetto è un'istanza della classe iterable.**

Strano? Per niente. Utilizzabile? Sì, assolutamente.

Il **protocollo di iterazione definito per l'oggetto file** è molto semplice: il suo metodo `__next__` restituisce semplicemente **la riga successiva letta dal file**.

Inoltre, ci si può aspettare che l'oggetto invochi automaticamente `close()` quando una delle letture del file raggiunge la fine del file.

Guardate il codice e notate come il codice sia diventato semplice e chiaro.

```
from os import strerror
try:
    ccnt = lcnt = 0
    for line in open('text.txt', 'rt'):
        lcnt += 1
        for ch in line:
            print(ch, end="")
            ccnt += 1
    print("\n\nCharacters in file:", ccnt)
    print("Lines in file:    ", lcnt)
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

## Gestione dei file di testo: write()

La scrittura di file di testo sembra essere più semplice, poiché in effetti esiste un metodo che può essere utilizzato per eseguire tale operazione.

Il metodo si chiama write() e si aspetta un solo argomento: una stringa che verrà trasferita a un file aperto (non dimenticate che la modalità di apertura deve riflettere il modo in cui i dati vengono trasferiti: la **scrittura di un file aperto in modalità lettura non avrà successo**).

Non viene aggiunto alcun carattere di newline all'argomento di write(), quindi è necessario aggiungerlo da soli se si vuole che il file sia riempito con un certo numero di righe.

L'esempio

We encourage you to test the behavior of the write() method locally on your machine.

```
from os import strerror
```

```
try:
```

```
    fo = open('newtext.txt', 'wt') # A new file (newtext.txt) is created.
```

```
    for i in range(10):
```

```
        s = "line #" + str(i+1) + "\n"
```

```
        for ch in s:
```

```
            fo.write(ch)
```

```
    fo.close()
```

```
except IOError as e:
```

```
    print("I/O error occurred: ", strerror(e.errno))
```

mostra un codice molto semplice che crea un file chiamato newtext.txt (nota: la modalità aperta w assicura che **il file venga creato da zero**, anche se esiste e contiene dati) e poi vi inserisce dieci righe.

La stringa da registrare è composta dalla parola linea, seguita dal numero della linea. Abbiamo deciso di scrivere il contenuto della stringa carattere per carattere (questo viene fatto dal ciclo for interno), ma non siamo obbligati a farlo in questo modo.

Volevamo solo dimostrare che write() è in grado di operare su singoli caratteri.

Il codice crea un file con il seguente testo:

line #1

line #2

line #3

line #4

line #5

line #6

line #7

line #8

line #9

line #10

## Gestione dei file di testo: continua

Guardate l'esempio

```
from os import strerror
try:
    fo = open('newtext.txt', 'wt')
    for i in range(10):
        fo.write("line #" + str(i+1) + "\n")
    fo.close()
except IOError as e:
    print("I/O error occurred: ", strerror(e.errno))
```

Abbiamo modificato il codice precedente per scrivere righe intere nel file di testo.

Il contenuto del nuovo file creato è lo stesso.

Nota: si può usare lo stesso metodo per scrivere sul flusso stderr, ma non si deve cercare di aprirlo, perché è sempre aperto implicitamente.

Ad esempio, se si vuole inviare una stringa di messaggio a stderr per distinguerla dal normale output del programma, il suo aspetto può essere il seguente:

```
import sys
sys.stderr.write("Error message")
```

## Che cos'è un bytearray?

Prima di iniziare a parlare di file binari, dobbiamo parlare di una delle **classi specializzate che Python utilizza per memorizzare dati amorfi**.

**I dati amorfi sono dati che non hanno una forma specifica:** sono solo una serie di byte.

Ciò non significa che questi byte non possano avere un significato proprio o che non possano rappresentare un oggetto utile, ad esempio una grafica bitmap.

L'aspetto più importante è che nel luogo in cui entriamo in contatto con i dati non siamo in grado, o semplicemente non vogliamo, saperne nulla.

I dati amorfi non possono essere memorizzati con nessuno dei mezzi presentati in precedenza: non sono né stringhe né elenchi.

Dovrebbe esistere un contenitore speciale in grado di gestire tali dati.

Python ha più di un contenitore di questo tipo: uno di questi è **una classe specializzata chiamata bytearray** che, come suggerisce il nome, è **un array contenente byte (amorfi)**.

Se si desidera avere un contenitore di questo tipo, ad esempio per leggere un'immagine bitmap ed elaborarla in qualsiasi modo, è necessario crearlo esplicitamente, utilizzando uno dei costruttori disponibili.

Date un'occhiata:

```
dati = bytearray(10)
```

Questa invocazione crea un oggetto bytearray in grado di memorizzare dieci byte.

Nota: questo costruttore **riempie l'intero array di zeri**.



### **Bytearray: continua**

I bytearray assomigliano agli elenchi per molti aspetti. Ad esempio, sono **mutabili**, sono oggetto della funzione `len()` e si può accedere a qualsiasi elemento utilizzando l'indicizzazione convenzionale.

C'è una limitazione importante: **non si deve impostare alcun elemento della matrice di byte con un valore che non sia un intero** (la violazione di questa regola causerà un'eccezione `TypeError`) e **non si può assegnare un valore che non sia compreso nell'intervallo 0-255** (a meno che non si voglia provocare un'eccezione `ValueError`).

È possibile **trattare gli elementi di una matrice di byte come valori interi**, proprio come nell'esempio dell'editor.

Nota: abbiamo usato due metodi per iterare gli array di byte e abbiamo usato la funzione `hex()` per vedere gli elementi stampati come valori esadecimali.

Ora vi mostreremo **come scrivere un array di byte in un file binario** - binario, poiché non vogliamo salvare la sua rappresentazione leggibile - ma una copia uno a uno del contenuto della memoria fisica, byte per byte.

```
data = bytearray(10)
```

```
for i in range(len(data)):
    data[i] = 10 - i
```

```
for b in data:
    print(hex(b))
```

## **Bytearray: continua**

### **Scrittura su file**

```
from os import strerror  
data = bytearray(10)
```

```
for i in range(len(data)):  
    data[i] = 10 + i
```

```
try:
```

```
    bf = open('file.bin', 'wb')  
    bf.write(data)  
    bf.close()
```

```
except IOError as e:
```

```
    print("I/O error occurred:", strerror(e.errno))
```

```
# Your code that reads bytes from the stream should go here.
```

## Come leggere i byte da un flusso

La lettura da un file binario richiede l'uso di un metodo specializzato, chiamato `readinto()`, che non crea un nuovo oggetto array di byte, ma ne riempie uno precedentemente creato con i valori presi dal file binario.

Nota:

il metodo restituisce il numero di byte letti con successo;

il metodo cerca di riempire tutto lo spazio disponibile all'interno del suo argomento; se nel file ci sono più dati che spazio nell'argomento, l'operazione di lettura si fermerà prima della fine del file; altrimenti, il risultato del metodo può indicare che l'array di byte è stato riempito solo in modo frammentario (il risultato mostrerà anche questo, e la parte dell'array non utilizzata dal contenuto appena letto rimane intatta)

Guardate il codice completo qui sotto:

```
from os import strerror
data = bytearray(10)
try:
    bf = open('file.bin', 'rb')
    bf.readinto(data)
    bf.close()

    for b in data:
        print(hex(b), end=' ')
except IOError as e:
    print("I/O error occurred:", strerror(e.errno))
```

Analizziamolo:

Per prima cosa, apriamo il file (quello creato con il codice precedente) con la modalità descritta come rb; quindi, leggiamo il suo contenuto nell'array di byte chiamato dati, di dimensioni pari a dieci byte;

Infine, stampiamo il contenuto dell'array di byte: è uguale a quello che ci si aspettava?

Eeguire il codice e verificare se funziona

## Come leggere i byte da un flusso

Un modo alternativo di leggere il contenuto di un file binario è offerto dal metodo `read()`.

Invocato senza argomenti, tenta di **leggere tutti i contenuti del file nella memoria**, rendendoli parte di un nuovo oggetto della classe `bytes`.

Questa classe ha alcune somiglianze con `bytearray`, con l'eccezione di una differenza significativa: è **immutabile**.

Fortunatamente, non ci sono ostacoli alla creazione di un array di byte prendendo il suo valore iniziale direttamente dall'oggetto `bytes`, proprio come in questo caso:

```
from os import strerror
```

```
try:
```

```
    bf = open('file.bin', 'rb')
```

```
    data = bytearray(bf.read())
```

```
    bf.close()
```

```
    for b in data:
```

```
        print(hex(b), end=' ')
```

```
except IOError as e:
```

```
    print("I/O error occurred:", strerror(e.errno))
```

Fate attenzione: **non usate questo tipo di lettura se non siete sicuri che il contenuto del file sia adatto alla memoria disponibile.**

### **Come leggere i byte da un flusso: continua**

Se il metodo `read()` viene invocato con un argomento, questo **specifica il numero massimo di byte da leggere**. Il metodo tenta di leggere il numero di byte desiderato dal file e la lunghezza dell'oggetto restituito può essere utilizzata per determinare il numero di byte effettivamente letti.

È possibile utilizzare il metodo come qui:

```
try:
```

```
    bf = open('file.bin', 'rb')
    data = bytearray(bf.read(5))
    bf.close()
```

```
    for b in data:
        print(hex(b), end=' ')
```

```
except IOError as e:
```

```
    print("I/O error occurred:", strerror(e.errno))
```

Nota: i primi cinque byte del file sono stati letti dal codice, mentre i cinque successivi sono ancora in attesa di essere elaborati.

## **Copiare i file: uno strumento semplice e funzionale**

Ora dovrete amalgamare tutte queste nuove conoscenze, aggiungervi alcuni elementi nuovi e usarle per scrivere un vero codice in grado di copiare effettivamente il contenuto di un file.

Naturalmente, lo scopo non è quello di creare un sostituto migliore di comandi come *copy* (MS Windows) o *cp* (Unix/Linux), ma di vedere un possibile modo di creare uno strumento funzionante, anche se nessuno vuole usarlo.

Guardate il codice

```
from os import strerror
srcname = input("Enter the source file name: ")
try:
    src = open(srcname, 'rb')
except IOError as e:
    print("Cannot open the source file: ", strerror(e.errno))
    exit(e.errno)
dstname = input("Enter the destination file name: ")
```

```
try:
    dst = open(dstname, 'wb')
except Exception as e:
    print("Cannot create the destination file: ", strerror(e.errno))
    src.close()
    exit(e.errno)

buffer = bytearray(65536)
total = 0
try:
    readin = src.readinto(buffer)
    while readin > 0:
        written = dst.write(buffer[:readin])
        total += written
        readin = src.readinto(buffer)
except IOError as e:
    print("Cannot create the destination file: ", strerror(e.errno))
    exit(e.errno)

print(total, 'byte(s) succesfully written')
src.close()
dst.close()
```



- righe da 3 a 8: chiedono all'utente il nome del file da copiare e cercano di aprirlo per leggerlo; terminano l'esecuzione del programma se l'apertura fallisce; nota: usare la funzione `exit()` per interrompere l'esecuzione del programma e passare il codice di completamento al sistema operativo; qualsiasi codice di completamento diverso da 0 indica che il programma ha incontrato qualche problema; usare il valore `errno` per specificare la natura del problema;
- righe da 10 a 16: ripetono quasi la stessa azione, ma questa volta per il file di output;
- riga 18: preparare una porzione di memoria per il trasferimento dei dati dal file di origine a quello di destinazione; tale area di trasferimento è spesso chiamata buffer, da cui il nome della variabile; la dimensione del buffer è arbitraria - in questo caso, abbiamo deciso di usare 64 kilobyte; tecnicamente, un buffer più grande è più veloce nella copia degli elementi, poiché un buffer più grande significa meno operazioni di I/O; in realtà, c'è sempre un limite, il cui superamento non comporta ulteriori miglioramenti; testatelo voi stessi se volete.
- riga 19: conta i byte copiati - questo è il contatore e il suo valore iniziale;
- riga 21: prova a riempire il buffer per la prima volta;
- riga 22: finché si ottiene un numero di byte non nullo, ripetere le stesse azioni;
- riga 23: scrive il contenuto del buffer sul file di output (nota: abbiamo usato una `slice` per limitare il numero di byte da scrivere, dato che `write()` preferisce sempre scrivere l'intero buffer)
- riga 24: aggiornare il contatore;
- Riga 25: leggere il successivo chunk del file;
- righe da 30 a 32: un po' di pulizia finale - il lavoro è finito.

## LAB-43

### Scenario

Un file di testo contiene del testo (niente di strano), ma abbiamo bisogno di sapere quanto spesso (o quanto raramente) ogni lettera appare nel testo. Un'analisi di questo tipo può essere utile in crittografia, quindi vogliamo essere in grado di farlo in riferimento all'alfabeto latino.

Il vostro compito è quello di scrivere un programma che:

- chiede all'utente il nome del file di input;
- legge il file (se possibile) e conta tutte le lettere latine (le lettere minuscole e maiuscole sono considerate uguali)
- stampa un semplice istogramma in ordine alfabetico (devono essere presentati solo i conteggi non nulli)

Creare un file di prova per il codice e verificare se l'istogramma contiene risultati validi.

Supponendo che il file di test contenga solo una riga riempita con:

aBc

il risultato atteso dovrebbe essere il seguente:

a -> 1

b -> 1

c -> 1

**Suggerimento:** Riteniamo che un dizionario sia un mezzo di raccolta dati perfetto per memorizzare i conteggi. Le lettere possono essere chiavi, mentre i contatori possono essere valori.

## LAB-44

### Scenario

Il codice precedente deve essere migliorato. Va bene, ma deve essere migliorato.

Il vostro compito è quello di apportare alcune modifiche che generano i seguenti risultati:

l'istogramma di uscita sarà ordinato in base alla frequenza dei caratteri (il contatore più grande dovrebbe essere presentato per primo)

l'istogramma deve essere inviato a un file con lo stesso nome di quello di input, ma con il suffisso '.hist' (deve essere concatenato al nome originale)

Supponendo che il file di input contenga una sola riga riempita con:

cBabAa

il risultato atteso dovrebbe essere il seguente:

a -> 3

b -> 2

c -> 1

**Suggerimento:** utilizzare un lambda per modificare l'ordine di ordinamento.

## LAB-45

### Scenario

Il Prof. Jekyll tiene lezioni con gli studenti e prende regolarmente appunti in un file di testo. Ogni riga del file contiene tre elementi: il nome dello studente, il suo cognome e il numero di punti che lo studente ha ricevuto durante alcune lezioni.

Gli elementi sono separati da spazi bianchi. Ogni studente può comparire più di una volta all'interno del file del Prof. Jekyll.

Il file può avere il seguente aspetto:

John Smith

Anna ,5

John Smith

Anna 11

Andrew Cox ,5

Il vostro compito è quello di scrivere un programma che:

chiede all'utente il nome del file del Prof. Jekyll;

legge il contenuto del file e conta la somma dei punti ricevuti per ogni studente;

stampa un rapporto semplice (ma ordinato), come questo:

Andrew Cox 1,5

Anna Bolena 15,5

John Smith 7.0

Nota:

Il programma deve essere completamente protetto contro tutti i possibili fallimenti: l'inesistenza del file, il suo vuoto o qualsiasi errore dei dati di input; l'incontro con qualsiasi errore dei dati deve causare l'immediata terminazione del programma e l'errore deve essere presentato all'utente; implementare e utilizzare la propria gerarchia di eccezioni. La seconda eccezione dovrebbe essere sollevata quando viene rilevata una riga errata e la terza quando il file sorgente esiste ma è vuoto.

**Suggerimento:** utilizzare un dizionario per memorizzare i dati degli studenti.

```
class StudentsDataException(Exception):  
    pass
```

```
class BadLine(StudentsDataException):  
    # Write your code here.
```

```
class FileEmpty(StudentsDataException):  
    # Write your code here.
```

## Punti di forza

1. Per leggere il contenuto di un file, è possibile utilizzare i seguenti metodi di flusso:

- `read(number)` - legge il numero di caratteri/byte dal file e li restituisce come stringa; è in grado di leggere l'intero file in una sola volta;
- `readline()` - legge una singola riga dal file di testo;
- `readlines(number)` - legge il numero di righe dal file di testo; è in grado di leggere tutte le righe contemporaneamente;
- `readinto(bytearray)` - legge i byte dal file e li riempie con il bytearray;

2. Per scrivere nuovo contenuto in un file, è possibile utilizzare i seguenti metodi di flusso:

- `write(stringa)` - scrive una stringa in un file di testo;
- `write(bytearray)` - scrive tutti i byte di bytearray in un file;

3. Il metodo `open()` restituisce un oggetto iterabile che può essere usato per iterare tutte le righe del file all'interno di un ciclo `for`. Ad esempio:

```
for line in open("file", "rt"):
    print(line, end="")
```

Il codice copia il contenuto del file nella console, riga per riga. **Nota:** lo stream si chiude **automaticamente** quando raggiunge la fine del file

### Esercizio 1

Cosa ci si aspetta dal metodo `readlines()` quando lo stream è associato a un file vuoto?

Soluzione

Un elenco vuoto (un elenco di lunghezza zero).

### Esercizio 2

Qual è l'obiettivo del codice seguente?

```
for line in open("file", "rt"):
    for char in line:
        if char.lower() not in "aeiouy ":
            print(char, end="")
```

Soluzione

Copia il contenuto del *file* nella console, ignorando tutte le vocali.

### Esercizio 3

Si sta per elaborare una bitmap memorizzata in un file chiamato `image.png` e si vuole leggere il suo contenuto nella sua interezza in una variabile *bytearray* chiamata `image`. Aggiungete una riga al codice seguente per raggiungere questo obiettivo.

```
try:
    stream = open("image.png", "rb")
    # Insert a line here.
    stream.close()
except IOError:
    print("failed")
else:
    print("success")
```

Risultato

```
image = bytearray(stream.read())
```