DISPENSA DI ELEMENTI DI INFORMATICA E PROGRAMMAZIONE

CHE COS'È UN COMPUTER?

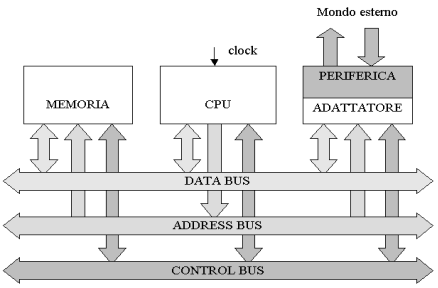
Un sistema di elaborazione è composto da elementi fisici (materiali) e logici (informazioni):

- hardware: parte fisica (elettronica, magnetica, ottica…);
- software: parte logica (materiale intangibile, ma indispensabile, come dati e programmi). I dati rappresentano qualunque tipo di informazione (numeri, testi, immagini, suoni, filmati…). I programmi sono formati da insiemi di istruzioni (elementari) che vengono eseguite in ordine.

L'architettura di un computer

Per capire i meccanismi di base della programmazione è necessario conoscere gli elementi hardware che costituiscono un computer.

Secondo il modello di Von Neumann, elaborato nel 1946, ancora prima della progettazione dei calcolatori; l'architettura è composta da quattro blocchi comunicanti tra loro per mezzo di un bus, un canale di scambio di informazioni. Il concetto di memoria è molto importante all'interno del calcolatore in quanto serve ad immagazzinare dati e programmi all'interno del computer. E' suddivisa in celle o locazioni di memoria, ognuna delle quali è identificata univocamente da un indirizzo. Ogni cella contiene un numero predefinito di bit, solitamente 32 o 64. Un bit ("Binary Digit") è un dato elementare, l'unità elementare di informazione, che può assumere due valori, convenzionalmente zero e uno. Un insieme di otto bit si chiama byte ed è l'unità di misura della memoria (si utilizzano anche multipli come i MEGAbyte).



Quindi: la CPU, la memoria primaria ed i circuiti elettronici che controllano il disco rigido e altri dispositivi periferici sono interconnessi mediante un insieme di linee elettriche che formano un bus. I dati transitano lungo il bus, passano dalla memoria e dai dispositivi periferici verso la CPU e viceversa. All'interno del PC si trova la scheda madre (mother-board), che contiene la CPU, la memoria primaria, il bus e gli alloggiamenti di espansione per il controllo delle periferiche.

L'unità centrale di elaborazione

Dal punto di vista logico, la CPU (central processing unit) è costituita da tre parti principali. Secondo il modello di Von Neumann la CPU è una parte di uno dei quattro blocchi collegati al bus. Il suo funzionamento è tipicamente ciclico e il periodo viene scandito dall'orologio di sistema, la cui frequenza costituisce una delle caratteristiche tecniche più importanti della CPU:

- l'unità logico-aritmetica (ALU, arithmetic-logic unit) effettua elaborazioni aritmetiche e logiche (si trova all'interno del blocco CPU);
- l'unità di controllo (CU, control unit) governa il funzionamento della CPU stessa;
- un insieme di registri, sono celle di accesso molto veloci per la memorizzazione temporanea di dati.

Inoltre, l'unità centrale di elaborazione reperisce i dati dalla memoria esterna e da altri dispositivi periferici e ve li rispedisce dopo averli elaborati. E' costituita da uno o più chip (microprocessori). Un chip (circuito integrato) è un componente elettronico con connettori metallici esterni detti pin e collegamenti esterni detti wire; costituito principalmente di silicio e alloggiato in un contenitore plastico o ceramico detto package. Tipicamente in un chip sono presenti alcuni miliardi di transitori collegati tra loro in maniera molto complicata (con dimensione nell'ordine nei nm). Nella CPU, il contatore di programma (program counter) contiene l'indirizzo dell'istruzione da eseguire.

Ciclo di funzionamento della CPU

Ogni ciclo di funzionamento è composto da tre fasi, ed e' detto ciclo fetch-decode-execute:

1. accesso: lettura dell'istruzione da eseguire e sua memorizzazione nel registro istruzione;
2. decodifica: decodifica dell'istruzione da eseguire;
3. esecuzione: esecuzione dell'istruzione.

La posizione dell'istruzione a cui si accede durante la fase di fetch è contenuta nel contenitore di programma. Questo viene automaticamente incrementato di un'unità ad ogni ciclo, in modo da eseguire istruzioni in sequenza, memorizzate in celle di memoria aventi indirizzi consecutivi.

La memoria primaria

La memoria primaria è veloce ma costosa (5€/GByte). E' costituita da chip di memoria realizzati con la stessa tecnologia (al silicio) utilizzata per la CPU ed è suddivisa in due parti:

- memoria di sola lettura ROM (Read-Only Memory) che contiene il programma utilizzato dal PC quando si accende. La ROM è una memoria non volatile che contiene i programmi necessari all'avvio del computer, che devono essere sempre disponibili. Nel PC tali programmi prendono il nome di BIOS (Basic Input/Output System). Una memoria ROM può essere scritta una sola volta, al momento della sua fabbricazione. In realtà, il BIOS è aggiornabile perché risiede in una EPROM (o tecnologia similare), memoria riscrivibile (lentamente e poche volte);
- memoria ad accesso casuale RAM (Random Access Memory). Con accesso casuale si intende che il tempo per accedere ad un dato non dipende dalla sua posizione nella memoria. Ciò distingue la RAM dalla ROM è che la prima è anche scrivibile, oltre che leggibile. Il tempo di accesso della RAM è estremamente piccolo e nell'ordine di grandezza della CPU. La memoria RAM contiene dati in fase di modifica e programmi che non devono essere sempre disponibili. Infatti, perde il proprio contenuto quando si spegne il computer e, per questo, è detto supporto volatile.

La memoria secondaria

La memoria secondaria, detta di massa, è solitamente un disco rigido (o disco fisso, hard disk) ed è un supporto non volatile e meno costoso della memoria primaria (circa 100 volte, come prezzo per byte) ma decisamente più lento (tempo di accesso di qualche ms anziché qualche ns come RAM o CPU). Il costo è di circa 0,05€/GByte. Programmi e dati risiedono sul disco rigido e vengono caricati nella RAM quando necessario, per poi tornarvi aggiornati se e quando necessario. Un disco rigido è formato da piatti rotanti rivestiti di materiale magnetico, con testine di lettura/scrittura. Il processo è simile a quello dei vecchi nastri audio o video. La memoria di massa mantiene l'informazione anche quando il contatore è spento. Sono (meno) usati anche altri tipi di memoria secondaria a tecnologia magnetica:

- floppy disk (dischetto flessibile), di capacità limitata ma con il vantaggio di essere agevolmente rimosso dal sistema ed essere trasferito ad un altro sistema (dispositivo di memoria esterno);
- tape (nastri per dati), di capacità elevatissima, molto economici, ma molto lenti, perché l'accesso ai dati è sequenziale anziché casuale (bisogna avvolgere o svolgere un nastro invece che spostare la testina di lettura sulla superficie di un disco (perfetti per attività di backup di grandi quantità di dati).

Sono molto diffusi anche altri tipi di memoria secondaria a tecnologia ottica come:

- CD-ROM (Compact Disc Read-Only Memory), viene letto da un dispositivo laser, esattamente come un CD audio; ha un'elevata capacità ed è molto economico e affidabile; è un supporto di sola lettura, utilizzato per distribuire programmi e informazioni;

- CR-R (Compact Disc Recordable), utilizza una tecnologia simile al CD-ROM ma può essere scritto dall'utente una sola volta (più volte se CD-RW);
- lettore di CD;
- DVD, ha rappresentato la nuova frontiera per questa tecnologia, con elevatissima capacità. Nel 2002 con l'introduzione del Blue-Ray si è raggiunta una capacità ancora più elevata, tuttavia ad un prezzo costoso.

Sono infine diffusi anche molti tipi di memoria secondaria a tecnologia microelettronica, come le chiavette USB. Queste usano una tecnologia molto simile a quella EPROM usata per il BIOS del PC. Hanno un costo di circa 0,15€/GByte, intermedio tra RAM e disco rigido. Non è volatile, tuttavia è portatile. Sono più lente dei dischi rigidi ma più veloci dei CD.

Il bus nel modello di Von Neumann

Il bus è, in realtà, costituito da tre bus distinti:

- bus dei dati: lo scambio dei dati è bidirezionale (da e verso bus e altri scomparti);
- bus degli indirizzi (segnali di controllo provenienti solo dalla CPU);
- bus dei segnali di controllo (segnali di controllo provenienti solo dalla CPU).

L'interazione fra l'utente umano ed il computer avviene mediante i cosiddetti dispositivi periferici di input (verso la CPU) e output. Tipici dispositivi di input sono la tastiera, il mouse, il microfono e lo scanner; mentre dispositivi di output sono lo schermo, le stampanti e gli altoparlanti. Esistono anche dispositivi di Input/Output bidirezionale come la connessione di rete ed il touchscreen.

DEFINIZIONE DI PROGRAMMA, PROGRAMMAZIONE E ALGORITMO

Ogni programma svolge una diversa funzione, anche complessa. Un computer è quindi una macchina che memorizza dati, interagisce con dispositivi ed esegue programmi. I programmi sono sequenze di istruzioni che il computer esegue e di decisioni che il computer prende per svolgere una certa attività. Nonostante i programmi siano molto sofisticati, le istruzioni di cui sono composti sono molto elementari e possono essere imperative (come la somma di due numeri) o istruzioni condizionali (che prendono decisioni). Il tutto avviene ad una velocità altissima che garantisce l'illusione di un'interazione fluida. Il computer è una macchina estremamente versatile.

La programmazione è l'attività di progettare e realizzare un programma (usare un computer non richiede alcuna attività di programmazione!). Un computer può risolvere soltanto problemi che potrebbero essere risolti manualmente in modo univoco, ovvero per i quali sia noto l'algoritmo.

Si dice algoritmo la descrizione di un metodo di soluzione di un problema che sia eseguibile, priva di ambiguità ed arrivi a conclusione in un tempo finito. Un algoritmo può essere descritto tramite diversi tipi di linguaggio (naturale, matematico, di programmazione…). Esiste anche un linguaggio detto "pseudocodice" utilizzato tipicamente nella comunicazione tra programmatori (sorta di astrazione dei vari linguaggi di programmazione). Dopo aver individuato un algoritmo, per renderlo eseguibile mediante un calcolatore bisogna scriverlo in un programma, cioè tradurlo in un linguaggio di programmazione che sia comprensibile al calcolatore.

LINGUAGGI DI PROGRAMMAZIONE PYTHON

Per descrivere al computer la metodologia di soluzione di un problema, cioè per scrivere un algoritmo comprensibile, è necessario un linguaggio di programmazione. Al giorno d'oggi sono stati progettati molti diversi linguaggi, tra i più diffusi troviamo Python, Java e C++. Tutti questi sono significativamente simili tra loro. Talvolta è necessario scrivere codici anche in linguaggi

ormai obsoleti, per attività di manutenzione di programmi esistenti (come COBOL, soprattutto per programmi bancari e finanziari).
Il linguaggio di programmazione Python è stato progettato nei primi anni Novanta da Guido Van Rossum. L'obiettivo era quello di avere una sintassi più semplice e chiara di quella degli altri linguaggi di programmazione diffusi.

Scrittura di un programma Python

La scrittura di un programma Python prevede alcune alternative:

- usare un ambiente di sviluppo integrato (IDE) come Idle, specifico per Python o Eclipse;
- usare un editor di testo come notepad in Windows o nedit/gedit in Linux. Non si parla di un impaginatore di testi (es. Microsoft Word) perchè i word processor potrebbero apportare modifiche potenzialmente "dannose" ai fini della programmazione.

In Python ciascun carattere occupa lo stesso spazio orizzontale (carattere monospaziato), maiuscole e minuscole sono considerate distinte. Un programma Python è costituito da istruzioni o enunciati, che verranno tradotti in linguaggio macchina dell'interprete Python, per poi essere eseguito dalla CPU:

- con il termine "run" si intende "esegui programma". "`>>>`" sulla sinistra dello schermo indica che l'esecuzione del programma è terminata;
- le righe con il carattere # sono commenti, quindi non verranno considerata durante l'esecuzione del programma: non essendo destinate all'interprete, i commenti possono essere in italiano, mentre le istruzioni in Python sono in inglese. Quando vogliamo scrivere commenti "lunghi", che si estendano su più righe, è comodo usare una sintassi alternativa: e' sufficiente porre " " " all'inizio e alla fine del commento (affinché sia valido è necessario andare a capo, ovvero hanno una riga "personalizzata/propria");
- le virgolette servono a "delimitare" il testo da stampare con la funzione print. Questo tipo di enunciato si chiama invocazione di funzione, perché print è una funzione di Python a cui trasferiamo informazioni affinché le elabori in qualche modo (l'elaborazione svolta è la visualizzazione delle informazioni ricevute).

Indentazione

In Python, l'indentazione è significativa, infatti indentare in modo incorretto può portare a comportamenti sbagliati del programma o a errori.

```
print("eseguito sempre all'inizio")
if condizione:
    print('eseguito in mezzo solo se la condizione è vera')
    print('eseguito in mezzo solo se la condizione è vera')
    print('eseguito in mezzo solo se la condizione è vera')
print('eseguito sempre alla fine')
```

In questo esempio possiamo vedere che:

- il primo print, l'if, e l'ultimo print hanno la stessa indentazione, e vengono eseguiti sempre;
- dopo l'if c'è un blocco di codice con indentazione maggiore, che include tre print;
- se la condizione dell'if è vera, i tre print vengono eseguiti;
- se la condizione dell'if è falsa, i tre print non vengono eseguiti.

In un programma Python tutti gli enunciati devono essere "incollati a sinistra", cioè devono tutti iniziare nella prima colonna, all'inizio della riga del testo nel file sorgente. Se ci sbagliamo, appare un IndentationError (ovvero un errore di impaginazione errato).

Errori di programmazione

L'attività di programmazione, come ogni altra attività di progettazione umana, è soggetta a errori:

- errori di sintassi: l'interprete riesce agevolmente ad individuare e segnalare l'errore di sintassi, perché identifica un nome che non conosce. L'errore viene segnalato indicando il nome del file contenente l'errore, il numero della riga contenente l'errore e il tipo di errore;
- errori in esecuzione: durante l'esecuzione si verifica una situazione "eccezionale";
- errori logici: l'eliminazione di errori logici richiede molta pazienza (il programma viene eseguito senza segnalazioni d'errore, ma non fa quello che dovrebbe fare).

FUNZIONI IN PYTHON

Le funzioni sono uno strumento che ci permette di raggruppare un insieme di istruzioni che eseguono un compito specifico: accettano in input 0 o più argomenti (o parametri), li elaborano, e restituiscono in output un risultato. Una volta definita una funzione, è possibile eseguirla, passando argomenti diversi a seconda della situazione. Questo ci permette di rendere il codice più ordinato ed evitare ripetizioni.

Funzioni predefinite in Python

Le funzioni predefinite nel linguaggio Python (come print, input, float, int, …) costituiscono in pratica un ulteriore modulo della libreria standard (builtins). Tutto funziona come se tale intero modulo fosse importato implicitamente all'inizio di qualsiasi programma.

Le funzioni non sono tutte predefinite perché all'aumentare delle funzioni note all'interprete Python, aumenta il tempo necessario alla traduzione del codice dei programmi (l'interprete deve sfogliare "dizionari" più voluminosi). Importare moduli e risorse non utilizzate non è un errore, ma è meglio non esagerare. Inoltre, avere un unico modulo (cioè nessun modulo…) porrebbe problemi per lo "spazio dei nomi" (namespace): non possono esserci due funzioni con lo stesso nome nello stesso modulo, ma possono esserci due funzioni con lo stesso nome in due moduli diversi. Nel caso di built-in-function (come print) interessa soltanto sapere:

- il nome (per invocarla);
- quali informazioni si aspetta di ricevere, ovvero i cosiddetti argomenti o parametri della funzioni, scritti all'interno delle parentesi tonde che seguono il nome della funzione;
- cosa fa (non importa il "come"), è il classico modello ingegneristico "a scatola nera".

E' molto importante che le funzioni siano versatili. In informatica, una sequenza di caratteri racchiusa tra "virgolette" viene chiamata stringa. In Python, una stringa può essere alternativamente delimitata da "virgolette" o da 'apici'. Un programma può essere costituito da più enunciati, che vengono eseguiti in presenza, dall'alto verso il basso, seguendo l'ordine su cui sono stati scritti dal programmatore nel file. Il file in cui scriviamo il programma è detto file sorgente, perché è la sorgente di informazioni per il processo di esecuzione del programma.

Funzione print() e str()

L'invocazione print(""), stampa una riga vuota (ma non è un errore!). Gli spazi contenuti all'interno delle stringhe sono significativi. Anche l'impaginazione delle informazioni visualizzate da un programma è un aspetto molto rilevante. Scrivendo più stringhe come argomento di print, una dopo l'altra, separate da una virgola, posta fuori dalle virgolette, viene stampata un'unica riga che contiene le stringhe, separate tra loro da uno spazio aggiunto automaticamente.

La funzione print non è limitata alla visualizzazione di stringhe, ma può anche visualizzare il risultato di operazioni aritmetiche. Quindi, gli argomenti di print possono essere espressioni

aritmetiche, delle quali viene visualizzato il risultato. Se voglio utilizzare il risultato di un'operazione aritmetica però, lo devo porre fuori dalle virgolette.

```
n3 = n1 + n2, print("La somma tra i due numeri è: ", n3)
```

Ricordiamo che print stampa ogni tipo di oggetto. Tuttavia, se necessario possiamo convertire gli oggetti in stringhe con str(). Ma come fa str() a capire come convertire in stringa oggetti di tipi diversi? Questo compito è delegato al progettista: in particolare, se nella classe un oggetto è definito il metodo __str__ , str() lo invocherà e userà la stringa invocata da tale metodo. Altrimenti str() stamperà una stringa che contiene informazioni (note all'interprete) sul tipo di oggetto e sul suo indirizzo in memoria.

```
class Student:
   . . .
   def __str__(self) :
       return str(self.grades) + \
              "Avg. =" + str(self.getAvgGrade())
```

Funzione print senza andare a capo

Ogni invocazione di print visualizza una riga di testo, per visualizzare i propri argomenti e, poi, va a capo. A volte, però, potremmo aver bisogno di visualizzare una porzione di riga, senza andare a capo, per completare la riga in seguito. Questo effetto si può ottenere aggiungendo, nell'invocazione print, un ulteriore argomento conclusivo, del tipo argomento con nome.

```
print('Hello, ', end='')
print('World!') #visualizza una riga unica
```

L'argomento end (che, se presente, deve essere l'ultimo), a cui diamo un valore con una assegnazione, contiene una stringa che viene visualizzata dopo tutti i caratteri visualizzati da print, invece di andare a capo.

Esempio. Data una stringa vogliamo visualizzare soltanto i suoi caratteri di indice dispari, uno di seguito all'altro su un'unica riga, senza spazi interposti. Esistono due alternative.

```
> Alternativa 1
s = "una stringa"
x = ""
i = 0
while i < len(s) :
     if i % 2 == 1 :
     x += s[i]
     i += 1
print(x)

> Alternativa 2
s = "una stringa"
i = 0
while i < len(s) :
   if i % 2 == 1 :
     print(s[i], end= "") #si considera anche lo spazio come carattere
   i += 1
print()#visualizza "n tig"
```

`Funzione input() e int(): variabili ed acquisizione di dati`

Per definire variabili in Python, è sufficiente utilizzare l'operatore di assegnazione (=). Per acquisire dati dall'utente viene introdotta la funzione `input`: questa funzione visualizza la stringa fornita come argomento e poi sospende l'esecuzione del programma, in attesa che l'utente scriva una stringa, terminando con il tasto Invio/Enter. La stringa acquisita deve essere memorizzata da qualche parte nella RAM, per essere utilizzata dai successivi enunciati del programma.

Nella programmazione avremo spesso bisogno di "zone di memoria" in cui archiviare informazioni temporanee, durante l'esecuzione del programma. Una zona di memoria utilizzata per conservare informazioni temporanee viene chiamata variabile. Un nome di variabile può essere composto da lettere, cifre e underscore, ma deve iniziare sempre con una lettera. Lo stesso nome verrà utilizzato per "scrivere" informazioni nella variabile e per "leggere" tali informazioni in seguito. La scrittura di informazioni è sempre distruttiva mentre la lettura non lo è.

Per scrivere il valore di una variabile usiamo l'istruzione o l'enunciato di assegnazione. In un'istruzione di assegnazione, a destra del segno uguale ci deve essere il valore che vogliamo assegnare alla variabile nominata a sinistra. Utilizzare il valore di una variabile significa leggere l'informazione che vi è stata memorizzata ed inserire tale valore all'interno di un'istruzione. Nel caso in cui la variabile sia stata modificata più volte, si legge il valore più recente (ogni scrittura cancella qualsiasi tipo di traccia di quella precedente).

Le espressioni aritmetiche possono contenere variabile, oltre ai numeri, che si chiamano tecnicamente "valori numerici letterali"

```
ageString = input("Quanti hanni hai?")
age = int(ageString)
print("Fra due anni avrai", age + 2, "anni")
```

Utilizzando int(), abbiamo trasformato la stringa, contenuta nella variabile ageString in un numero intero corrispondente (poi memorizzando nella variabile age) invocando la funzione int.

`Funzione float()`

La funzione float, predefinita in Python, vuole come argomento una stringa i cui caratteri descrivano un numero intero oppure un numero "con la virgola" e restituisce il numero corrispondente sotto forma di numero "con la virgola" ("virgola zero" se è intero, es. 3.0).

In informatica i numeri con la virgola rappresentano un sottoinsieme finito dei numeri reali. Elaborare in modo specifico i numeri interi fa risparmiare tempo durante l'esecuzione del programma (il calcolo 2+3 è più veloce del calcolo 2.0 + 3.0).

```
age = int(input("Quanti anni hai? "))
money = float(input("Quanti dollari hai? "))
```

`Funzione random() e randint()`

Python mette a disposizione la funzione random() per generare sequenze di numeri pseudo casuali. Ogni invocazione della funzione restituisce un numero frazionario pseudo-casuale nell'intervallo. Una funzione simile è randint (), che svolge esattamente lo stesso compito, restituendo un numero intero. Per utilizzare le funzioni random e randint nei nostri programmi dobbiamo importarli all'inizio del programma dal modulo random

```
from random import random
from random import randint
# oppure from random import random, randint
```

```
Esempio:  "Lancio di un dado”
from random import randint
trials = int(input("Quanti lanci? "))
sum = 0
count = 0
while count <  trials :
   d = randint(1,6)
   sum += d
   count += 1
avg = sum / trials
print ("Average: ",  avg)
```

Eseguendo il programma più volte, si ottiene un diverso valore medio ogni volta, a conferma che i numeri casuali generati variano a ogni esecuzione, come dev'essere. Curiosità: si osserva che il valore medio che si ottiene è sempre piuttosto vicino a 3.5. La "legge dei grandi numeri" (o Teorema di Bernoulli) afferma che il valore 3.5 viene raggiunto esattamente con un numero infinito di lanci del dado. Praticamente, è sufficiente un numero di lanci di qualche milione…

`Funzione len()`

Una stringa è una sequenza di caratteri racchiusi tra virgolette o singoli apici che non fanno parte della stringa. Talvolta sarà utile sapere quanto è lunga una stringa, cioè quanti sono i caratteri che la compongono. Questa informazione viene calcolata e restituita dalla funzione predefinita len (parte iniziale della parola length), che riceve una stringa come argomento.

```
emptyString = ''
print(len(emptyString))
```
il risultato ottenuto è 0

`Funzione ord() e chr(): Python e unicode`

Python mette a disposizione due funzioni predefinite che riguardano la codifica dei caratteri:

- la funzione ord riceve un carattere (cioè una stringa di lunghezza unitaria) e restituisce il numero intero non negativo che lo rappresenta nella codifica Unicode);
- la funzione chr, al contrario, riceve un numero intero non negativo e restituisce il carattere (cioè una stringa di lunghezza unitaria) che corrisponde a tale codice nello standard Unicode.

```
print(chr(3+ord("a")))
```
#visualizza d

`Funzione main()`

L'invocazione di main () contiene il programma principale, utilizzata solitamente nel collaudo di nuovi moduli. La possibilità di definire funzioni è uno strumento che, in generale, può essere di grande aiuto nella scomposizione di un problema complesso in problemi più semplici.
La definizione di funzioni invocate più volte in un programma consente di concentrare in un unico punto le modifiche da apportare alla porzione di algoritmo implementata da ciascuna funzione, assegnando anche un nome esplicativo alla funzionalità svolta da una sezione di codice.

```
#script_a
def main():
    print("Riga eseguita se lo script è stato lanciato come
programma")

def print_tre_volte(parola):
    print(parola)
```

```
    print(parola)
    print(parola)

if __name__ == "__main__":
    main()
```

L'if statement (clausola di controllo) controlla e compara __name__ alla stringa "__main__". Quando l'if ritorna True, Python esegue main(). Questa funzione permette che il codice abbia un doppio utilizzo, sia come programma, che come modulo. Se io ho un file Python denominato script_a ed eseguo il codice scritto sopra, visualizzerò la prima riga di print (ovvero la funzione viene eseguita come programma). Se prendo un altro file, nella medesima cartella di script_a, chiamato script_b e importo la funzione, allora visualizzerò i tre print successivi

```
#script_b
import script_a
print(script_a.__name__) #script_a (name associato al modulo)
script_a.print_tre_volte("abc") #abc abc abc (su tre righe separate)
```

`Funzione type()`

Per sapere di che tipo è un oggetto si può usare la funzione predefinita type, che restituisce il tipo dell'oggetto ricevuto come argomento: utile per il debugging o anche per altro. La funzione type si può usare anche per verificare che un valore sia del tipo corretto. Si confronta il valore restituito da type con il nome del tipo di dato.

```
print(type(range(5))) # <class 'range'>
print(type(2))        # <class 'int'>
print(type((2, )))    # <class 'tuple'>
x = "pippo" # anche con variabili
print(type(x))        # <class 'str'>
```

`Funzione exit()`

Durante la fase iniziale di un programma, quando si verifica la validità dei dati acquisiti in input, spesso si vorrebbe terminare il programma prematuramente (in caso di dati errati). Questo è possibile con la funzione exit del modulo sys, la cui invocazione provoca la terminazione immediata del programma, dopo aver visualizzato il messaggio fornito come argomento.

```
from sys import exit
n = int(input("Inserisci un numero positivo: "))
if n < 0:
     exit("Errore grave")
```

`Funzione help()`

Usando la shell di Python (anche all'interno di Idle) può essere comoda la funzione help() che visualizza la documentazione sintetica di una funzione. Per ottenere informazioni su un modulo bisogna prima importarlo.

```
>>> help(abs)
Help on built-in function abs in module builtints:
abs(x, /)
    Return the absolute value of the argument
```

Valore restituito da una funzione

Prima di valutare un'espressione, vengono eseguite tutte le invocazioni di funzioni che eventualmente contiene. Tuttavia, non sempre l'invocazione di una funzione restituisce un valore: in tal caso, l'invocazione della funzione non deve comparire all'interno di un'espressione, bensì solamente dal punto di vista sintattico in modi equivalenti.

```
ageString = input("Quanti anni hai?")
x = 3 + int(ageString)
è equivalente a x = 3 + int(input("Quanti anni hai?"))
```

La seconda istruzione dell'esempio contiene un'invocazione annidata, spesso utilizzate per accorciare la lunghezza dell'istruzione.

La parola chiave return viene usata per restituire un valore al chiamante, che può assegnarlo a una variabile o utilizzarlo per altre operazioni:

```
def square(n) :
    return n**2

x = square(5)
print(x) #25
```

Una funzione può avere 0 o più return, e una volta che un return viene eseguito, la funzione termina immediatamente. Questo vuol dire che solo uno dei return viene eseguito:

```
def abs(n):
    if n < 0:
        return -n # eseguito se n è negativo
    else:
        return n # eseguito se n è positivo

print(abs(-5)) #5
print(abs(5)) #5
```

Return è in genere seguito dal valore di ritorno, ma è anche possibile omettere il valore e usare return per terminare la funzione: in questo caso None viene ritornato automaticamente.

```
def print_twice(text):
    print(text)
    print(text) # ritorna None al termine della funzione
    if not text: # termina immediatamente se text è una stringa vuota
        return

res = print_twice('Python') #Python Python (su due righe)
print(res) #None
res = print_twice('') # visualizza una riga vuota
print(res) #None
```

Memorizzazione di risultati intermedi

La memorizzazione di risultati intermedi risulta utile quando l'intera espressione senza risultati intermedi sarebbe difficile di comprensione. Un'altra utilità del visualizzare risultati intermedi è, durante il collaudo, facilitare un'eventuale fase di diagnosi d'errore. Ad esempio, se i primi risultati intermedi visualizzati sono corretti, l'errore si troverà nella sezione di programma successiva. Nel

caso in cui dovesse sorgere un print(“DEBUG,..), invece di eliminare gli enunciati, è utile commentare utilizzando # all’inizio della stringa, lasciandolo quindi nel codice. In questo modo, sarà semplice ripristinarlo, se necessario.

DEFINIZIONE DI FUNZIONI IN PYTHON

La prima riga della definizione di una funzione, contenente la parola chiave def, viene solitamente chiamata firma o intestazione (header) della funzione stessa. Il corpo della funzione, che definisce le istruzioni che verranno eseguite ogni volta che la funzione verrà invocata, deve essere indentato, come ogni corpo: le variabili parametro assumeranno i valori forniti come argomenti durante ciascuna invocazione della funzione.

```
def is_even(n):
    if n%2 == 0:
       return True
    else:
       return False

print(is_even(2)) #True
```

È anche possibile documentare una funzione usando una docstring, cioè una stringa (in genere racchiusa tra """...""") che si trova come prima istruzione all'interno di una funzione:

```
def is_even(n):
    """Return True if n is even, False otherwise."""
    #codice come sopra
```

Passaggio di argomenti

Quando una funzione viene chiamata, è possibile passare 0 o più argomenti. Questi argomenti possono essere passati per posizione o per nome:

```
def calc_rect_area(width, height):
    """Return the area of the rectangle."""
    return width * height

print(calc_rect_area(3, 5)) #15
```

Definizione di parametri nelle funzioni

Esempio1.

```
def say_hello():
    print('Hello World!')
print(say_hello()) #Hello World!
```

In questo primo esempio abbiamo definito una funzione con 0 parametri, che quindi non è in grado di accettare nessun argomento.

Esempio2.

```
def say_hello(name):
    print('Hello {}!'.format(name))
print(say_hello())
#Traceback (most recent call last):
#File "<stdin>", line 1, in <module>
#TypeError: say_hello() missing 1 required positional argument: 'name'
```

```
print(say_hello('Python')) #Hello Python!
#oppure
say_hello(name='Python') #Hello Python!
```
In questo esempio abbiamo aggiunto un parametro name alla funzione, che viene usato nel messaggio stampato dalla funzione print(). Se proviamo a chiamare la funzione con 0 argomenti o con più di un argomento, riceviamo un TypeError, perché il numero di argomenti passati deve corrispondere a quello dei parametri definiti.

```
Esempio3.
def say_hello(name='World'):
    print('Hello {}!'.format(name))

print(say_hello()) #Hello World!
print(say_hello('Python')) #Hello Python!
```
In questo esempio abbiamo invece aggiunto un valore di default per il name, usando name='World'. Questo rende l'argomento corrispondente a name opzionale: se non viene passato, il valore di name sarà 'World', altrimenti sarà il valore passato come argomento.

```
Esempio4.
def say_hello(*names):
    print('Hello {}!'.format(', '.join(names)))

say_hello('Python') #Hello Python!
say_hello('Python', 'PyPy', 'Jython', 'IronPython')
#Hello Python, PyPy, Jython, IronPython!
```
La * immediatamente prima del nome di un parametro (ad esempio *names) permette alla funzione di accettare un numero variabile di argomenti posizionali. In seguito alla chiamata, la variabile names si riferisce a una tupla che contiene tutti gli argomenti.

```
Nomi locali
```
I nomi delle variabili definite all'interno di una funzione (comprese le sue variabili parametro) non sono visibili al di fuori di essa, per questo si chiamano variabili locali. Invece, le variabili definite al di fuori di una funzione (come tutte quelle che abbiamo usato finora) si chiamano variabili globali. Si parla di ambito di visibilità: variabili che appartengono a due zone di memoria diverse si trovano in ambiti diversi di visibilità.
```
def inputPositiveInteger(message) :
    while True :
        n = int(input(message))
        if n > 0 :
            break
    return n
message = 'ciao'
x = inputPositiveInteger('X: ')
print(message) #"ciao"
```

Il fatto che, durante l'esecuzione della funzione inputPositiveInteger, alla sua variabile locale message venga implicitamente assegnata la stringa "X: " (che è stata ricevuta come argomento) non ha alcun effetto sul valore della variabile globale message, che rimane "ciao", come verifichiamo con l'enunciato print finale.

NB: non possiamo usare le “parole chiave” del linguaggio (es. if, and, while…) come nome di variabile (altrimenti ci verrà segnalato l’errore “object is not callable”).

Scope delle variabili

Tutti i parametri e le variabili create all'interno di una funzione, sono locali alla funzione, cioè possono essere usate solo da codice che si trova all'interno della funzione. Se proviamo ad accedere a queste variabili dall'esterno della funzione otteniamo un NameError. Python segue una semplice regola di risoluzione dei nomi:

1. prima verifica se il nome esiste nel namespace locale;
2. se non esiste lo cerca nel namespace globale;
3. se non esiste neanche nel namespace globale, lo cerca tra gli oggetti builtin. Se un nome non è presente neanche tra gli oggetti built in, Python restituisce NameError.

Variabili diverse in namespace diversi possono riferirsi allo stesso oggetto, ad esempio:

```
def add_elem(seq, elem):
    seq.append(elem)

lista = [1, 2, 3, 4, 5]
print(lista) #lista [1, 2, 3, 4, 5]
add_elem(lista, 6)
print(lista) #lista [1, 2, 3, 4, 5, 6]
```

Sia la variabile globale lista che la variabile locale seq fanno riferimento alla stessa lista. Questo vuol dire che le modifiche fatte alla lista dalla funzione add_elem() saranno visibili anche all'esterno della funzione. Questo ovviamente può accadere solo quando vengono passati a una funzione oggetti mutabili e quando la funzione modifica l'oggetto che è stato passato.

Commenti nella definizione di funzione

Ci sono vari stili “standard” per descrivere in un commento il compito svolto da una funzione. Esistono programmi che leggono i file alla ricerca di questi commenti e generano automaticamente file HTML con la documentazione della funzione, simili a quelli della libreria standard.

```
## Acquisisce in input un numero positivo.
# ... spiegazioni ulteriori...
# @param message il messaggio per l’utente
# @return il numero acquisito
#
def inputPositiveInteger(message) :
    while True :
        n = int(input(message))
        if n > 0 :
            break
    return n
```

Una prima riga con ## descrive sinteticamente il compito svolto poi ci sono, eventualmente successive righe che illustrano meglio il compito svolto. Si trova poi una riga per ogni parametro con la parola chiave @param che precede il nome del parametro e la sua descrizione sintetica. Infine, si trova un eventuale riga @return che descrive l’eventuale valore restituito.

Progettazione mediante stub

Spesso quando si inizia un progetto ci si vuole concentrare sulla logica principale del programma, senza doversi preoccupare dei dettagli (come può essere l'acquisizione di un numero intero). Tuttavia, senza il codice che svolge anche le funzioni più elementari, non si può collaudare il programma completo. Una soluzione può essere quella di progettare inizialmente funzioni che svolgono il loro compito in modo "essenziale", senza occuparsi dei dettagli. Queste funzioni sono dette stub ("mozziconi") o mock ("imitazione").

```
def main() :
   n1 = inputPositiveInteger("Numero 1: ")
   n2 = inputPositiveInteger("Numero 2: ")
   n3 = inputPositiveInteger("Numero 3: ")
   ... # voglio concentrarmi prima su questa sezione...

def inputPositiveInteger(message) : # stub o mock
   print("DEBUG Ricordati di finire inputPositiveInteger")
   return int(input(message)) # il minimo indispensabile
main()
```

Identificatori in Python

Il fatto che ogni informazione sia un oggetto (un numero intero, una stringa, la definizione di una funzione) e che una variabile possa contenere l'indirizzo di un oggetto di qualsiasi tipo, ci consente di fare alcune cose "strane" che in altri linguaggi non sono possibili. Possiamo creare un alias per il nome di una funzione

```
visualizza = print
visualizza("ciao") #come print("ciao")
```

Questo è interessante perché possiamo passare una funzione come un parametro:

```
def doWhatYouWant(x, func) :
    print(func(x))

doWhatYouWant(-2, abs) #visualizza 2
doWhatYouWant(2, float) #visualizza 2.0
import math
doWhatYouWant(2, math.sqrt) #visualizza 1.4142135…
```

Tracciamento dell'esecuzione di un programma

Il collaudo di un programma consente, a volte, di scoprire, che è presente un errore (rilevazione d'errore): questo accade tutte le volte in cui il programma non fa quello che dovrebbe fare (tuttavia, anche se il collaudo va a buon fine, non significa che non ci siano errori!). Dopo aver rilevato un errore, la prima attività da compiere è la diagnosi dell'errore stesso:

- capire dove si trova: fase di localizzazione, richiede il tracciamento dell'esecuzione. Questo processo prevede l'inserimento di elementi print() per poter visualizzare i passaggi intermedi del programma;
- capire qual è: sono necessari vari tentativi inserendo anche appositamente variabili errate per testare la risposta;
- sperare di correggerlo.

Terminata la diagnosi e corretto l'errore, è bene eseguire nuovamente il programma, osservando ancora l'output del tracciamento, per verificare che il problema non sia più presente e successivamente verificare se il collaudo viene superato. E' necessario poi eliminare i messaggi di tracciamento (è molto più efficiente commentarli, potrebbero tornare utili). Una strategia di "programmazione pessimista" è quella di progettare inserendo già messaggi di tracciamento.

Funzioni con numero variabile di argomenti e/o con argomenti predefiniti (informazioni valide anche per metodi e costruttori)

Per definire una funzione che accetta un numero variabile di argomenti, bisogna agire sulla sua firma, premettendo un asterisco al nome della variabile parametro che, al momento dell'invocazione, riceverà un riferimento a una tupla contenente tutti i valori (eventualmente nessuno) forniti come argomenti.

```
def mysum( *values ) :
   sum = 0
   for val in values :
     sum += val
   return sum

print(mysum(1, 2, 3, 4, 5)  #15
```

Si può anche decidere che la funzione debba avere un numero minimo di argomenti, ancorché variabile. Nell'esempio sotto, se la funzione viene invocata con meno di due argomenti, l'interprete segnala errore. Una funzione può avere un unico argomento con l'asterisco, e questo deve essere l'ultimo.

```
def func(firstArg, secondArg, *values) :
    print(firstArg, secondArg)
    for val in values:
        print(val)

func(3, 4, 5) # 3 e 4 su una riga e 5 su un'altra
```

A volte fa comodo definire una funzione che possa ricevere, ad esempio, un parametro obbligatorio e uno facoltativo, con un valore predefinito per il parametro facoltativo mancante. Se la funzione viene invocata con due argomenti, il primo viene assegnato al parametro obbligatorio, il secondo al parametro facoltativo, ignorando il suo valore predefinito (nel caso sotto 22). Se la funzione viene invocata con un argomento, paramFacoltativo assume il valore predefinito. La variabile parametro con valore predefinito deve essere l'ultima nell'intestazione della funzione. Non è necessario che ci siano parametri obbligatori, una funzione può anche avere solo un parametro facoltativo.

```
def func(paramObbligatorio, paramFacoltativo = 22):
    print(paramObbligatorio, paramFacoltativo)

func(2) #2 22
func(2, 3) #2 3
```

Si possono anche definire più parametri con valori predefiniti. Nel caso in cui si voglia fornire un valore solo a p3 e non a p2, bisogna necessariamente utilizzare la tecnica del "parametro con nome" vista in print(..., end=""), ovvero, nel caso dell'esempio sotto, func(3, p3="yy").

```
def func(p1, p2 = 22, p3 = "xx"):
```

```
    print(p1, p2, p3)

func(2) # 2 22 xx
func(2, p2 = 3) # 2 3 xx
func(2, p3 = 3) # 2 22 3
func(2, 3, 3) # 2 3 3
```

NUMERI ED OPERATORI LOGICI

In Python esistono numeri interi e numeri frazionari (eventualmente con parte frazionaria uguale a zero), anche detti “in virgola mobile”. I numeri interi vengono visualizzati senza separatore decimale, mentre i numeri in virgola mobile vengono sempre visualizzati con il separatore decimale, anche quando la parte frazionaria è uguale a zero (i numeri interi sono diversi dai numeri in virgola mobile con parte frazionaria uguale a zero!!). In alcune situazioni Python richiede la presenza di numeri interi, ad esempio nelle stringhe.

I letterali numerici sono del tipo “naturale”, cioè se contengono esplicitamente il punto decimale (anche con parte frazionaria = 0) sono frazionari. Se sono espressi con la notazione scientifica sono frazionari anche se base ed esponente sono interi; altrimenti sono interi. L’operatore \ genera sempre un risultato frazionario, anche quando entrambi gli operandi sono interi.

Operatori aritmetici

Le espressioni aritmetiche possono contenere numeri interi e numeri reali. Nella scrittura dei numeri come separatore decimale si usa il punto. Non si usano i separatori delle migliaia. In generale, gli operatori aritmetici sono:

- `+` per l’addizione;
- `-` per la sottrazione;
- `*` per la moltiplicazione;
- `/` per la divisione (9/4 = 2.25);
- `//` per la divisione intera (9//4 = 2);
- `%` per il resto della divisione (9 % 4 = 1);
- `**` per l’elevamento a potenza.

Le regole di precedenza tra gli operatori aritmetici sono uguali a quelle dell’algebra ordinaria. Nelle espressioni si usano solo le parentesi tonde. Inoltre, diversamente dall’algebra ordinaria, non è possibile lasciare sottinteso il simbolo di moltiplicazione davanti ad una parentesi.

Operatori di confronto

Python supporta anche operatori di confronto, che restituiscono True o False:

- == uguale a;
- != diverso da;
- < minore di;
- <= minore o uguale a;
- > maggiore di;
- >= maggiore o uguale a.

Operatori Booleani

Ogni espressione ha un valore che si ottiene sostituendo le variabili con i loro valori e le invocazioni di funzione con i valori restituiti. Se la variabile x contiene un numero allora x + 10 è un’espressione aritmetica e ha un valore numerico, mentre x < 10 è un’espressione relazionale e ha un valore booleano (dal nome del matematico George Boole).

In Python ogni oggetto è o vero (numeri diversi da 0, la costante True, o contenitori che contengono almeno un elemento) o falso (ovvero il numero 0, le costanti False e None, contenitori vuoti). È possibile verificare se un oggetto è vero o falso usando bool(oggetto). In Python esistono gli operatori booleani and, or e not:

- and ritorna True se entrambi gli operandi sono veri, altrimenti False;
- or ritorna True se almeno uno degli operandi e' vero, altrimenti False;
- not ritorna False se l'operando e' vero, True se l'operando e' falso.

Uso di costanti

Così come si usano nomi simbolici descrittivi per le variabili, è opportuno assegnare nomi anche alle costanti. Un grande vantaggio è che se il valore della costante deve cambiare, in una nuova versione del programma, la modifica va fatta in un solo punto del codice, e non in ogni singola stringa dove se ne utilizza il nome. Per convenzione, i nomi di costanti sono formati da lettere maiuscole. I nomi composti si ottengono collegando le parole successive alla prima mediante un carattere _ . Un primo vantaggio molto importante di questa notazione è la migliore leggibilità.

Funzioni matematiche

Per assegnare a una variabile il valore assoluto di un'espressione, Python mette a disposizione la funzione predefinita abs, `variabile = abs(espressione)`

Round arrotonda un numero all'intero più vicino. La funzione può anche ricevere un secondo argomento, un numero intero che specifica il numero di cifre che devono comporre la parte frazionaria del numero arrotondato restituito, `x = round(2.345742,3) # x vale 2.346.`
Se invece, si vuole ottenere la parte intera di un numero frazionario (senza arrotondamento), si può usare la funzione int. Int, oltre a poter ricevere una stringa da convertire in numero intero, può anche ricevere un numero frazionario, che convertirà in numero intero mediante troncamento alla parte intera, `x = int(2.6) #x vale 2`

Per calcolare il valore minimo in un insieme di valori, si può utilizzare la funzione predefinita min, che può ricevere valori numerici qualsiasi, interi o frazionari. Analogamente, la funzione max calcola il valore massimo (queste funzioni danno errore se ricevono un solo parametro).

```
x = min(2.6, 3, 1.22) # x vale 1.22
```

STRINGHE ED ELABORAZIONE DI STRINGHE

Operazioni di assegnazione

In generale, considerando la semantica dell'enunciato di assegnazione, alla variabile nominata a sinistra dell'uguale viene assegnato il valore ottenuto valutando l'espressione che si trova a destra dell'uguale. In informatica il segno = si usa per indicare un'operazione di assegnazione, che ha una semantica completamente diversa. Capita spesso che il nome della variabile a cui si sta assegnando un nuovo valore compaia anche nell'espressione che si trova alla destra di =.

```
age = int(input("Quanti anni hai? "))
age = age + 2 #equivalente a x +=2
print("Fra due anni avrai", age, "anni")
```

Concatenazione di stringhe

Esempio. Concatenazione di stringhe

```
s1 = "eu"
s2 = "ro"
s3 = s1 + s2 quindi s3 contiene "euro"
```

Per concatenare due stringhe si usa l'operatore + . L'operatore di concatenazione è identico all'operatore di addizione, ma la sua semantica è completamente diversa. I due operandi dell'operatore + devono essere entrambi stringhe o entrambi numeri (e in questo secondo caso viene effettuata un'addizione), altrimenti si verifica un errore. Se vogliamo concatenare una stringa e un numero (valore numerico), dobbiamo prima "convertire il numero in stringa".

```
eu = "euro"
s = str(12.45) + eu
print (s) ottengo 12.45euro (no spazio)
se aggiungo uno spazio, ovvero s = str(12.45) + " " + eu, ottengo
12.45 euro (spazio).
```

`Concatenazione mediante ripetizione o "stringa ripetuta"`

Python mette a disposizione anche una forma di concatenazione mediante ripetizione, che può essere comoda. Questa operazione è, in realtà, "commutativa", nel senso che il fattore (intero) di ripetizione può indifferentemente essere il primo o il secondo operando (e l'altro deve ovviamente essere una stringa), ma è consuetudine che la stringa sia il primo operando.

```
s = "Ciao " * 4
print(s) #visualizza Ciao Ciao Ciao Ciao
```

`Formatted String Literal, o f-string.`

Con le f-string valori, variabili o espressioni possono essere passati all'interno di parentesi graffe.

```
f"Ciao {nome}, il tuo posto è il n{numero}! Benvenuto!"
#'Ciao Jack,  il tuo posto è il n18! Benvenuto!'
```

`Indexing e slicing`

In Python, è possibile accedere agli elementi di una sequenza, usando la sintassi `sequenza [indice]`. Questo restituirà l'elemento in posizione indice (il primo elemento ha sempre indice 0). È inoltre possibile specificare indici negativi che partono dalla fine della sequenza (l'ultimo elemento ha indice -1, il penultimo -2, ecc.). Questa operazione è chiamata indexing.

```
s = 'Python'
print(s[0]) # elemento in posizione 0 'P'
print(s[5]) # elemento in posizione 5 'n'
print(s[-1]) # elemento in posizione -1 'n'
print(s[-4]) # elemento in posizione -4 't'
```

La sintassi `sequenza[inizio:fine]` ci permette di ottenere una nuova sequenza dello stesso tipo che include tutti gli elementi partendo dall'indice inizio (incluso) all'indice fine (escluso). Se inizio è omesso, gli elementi verranno presi dall'inizio, se fine è omesso, gli elementi verranno presi fino alla fine. Questa operazione è chiamata slicing (letteralmente "affettare").

```
s = 'Python'
print(s[0:2])  #elementi da 0 (incluso) a 2 (escluso) 'Py'
print(s[:2])   #dall'inizio all'elemento con indice 2 (escluso) 'Py'
print(s[3:5])  #dall'elemento con indice 3 (incluso) a 5 (escluso)'ho'
print(s[4:])   #dall'elemento con indice 4 (incluso) alla fine 'on'
```

Possiamo verificare se una stringa è una sottostringa iniziale (detta prefisso) di un'altra oppure se ne è una sottostringa finale (detta suffisso). Se uno di questi due metodi restituisce True, altrettanto farebbe l'operatore in, ma non necessariamente il viceversa.

```
s = "topolino e minnie"
if s.startswith("topo") : print("OK") #ok
if s.endswith ("ie") : print("OK")#ok
```

Se si usa un indice che rappresenta una posizione che non è presente nella stringa, si verifica un errore in esecuzione (non di sintassi), perché è un'operazione non eseguibile (come una divisione per 0). In una stringa vuota non esiste alcun indice valido. Dentro le parentesi quadre deve esserci un valore numerico intero (4 e non 4.0), non necessariamente un valore letterale.

```
Esempi:
1. s = '012345'
   i = 0
   while i < len(s) :
      s = s[i:]
      i += 1
   print(s)  #visualizza 012345, 12345, 345
2. s = "xxx"
   for i in range(4) :
      if i % 2 == 1 :
         print(s[i], end="") #termina con IndexError
3. s = "xxx"
   i = 0
   while i <= 3 :
      if i % 2 == 1 :
         print(s[i], end="") #termina con IndexError
      i += 1
4. s = "xxx"
   for i in range(4):
      if i % 2 == 0 :
         print(s[i], end="") #stampa xx e termina correttamente
5. s = "xxx"
   for i in range(4) :
      if i % 2 == 1 :
         print(s[i], end="") #termina con indexError
6. s = "xxx"
   for i in range(4) :
      print(s[i], end="") #termina con indexError
7. s = "xxx"
   for i in range(3) :
      if i % 2 == 1 :
         print(s[i], end="") #stampa x e termina correttamente
8. s = "xxx"
   i = 0
   while i <= 3 :
      print(s[i], end="")
      i += 1 #termina con IndexError
```

`Contenimento: operatori in e not`

Gli operatori in e not in possono essere usati per verificare se un elemento fa parte di una sequenza o no. Nel caso delle stringhe, è anche possibile verificare se una sottostringa è contenuta in una stringa:

```
s = 'Python'
'P' in s  # controlla se 'P' è contenuto nella stringa s, ritorna True
'x' in s  # il carattere 'x' non è in s, quindi ritorna False
'x' not in s  # "not in" esegue l'operazione inversa, ritorna True
'py' in s  # il controllo è case-sensitive (caratteri maiuscoli e
minuscoli sono valutati come diversi, quindi ritorna False
```

`Metodi per stringhe`

Le funzioni accettano 0 o più argomenti e possono essere usate con oggetti di diversi tipi, usando la sintassi `funzione(argomenti)`. I metodi sono simili alle funzioni ma sono legati al tipo dell'oggetto e hanno una sintassi diversa: `oggetto.metodo(argomenti)`.

Così come le funzioni, i metodi possono accettare 0 o più argomenti. Molti linguaggi moderni si caratterizzano per la capacità di implementare la programmazione orientata agli oggetti (OOP, object-oriented programming). Nella programmazione, un oggetto è un'entità (software) che rappresenta il valore di un dato avente un determinato comportamento. Il comportamento di un oggetto è definito dai metodi con i quali lo si può manipolare. I metodi, come le funzioni, hanno un nome e sono sequenze di istruzioni che portano a termine un compito specifico elaborando un oggetto di un determinato tipo. Metodi utili per stringhe sono:

- `s.upper()` restituisce lettere maiuscole al posto di lettere minuscole;
- `s.lower()` restituisce lettere minuscole al posto di lettere maiuscole;
- `s.replace(old, new)` restituisce una stringa costruita a partire dal contenuto s, sostituendo ogni eventuale occorrenza della stringa old con la stringa new.

  ```
  s = "one two one two one"
  print(s.replace(' ', '-')) #visualizza one-two-one-two-one
  ```

Nessuno di questi metodi modifica il contenuto della stringa s, che viene soltanto usata come sorgente di informazioni per costruire la (nuova) stringa restituita.

Nell'elaborazione di stringhe, possono essere molto utili anche questi altri metodi (spesso applicati a stringhe di lunghezza unitaria, ma non solo):

- `s.isalpha()` restituisce True se e solo se la stringa s contiene soltanto lettere, maiuscole o minuscole, e non è vuota;

- `s.isdigit()` restituisce True se e solo se la stringa s contiene soltanto cifre (da 0 a 9) e non è vuota;

- `s.isalnum()` restituisce True se e solo se la stringa s contiene soltanto cifre (da 0 a 9) e lettere, maiuscole o minuscole, e non è vuota;

- `s.islower()` restituisce True se e solo se la stringa s contiene almeno una lettera e tutte le lettere contenute sono minuscole. Possono esserci anche numeri;

- `s.isupper()` restituisce True se e solo se la stringa s contiene almeno una lettera e tutte le lettere contenute sono maiuscole. Possono esserci anche numeri;

- metodi di eliminazione degli spazi:
  - `s.lstrip()`: elimina spaziature iniziali (left)
  - `s.rstrip()`: elimina spaziature finali (right)
  - `s.strip()`: elimina spaziature finali e iniziali

  Si possono usare i seguenti metodi che, come al solito, restituiscono una nuova stringa, senza modificare la stringa s che elaborano. Questi metodi possono anche ricevere una stringa come parametro tra parentesi ed elimineranno tutti i caratteri iniziali e/o finali uguali a quelli contenuti nella stringa ricevuta come parametro tra parentesi. La forma senza parametro equivale a usare \n\t.

  ```
  s.strip("ab") restituisce una stringa uguale a s ma priva di eventuali caratteri a e b iniziali e/o finali, in qualsiasi ordine
  print("aababccacabcaabbaa".lstrip("ab")) #ccacabcaabbaa;
  print("aababccacabcaabbaa".rstrip("ab")) #aababccacabc;
  print("aababccacabcaabbaa".strip("ab")) #ccacabc.
  ```

- `s.split()` facilita la scomposizione di una stringa in sottostringhe sulla base della presenza di caratteri di delimitazione. Split costruisce e restituisce una lista di stringhe ottenuta suddividendo la stringa su cui opera in corrispondenza delle occorrenze della stringa ricevuta come parametro, detta “separatore”. Se la stringa non contiene il separatore, viene restituita una lista di lunghezza unitaria il cui unico elemento è la stringa intera originaria. Se non viene fornito il separatore, viene usato lo spazio, con un comportamento aggiuntivo, ovvero dalla lista sono escluse eventuali stringhe vuote.

  ```
  s = "1763452; 30; 28; 30; 26"
  grades = s.split(";") [1:]
  print(grades) #[' 30', ' 28', ' 30', ' 26']
  ```

- `splitlines()` è una scorciatoia per split(“\n”). Utile, ad esempio, per scomporre la stringa letta da file usando read();

  ```
  print("*\n**\n***") #equivale a scrivere
  print("*")      #*
  print("**")     #**
  print("***")    #***
  ```

- `'separatore da usare'.join()` genera una stringa, data una sequenza di stringhe, che ne è la concatenazione, interponendo un separatore tra stringhe consecutive. Se il separatore è la stringa vuota, il metodo join fa semplicemente la concatenazione delle stringhe presenti nella sequenza.

  ```
  seq = ("ab", "xy", "zzz")
  sep = "**"
  t = sep.join(seq)
  print(t) #t = "ab**xy**zzz"
  ```

- `sorted()` è in grado di ordinare una sequenza di qualsiasi tipo. Per meglio dire, riceve una sequenza (stringa, lista, tupla, range) e restituisce una lista ordinata contenente gli

stessi elementi. Con questo metodo è possibile ordinare una tupla, ovvero crearne una nuova (a partire da una lista ordinata) e assegnarla alla stessa variabile.
`s = sorted("acdb")# s = [“a”, “b”, “c”, “d”]`

Sequenze di escape

Per visualizzare una stringa che contiene delle virgolette, ci sono due possibili soluzioni:

- utilizzare come delimitatori le virgolette singole: `print(‘Hello, “World”!’));`
- usare una sequenza di escape (`print("Hello,\"World\"!")`). Si parla di sequenza di escape, perché serve a far uscire l’interprete dalla normale analisi lessicale. A questo punto, per inserire un carattere backslash in una stringa, si usa la sequenza escape \\.

Le sequenze di escape si usano anche per inserire caratteri di lingue straniere o simboli che non si trovano sulla tastiera. Ad esempio, per scrivere parole italiane con lettere accentate senza avere a disposizione una tastiera italiana si usa `print("Perch\u00E9")`.

Confronto lessicografico tra stringhe

Gli operatori relazionali di uguaglianza e diversità funzionano correttamente anche per confrontare stringhe. Due stringhe sono uguali se e solo se hanno la stessa lunghezza e caratteri identici in posizioni corrispondenti (z e Z sono caratteri diversi!). Si definisce l’ordinamento lessicografico come estensione dell’ordinamento alfabetico. Partendo dall’inizio delle stringhe (sinistra), si confrontano a due a due i caratteri in posizioni corrispondenti, finché una delle stringhe termina oppure due caratteri sono diversi:

- se una stringa termina, precede l’altra; se terminano contemporaneamente sono uguali;
- l’ordinamento alfabetico tra due stringhe è uguale all’ordinamento alfabetico tra i primi due caratteri diversi (indipendentemente da quali siano gli eventuali caratteri successivi).

Il confronto lessicografico induce un ordinamento simile a quello alfabetico ad un comune dizionario. L’algoritmo è identico, l’unica cosa che serve è definire un “ordinamento lessicografico tra singoli caratteri”, che sostituisca quello alfabetico. Questo ordine è definito dallo standard Unicode (www.unicode.org). Usando gli operatori relazionali <, <=, > e >= tra stringhe, si ottengono risultati che dipendono dall'ordinamento lessicografico. La stringa X “è minore” della stringa Y se e solo se X precede Y nell’ordinamento lessicografico. Ovviamente X “è maggiore” di Y se e solo se Y “è minore” di X. Alcune regole generali:

- le cifre numeriche precedono le lettere;
- tutte le lettere maiuscole precedono tutte le lettere minuscole;
- il carattere di spazio precede tutti gli altri caratteri

CICLI (O INTERAZIONI)

In Python esistono due tipi di cicli (anche detti loop):

- il ciclo for: esegue un'iterazione per ogni elemento di un iterabile;
- il ciclo while: itera fintanto che una condizione è vera.

Il ciclo for

Il ciclo for ci permette di iterare su tutti gli elementi di un iterabile ed eseguire un determinato blocco di codice. Un iterabile è un qualsiasi oggetto in grado di restituire tutti gli elementi uno dopo l'altro, come ad esempio liste, tuple, set, dizionari (restituiscono le chiavi), ecc.

```
seq = [1, 2, 3, 4, 5]
for n in seq:
print('Il quadrato di', n, 'è', n**2)
```

Possiamo notare che:

- il ciclo for è introdotto dalla keyword for, seguita da una variabile, dalla keyword in, da un iterabile, e infine dai due punti (:);
- dopo i due punti è presente un blocco di codice indentato;
- il ciclo for itera su tutti gli elementi della sequenza, li assegna alla variabile n, ed esegue il blocco di codice;
- una volta che il blocco di codice è stato eseguito per tutti i valori, il ciclo for termina.

E' possibile usare un if all'interno di un ciclo for:

```
seq = [1, 2, 3, 4, 5]
for n in seq:
    print('Il numero', n, 'è', end=' ')
#le frasi vengono stampate sulla stessa riga perchè utilizzo "for"
if n%2 == 0:
    print('pari')
else:
    print('dispari')

Esempio. Ciclo for e stringhe
random_alnum = "dj1oi3u4nuqnoru01u3m3mdasd"
counter = 0
match = "d"

for char in random_alnum:
    if char == match:
        counter += 1

output = f"Ho trovato {counter} caratteri '{match}' "
# "Ho trovato 3 caratteri 'd' "

Funzione built-in range
```

La funzione range è particolarmente utile se combinata con il ciclo for. Dato che spesso accade di voler lavorare su sequenze di numeri, Python fornisce una funzione built-in chiamata range che permette di specificare un valore iniziale o start (incluso), un valore finale o stop (escluso), e uno step, e che ritorna una sequenza di numeri interi:

```
range(5)  #oggetto range con start uguale a 0 e stop uguale a 5
print(list(range(5)))  #convertendolo in lista vediamo [0, 1, 2, 3, 4]
print(list(range(5, 10)))  #specifichiamo start e stop [5, 6, 7, 8, 9]
print(list(range(0, 10, 2)))  #specifichiamo lo step [0, 2, 4, 6, 8]
```

Range può anche essere usato in combinazione con il ciclo for se vogliamo ripetere un blocco di codice un numero fisso di volte:

```
for x in range(3):
    print('Python')

Esempio1. La stringa contiene almeno una coppia di caratteri
consecutivi uguali?
s = "mamma" #se s = "mano", la funzione ritorna False
hasDuplicates = False
```

```
for i in range(len(s)-1) :
    if s[i] == s[i+1] :
      hasDuplicates = True
      break
print(hasDuplicates)#visualizza "True" (è vero che ci sono duplicati)
```

Il ciclo while

Il ciclo while itera fintanto che una condizione è vera. La condizione `while` permette di accorciare molti programmi ed ha la medesima sintassi di if. Con if sarebbe necessario riscrivere la condizione ogni singola volta che si ripete il carico. Si noti anche che con if non si è sempre sicuri di quante volte è necessario scriverlo, con while questo problema si risolve alla base perchè il numero è "intrinseco" a se stesso.

```
Esempio1. Rimuovi e printa numeri da seq finchè ne rimangono solo 3
seq = [10, 20, 30, 40, 50, 60]
while len(seq) > 3:
    print(seq.pop())# 60 50 40 (su 3 righe diverse)
```

Possiamo notare che:

- il ciclo while è introdotto dalla keyword while, seguita da una condizione (len(seq) > 3) e dai due punti (:);
- dopo i due punti è presente un blocco di codice indentato;
- il ciclo while esegue il blocco di codice fintanto che la condizione è vera;
- una volta che la sequenza è rimasta con solo 3 elementi, la condizione len(seq) > 3 diventa falsa e il ciclo termina.

Break e continue

Python prevede 2 costrutti che possono essere usati nei cicli for e while:

- break: interrompe il ciclo;
- continue: interrompe l'iterazione corrente e procede alla successiva.

```
Esempio1. Ciclo for per cercare un elemento in una lista e
interrompere la ricerca appena l'elemento viene trovato:
seq = ['alpha', 'beta', 'gamma', 'delta']
for elem in seq:
    print('Sto controllando', elem)
    #Sto controllando alpha
    #Sto controllando beta
    #Sto controllando gamma
    if elem == 'gamma':
        print('Elemento trovato!') #Elemento trovato!
        break
```

Non appena il ciclo raggiunge l'elemento 'gamma', la condizione dell'if diventa vera e il break interrompe il ciclo for. Dall'output si può vedere che 'delta' non viene controllato.

Ciclo infinito

Esistono errori logici che impediscono la terminazione di un ciclo, dando luogo ad un ciclo infinito. L'esecuzione del programma continua ininterrottamente. Bisogna arrestare il programma con un comando del sistema operativo (Ctrl-C in Windows), oppure addirittura riavviare il computer. Se

dovesse capitare che si crei un ciclo infinito, basta andare sul terminare, scrivere xkill e, successivamente, selezionare la finestra che si vuole chiudere.

Diagrammi di flusso

Per comprendere il funzionamento di un ciclo e/o verificare di averlo progettato correttamente, è spesso utile seguire la sua esecuzione passo dopo passo, confrontandola con quanto previsto "a mano". Per farlo, si possono inserire degli enunciati print all'interno del corpo del ciclo, per poi rimuoverli (o commentarli).

I diagrammi di flusso rappresentano graficamente la successione di enunciati che vengono eseguiti in un programma e sono utili per rappresentare algoritmi semplici o parti "critiche" di un algoritmo. Basta seguire le frecce che, solitamente, sono dirette dall'alto verso il basso e da sinistra verso destra:

- un blocco rettangolare contiene istruzioni che vengono eseguite attraversandolo. Può avere più ingressi e deve avere un'unica uscita (se rappresenta un algoritmo non devono esserci ambiguità);
- un blocco romboidale contiene una domanda. Può avere più ingressi e deve avere almeno due uscite. A ogni possibile risposta alla domanda, deve corrispondere una e una sola uscita dal blocco, identificata mediante etichette. Istruzioni e domanda possono essere espresse in qualsiasi linguaggio, anche naturale o misto. Per realizzare un'alternativa, si utilizza else, la clausola facoltativa dell'enunciato if. Se la verifica ha successo, viene eseguito il primo corpo dell'enunciato if/else altrimenti, viene eseguito il secondo corpo. Non è un costrutto sintattico obbligatorio, ma è molto utile.

Enunciati if annidati e alternative multiple: if-elif-else

E' possibile annidare enunciati, tuttavia è fondamentale ricordarsi la sintassi. Quando un if si trova all'interno del corpo di un altro if o di una clausola else si parla di "if annidato" (nested, in inglese). Non è una costruzione in particolare, semplicemente è una delle possibiità previste dalle regole sintattiche, molto utilizzata. Utilizzando in modo opportuni gli if annidati, si possono realizzare alternative multiple. In questo modo si genera una struttura decisionale equivalente, nei diagrammi di flusso, a un rombo con più di due uscite (e relative etichette).
Se le alternative sono molte, il codice "scivola" rapidamente verso destra, per ovviare a questo e rendere il codice più chiaro, Python mette a disposizione la clausola `elif` che va indentata come l'if a cui appartiene. La clausola conclusiva, else, è anche in questo caso facoltativa (dipende dalla logica che si vuole realizzare). Le istruzioni condizionali vengono utilizzate quando vogliamo eseguire un blocco di codice solo nel caso in cui una condizione sia vera o falsa.

Acquisire una sequenza di dati

Molti problemi di elaborazione richiedono la lettura di una sequenza di dati in ingresso. Ad esempio, calcolare la somma di dieci numeri forniti dall'utente.

```
print('Scrivi 10 numeri, uno per riga')
sum = 0
count = 0
while count < 10 : #ciclo a contatore
  sum = sum + float(input())
  count = count + 1
print('Somma:', sum)
```

Per rendere il programma più flessibile, si può chiedere all'utente la lunghezza della sequenza (è ancora un ciclo a contatore, ma il valore finale del conteggio non è più prefissato).

```
NUM = int(input("Quanti numeri? "))
while count < NUM :
```

Può succedere, però, che l'utente non sappia quanti saranno i dati che fornirà in ingresso. Ad esempio perché li sta acquisendo a sua volta da uno strumento di misura e la durata dell'esperimento non è prefissata oppure, più in generale, perché è scomodo o impossibile contarli prima di iniziare a scriverli.

```
print('Questo programma somma numeri positivi')
print('Termina i dati scrivendo un numero negativo')
num = float(input())
while num > 0: #sentinella negativa per terminare il programma
... #come sopra
```

Approccio a sentinella

L'utente introduce un dato "non valido" che segnala la fine dei veri dati. Tuttavia, non è sempre possibile individuare tali valori "non validi", cioè non appartenenti al dominio di elaborazione del problema. Ad esempio, un programma che somma numeri deve poter accettare qualunque numero… si può terminare la sequenza di dati con una stringa non numerica (cioè usando una sentinella di tipo diverso dai dati), detta "fuori dominio", anche se questo complica il codice.

```
STOP = 'STOP'
print('scrivi i numeri da sommare, uno per riga')
print('Termina i dati scrivendo", STOP')
sum = 0
done = False
while not done :
   readString = input()
   if readString == STOP :
      done = True
   else:
      sum = sum + float(readString)
print('Somma: ', sum)

> Alternativa 1. Uso una variabile booleana avente un significato
diverso, quindi ne cambio il nome
. . . prime 4 righe come sopra
more = True
while more :
    readString = input()
    if readString == STOP :
       more = False
    else:
        sum = sum + float(readString)
print('Somma:', sum)

> Alternativa 2. Uso break
. . . prime righe come sopra
while True : # sembra un ciclo apparentemente infinito
    readString = input()
```

```
    if readString == STOP :
       break # termina il ciclo immediatamente!
    else:
        sum = sum + float(readString)
print('Somma:', sum)
```

Quando l'interprete esegue l'enunciato break, pone termine immediatamente al ciclo che sta eseguendo. L'iterazione in corso NON viene completata, il ciclo termina "bruscamente" ma è ciò che voglio… L'esecuzione dell'enunciato break deve essere condizionata, cioè deve essere nel corpo di un if o else, altrimenti il ciclo si interrompe sempre durante la sua prima iterazione.

Cicli che elaborano stringhe_esempi

Capita frequentemente di utilizzare cicli che elaborano stringhe, perché ci sono molti problemi che si risolvono compiendo la medesima elaborazione su ciascun carattere di una stringa.

Esempio1. Stampa una per riga le lettere di una stringa: si usa una "variabile indice". Tale indice viene usato per estrarre sottostringhe di lunghezza unitaria dalla stringa elaborata

```
s = 'AUGURI'
i = 0
while i < len(s) :
   print(s[i])
   i += 1 # A U G U R I (ogni lettera su una riga diversa)
```

Esempio2. Conta i caratteri della stringa che soddisfano una determinata condizione: la struttura del ciclo è identica alla precedente, cambia solo l'elaborazione eseguita dal corpo. Ci sono due contatori: i controlla il ciclo, mentre count conta gli eventi

```
s = 'AUGURI'
count = 0
letter = 'U'
i = 0
while i < len(s) :
   if s[i] == letter :
      count +=1
   i = i + 1
print('contiene', count, letter) #contiene 2 U
```

Esempio3. Spazio tra le lettere di una stringa: per progettare un ciclo come questo bisogna concentrarsi su ciò che dovrà fare una qualsiasi delle sue interazioni (cioè il corpo del ciclo). In pratica, bisogna ipotizzare che le interazioni precedenti abbiano svolto il loro compito, producendo una soluzione parziale.

```
s = "AUGURI"
r = "" # stringa vuota che crescera`...
i=0
while i < len(s) :
   r = r + s[i] + " "
   i += 1
print(r) # A U G U R I
```

```
Esempio4. Visualizzare la stringa inversa:
s = 'AUGURI'
r = '' #comoda per iniziare a costruire il risultato (stringa vuota)
i = 0
while i < len(s) :
   r = s[i] + r #la concatenazione non è commutativa!
   i += 1
print(r)

Esempio5. Ci sono (almeno) due caratteri consecutivi uguali?
s = 'Auguri Marcello'
found = False
i = 0
while i < len (s) - 1 :
   if s[i] == s[i + 1] :
      found = True
      break #è importante non estrarre caratteri oltre la fine
   i += 1
if found :
   print('Yes')
else :
   print('No') #yes

Esempio6. Combinando le soluzioni di problemi semplici si possono
risolvere problemi più complessi, anche con cicli annidati.
STOP = 'STOP'
print('Scrivi stringhe e termina con', STOP)
while True:
   s = input()
   if s == STOP : break
   found = False
   i = 1
   while i < len(s) :
      if s[i - 1] == s[i] :
         found = True
         break
      i += 1
   if found : print(s, 'contiene caratteri duplicati')
   else: print(s, 'non contiene caratteri duplicati')
```

IMPAGINAZIONE DEI RISULTATI

Spesso quando si visualizzano i risultati dell'elaborazione svolta da un programma, si vuole avere un controllo preciso sul formato delle informazioni visualizzate. In Python esistono gli indicatori di formato. Questi iniziano con un carattere %, seguito da informazioni opzionali e da un carattere che identifica il tipo di dato che verrà inserito nella stringa prodotta:

- % f per arrotondare i numeri frazionari (anche se la parte frazionaria è zero);
- % e per usare la notazione esponenziale o scientifica;
- % d per troncare alla parte intera un numero frazionario con parte frazionaria !=0;
- % s per una stringa;

- %% per il carattere % (analogamente a quanto visto per le sequenze di escape).

Le informazioni opzionali vanno inserite dopo il carattere iniziale % ma prima del carattere finale che identifica il tipo di valore:

- n numero intero indica il minimo numero di posizioni che saranno occupate dal valore
  `x = '%5s' % 'yz' , print(x) #visualizza x = "   yz" (3 spazi)`
- un numero intero preceduto da un“punto specifica il numero di cifre dopo la virgola
  `x = "%.4f"  % 5.678967897 , print(x) #5.6790 -> arrotonda`
- se le informazioni precedenti sono presenti entrambe, vanno scritte come sotto:
  `x = "%10.4f" % 5.678967897 , print(x) # '    5.6790'`

La stringa di formato, cioè l'operando sinistro dell'operatore di formato, è semplicemente una stringa, quindi non necessariamente una stringa letterale. Tale stringa di formato può anche essere costruita durante l'elaborazione svolta dal programma

```
x = "paperino"
y = "pippo"
m = max(len(x), len(y))
s = "%" + str(m) + "s" #in questo caso %8s, stringhe di lunghezza 8
print(s % x)
print(s % y)
#visualizza
paperino
   pippo
```

Tabelle o matrici

Python non ha una struttura specifica per memorizzare tabelle o matrici, perché una tabella o matrice è una lista/tupla di righe (o colonne) e ciascuna riga (o colonna) è, a sua volta, una lista tupla di dati. Dato che, in Python, i singoli elementi di una lista/tupla possono essere dati di tipo qualsiasi, possono anche essere liste/tuple. Si usa una diversa lista/tupla per contenere i dati di ciascuna singola riga, poi una lista/tupla per contenere tutte tali liste/tuple che rappresentano le singole righe oppure, analogamente, una lista/tupla di "colonne", ciascuna delle quali viene memorizzata in una lista/tupla.

```
Esempio1. Stampare una matrice
x=[
[2, 4, 6, 8, 10], [3, 6, 9, 12, 15], [4, 8, 12, 16, 20]
    ]
> metodo 1
for i in range(len(x)) : # x[i] è la riga i-esima
   for j in range(len(x[i])) :
      print(str(x[i][j]) + " ", end="")
   print()
print() # riga vuota separatrice

> metodo 2
for i in range(len(x)) : # con indici
   row = x[i] # riga i-esima
   for j in range(len(row)) :
      print(str(row[j]) + " ", end="")
   print()
```

```
> metodo 3
for row in x : # senza indici
   for element in row :
      print(str(element) + " ", end="")
   print()

#Visualizza in tutti e 3 i casi
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
   LISTE: MEMORIZZAZIONE DI STRINGHE
Liste modificabili
```

Python mette a disposizione la funzione “lista” (array o vettore). Questa funzione funziona perfettamente anche con una sequenza di numeri senza bisogno di alcuna modifica

```
x = [] #sarà una lista di stringhe
while True :
  read = input("Inserisci dei numeri e termina con STOP: ")
  if read == "STOP" : # sentinella
     break
  if read in x : # cerco read nella lista
     print("Duplicated")
  else:
      x.append(read) # aggiungo (in fondo) alla lista
  print(x)

x = [] # sarà una lista di numeri
while True :
   read = input('Inserisci numero, per terminare inserisci una riga
vuota: ')
   if read == "" :
       break
   n = int(read)
   if n in x :
       print("Duplicated")
   else:
       x.append(n)
print(x)
```

Una novità rispetto alle stringhe è che le liste sono sequenze modificabili.

```
x = [2, 3, 43, 7, 2, 3]
x[3] = 12 # modifico il quarto elemento
print(x) # visualizza [2, 3, 43, 12, 2, 3]
```

Usando la stessa sintassi vista per le stringhe, possiamo estrarre una porzione di lista, che è essa stessa una lista. Questa operazione si chiama slicing. Dato che le liste sono modificabili, l’operazione di slicing ha applicazioni interessanti e complicate.

```
Esempio. Porzioni di lista
```

```
x = [2, 3, 43, 7, 2, 3]
# sostituisco una porzione con un'altra anche di dimensioni diverse
x[1 : 4] = [5, 1]
print(x) # visualizza [2, 5, 1, 2, 3]
```

Lista immodificabile: tupla

Spesso capita di dover usare liste che, per la natura dell'algoritmo che le utilizza, sono immutabili. In Python una lista immutabile si chiama tupla ed è una funzione built-in. Per creare una tupla si usano le parentesi tonde anziché quadre. Per una tupla vuota le parentesi sono obbligatorie. Se sappiamo che una lista sarà immutabile, è meglio usare una tupla, per motivi di efficienza di esecuzione. Usando i tipi predefiniti di Python è possibile affermare che una tupla di stringhe puo' far parte di un insieme e, inoltre, puo' svolgere il ruolo di chiave in un dizionario (mentre una lista di stringhe (o un insieme di stringhe) non puo' far parte di un insieme e non può svolgere il ruolo di chiave in un dizionario). C'è un piccolo problema nella definizione di una tupla letterale di lunghezza unitaria, perché l'interprete la confonde con un'espressione tra parentesi. Per evitare il problema, aggiungiamo una virgola dentro le parentesi

```
x = (5, )
print(len(x))  # 1
```

Scandire il contenuto di una lista

Per scandire il contenuto di una lista si può procedere in due modi diversi:

- un ciclo a contatore (for o while) che usi tutti gli indici validi nella lista, come abbiamo visto per le stringhe (soluzione più flessibile);
- un ciclo for che scandisca gli elementi della lista senza usare esplicitamente gli indici (soluzione più semplice).

Due liste possono essere concatenate utilizzando l'operatore +, che genera una nuova lista

```
x = [2, 3]
y = [1, 7]
z = x + y # z = [2, 3, 1, 7] # x e y non vengono modificate
```

Analogamente, si concatenano tuple, MA non si concatenano liste e tuple. Per concatenare una lista è una tupla generando una lista, devo prima "trasformare" la tupla in lista, usando la funzione predefinita list. Viceversa, si usa la funzione tuple.

```
x = [2, 3]
y = (1, 7)
z = x + list(y)
print(z) # z = [2,3,1,7]

x = [2, 3]
y = (1, 7)
z = tuple(x) + y
print(z) # z = (2,3,1,7)
```

Come per le stringhe, è applicabile l'operatore di replicazione (*), molto comodo per creare una lista/tupla di elementi identici (ad esempio una lista di zeri)

```
x = [0] * 12
print(len(x)) # 12
y = [2, 4] * 3
```

```
print(y) # y = [2, 4, 2, 4, 2, 4]
```

Il metodo append aggiunge un nuovo elemento in fondo a una lista, aumentando la lunghezza di un'unità. Non funziona con le tuple, che sono liste non modificabili.

```
x = [3, 2, 3]
x.append(7), print(x) # x = [3, 2, 3, 7]
```

Operatori di confronto tra liste e tuple

Gli operatori di uguaglianza e diversità funzionano correttamente con liste e tuple: due liste/tuple sono uguali se e solo se hanno la stessa lunghezza e hanno elementi identici in posizioni corrispondenti. Liste e tuple aventi lo stesso contenuto sono DIVERSE. Gli operatori di confronto < e > funzionano in modo "ragionevole", utilizzando un'estensione ovvia dell'algoritmo di confronto lessicografico tra i dati in sequenza. Ovviamente funzionano anche gli operatori <= e >=. Se i tipi di dati non sono compatibili, si verifica un errore.

```
Esempio1.
x = [2, 3]
y = [2, 3]
if x == y :
   print("OK") # OK

Esempio2.
x = [2, 3]
y = [2, 1]
if y < x :
   print("OK") # OK

Esempio3.
x = [2, 3]
y = ["1", 4]
if y < x : ... # ERRORE
```

NB:

- se a e b sono variabili che contengono liste ed e' vero che a+b == b+a, NON e' necessariamente vero che a == b (ovvero, può essere vero che a != b);
- se e' vero che a+b != b+a, NON puo' essere vero che a == b.

```
Operatori in e not in per liste e tuple
friends = ["Alan", "Helen", "Mike" ]
if "Alan" in friends : print("OK") # OK
if "Goofy" not in friends : print("OK") # OK
if not ("Goofy" in friends) : print("OK") # OK
if "ele" not in friends : print("OK") # OK

x = (2, 5, 3, 7) # con liste di ogni tipo di dato
if 3 in x : print("OK") # OK
a = 4
if a + 1 in x : print("OK") # OK
```

Gli operatori in e not in funzioneranno anche quando il secondo operando è una stringa (anziché una lista/tupla). In questo caso, questi operatori vogliono due stringhe come operandi, una a sinistra e una a destra, e producono come risultato il valore booleano True se e solo se la stringa di sinistra è (o, rispettivamente, non è) una sottostringa della stringa di destra.

```
s = "topolino e minnie"
if "lino" in s : print("OK")    # OK
if "xx" not in s : print("OK") # OK
```

L'utilità di questi operatori risulta evidente pensando a come si risolvono gli stessi problemi usando cicli di scansione. Altre utili funzioni predefinite:

- `sum` restituisce la somma dei valori contenuti in una lista/tupla di valori numerici: `x = sum((3, 1, 2)) # x = 6.`
  Non funziona con liste/tuple di più stringhe (cioè non concatena stringhe...); se è vuota restituisce zero: `x = sum([]) + sum(()) # x = 0`
- `min` restituisce il minimo tra i valori numerici contenuti nella lista/tupla, che non deve essere vuota (ma può avere un solo valore): `x = min([3.2, 1, 2]) # x = 1.` Funziona anche con una lista/tupla di più stringhe, usando il confronto lessicografico: `x = min(["aca", "ab"]) # x = "ab".` Funziona anche con una stringa, convertita in una tupla di singoli caratteri: `x = min("zxaf") #x = "a"`
- analogamente, `max` restituisce il valore massimo

`Clonazione di liste`

```
x = [1, 2, 3]
y = x #è un alias
z = list(x) #è un clone
```

Y aggiunge semplicemente una nuova etichetta alla scatola [1, 2, 3]. Si crea un alias, cioè un riferimento, di nome diverso, con cui accedere in lettura e scrittura alla medesima lista individuata da x. Z, invece, crea una nuova lista, un clone della prima. Al termine dell'esecuzione della funzione list, le due liste sono identiche. In futuro, le due liste non avranno alcuna relazione tra loro e quindi modifiche ad una non comporteranno modifiche all'altra.

`Lista e tuple come argomenti di funzione`

Come abbiamo visto, una lista o una tupla, come qualsiasi altro tipo di dato, può essere fornita a una funzione come argomento.

```
def mysum(values) : # come sum
   total = 0
   for element in values :
      total += element
      return total
x = [1, 3, 5, 4, 7]
print(mysum(x)) # 20
x = (1, 3, 5, 4, 7)
print(mysum(x)) # 20
```

`Scambio di variabile temporanea`

Per scambiare il contenuto di due posizioni di una lista serve una variabile temporanea. Per molti aspetti, una posizione in una lista si comporta come una singola variabile dello stesso tipo.

```
# Richiediamo all'utente di inserire i valori delle variabili a e b
```

```
a = int(input("Inserisci il valore di a: ")) #inserisco 6 (esempio)
b = int(input("Inserisci il valore di b: ")) #inserisco 8

# Stampiamo i valori iniziali delle variabili
print("I valori inseriti sono a:", a, "e b:", b)#a=6, b=8

# Utilizziamo una variabile temporanea per lo scambio
temp = a
a = b
b = temp

# Stampiamo i valori delle variabili dopo lo scambio
print("Dopo lo scambio, i valori sono a:", a, "e b:", b) #a=8, b=6
```

Metodi che operano su liste e (alcuni) su tuple

- `lista.index`: Per sapere in quale posizione si trova l'elemento. Se l'elemento è presente più volte, viene restituito l'indice minimo; se l'elemento non c'è, viene generato un errore (ValueError).

  ```
  x = [5, 6, 6]
  print(x.index(6)) # 1
  ```

  Il metodo index può anche ricevere un secondo parametro, un numero intero che indica l'indice da cui iniziare la ricerca (proseguendo, poi, verso indici crescenti); se tale secondo parametro è assente, viene assunto uguale a zero.

  Il metodo find è un'alternativa a index. Se l'argomento non è una sottostringa della stringa con cui il metodo è stato invocato, restituisce -1 (index solleverebbe l'eccezione IndexError). Altrimenti restituisce il minimo indice (non negativo) in cui, nella stringa in cui si cerca, inizia una delle occorrenze della sottostringa cercata. Come index, ha anche la versione con due parametri, per specificare il punto di partenza.

  ```
  s = "mototopo e autogatto"
  print(s.find("to")) # 2
  print(s.find("x"))  # -1
  ```

- `lista.append(elem)`: aggiunge elem alla fine della lista;
- `lista.extend(seq)`: estende la lista aggiungendo alla fine gli elementi di seq;

- `lista.insert(indice, elem)`:

  Il metodo insert consente di aggiungere un elemento a una lista in qualsiasi sua posizione. Riceve due argomenti, nell'ordine: l'indice in cui posizionare l'elemento e il nuovo elemento. Se l'indice è uguale alla lunghezza della lista, l'effetto è identico a quello di append: nuovo elemento aggiunto in fondo, senza spostare altri elementi.

  ```
  x = [1, 3, 4]
  x.insert(1, 2) #(index, element)
  print(x) # visualizza [1, 2, 3, 4]
  ```

- `lista.pop()`:Il metodo pop riceve come argomento un indice ed elimina dalla lista l'elemento che si trova in posizione indice, diminuendo di un'unità la lunghezza della lista. Tutti gli elementi aventi indice maggiore di indice vengono spostati nella posizione immediatamente precedente. Se l'indice non è valido nella lista, si verifica un errore IndexError. Il metodo pop restituisce il valore eliminato.

  ```
  x = [1, 2, 3, 4, 5]
  ```

```
x.pop(2)
print(x.pop(2)) #visualizza 3, che è l'elemento rimosso
print(x) #visualizza [1, 2, 4, 5]
```

- `lista.remove(elem)`: Il metodo remove elimina dalla lista l'elemento ricevuto, diminuendo di un'unità la lunghezza della lista. Se ci sono più elementi uguali, viene eliminato quello con indice minore.

```
x = [1, 2, 3, 2]
x.remove(2)
print(x) #visualizza [1, 3, 2]
```

- `lista.sort()`: ordina gli elementi della lista dal più piccolo al più grande;
- `lista.reverse()`: inverte l'ordine degli elementi della lista;
- `lista.copy()`: crea e restituisce una copia della lista;
- `lista.clear()`: rimuove tutti gli elementi della lista.

`Algoritmi di ricerca in liste e tuple`

Per calcolare il valore massimo in una lista/tupla possiamo usare la funzione max. Questa funzione trova anche la "stringa massima" in una lista/tupla di stringhe, nel senso di stringa che si troverebbe alla fine della sequenza se questa venisse ordinata secondo il criterio lessicografico.

```
Esempio1. Trovare la stringa più lunga
x = ['ciao', 'matilde', 'forchetta'] # una lista/tupla di stringhe
longest = x[0] # ipotesi: Questa è la più lunga... vista finora!
for i in range(0, len(x)) :
  if len(x[i]) > len(longest) : # trovata una più lunga
      longest = x[i] # aggiorno "la più lunga finora"
print(longest)#forchetta

Esempio2. Posizione della stringa più lunga
x = ['ciao', 'matilde', 'forchetta']
pos = 0 # ipotesi di partenza...
for i in range(1, len(x)) :
  if len(x[i]) > len(x[pos]) :
      pos = i # aggiorno
print(pos)#2
print(x[pos])#forchetta

Esempio3. Cercare la posizione di un elemento aventi specifiche
caratteristiche. Ad esempio, una stringa avente iniziale maiuscola
x = ['ciao', 'Matilde', 'forchetta']
found = False
pos = 0
while pos < len(x) and not found :
  if x[pos][0].isupper() :
     found = True
  else :
     pos += 1
if found :
   print("Found in", pos) #found in 1
```

```
else :
   print("Not found")
```

NB: se x è una variabile contenente un valore numerico:

- l'espressione not(x < 10 or x > 0) NON è vera per nessun valore di x;
- l'espressione not(x < 0 and x > 10) è sempre vera, per qualsiasi valore di x;
- l'espressione not(x < 10 and x > 0) è vera quando x <= 0 o x >= 10.

REDIREZIONE DI INPUT E OUTPUT

Quando si utilizzano programmi che acquisiscono una sequenza di dati in ingresso (ad esempio, per sommarli o trovarne il valore medio), è molto scomodo dover scrivere l'intera sequenza sulla tastiera. E' decisamente più facile istruire il programma perché legga il file. In questo caso, chiediamo l'aiuto al sistema operativo, o più precisamente alla shell dei comandi (in Windows chiamata "prompt dei comandi"). Quando eseguiamo l'interprete Python, istruendo perché esegua uno specifico sorgente Python scriviamo nella shell python (ad esempio) myProgram.py.
Aggiungendo una clausola che inizia con il carattere < e contiene il nome di un file, chiediamo al sistema operativo di cooperare con l'interprete perché questo usi un flusso di input diverso da quello proveniente dalla tastiera `python myProgram.py < textFile.txt`

Durante l'esecuzione del programma da parte dell'interprete, quando viene eseguita la funzione input, i caratteri vengono letti ordinatamente dal file anziché dalla tastiera. Durante l'esecuzione del programma la tastiera non funziona più: i dati devono essere inseriti nel file esattamente come li scriveremmo sulla tastiera, andando a capo quando serve. Questo non richiede nessuna modifica al sorgente del programma. Questo fenomeno, del quale il programma non è nemmeno consapevole, si chiama redirezione o reindirizzamento di input.

Archiviazione di dati in output

Quando eseguiamo un programma che visualizza molti dati, sarebbe comodo poterli archiviare in un file, per poterli analizzare successivamente. Possiamo di nuovo chiedere l'aiuto del sistema operativo con la redirezione di output `python myProgram.py > output.txt`

L'intero flusso di caratteri visualizzato dal programma con le invocazioni della funzione print non finisce più nella solita finestra ma nel file. Se il file è già presente, viene sovrascritto, cioè il suo contenuto viene cancellato e vi vengono scritti soltanto i caratteri prodotti dall'esecuzione.

GESTIONE DI FILE

Mediante la redirezione di input/output un programma Python può leggere dati da un file e scrivere dati in un file in modo "trasparente" (o "implicito") cioè senza usare alcuna funzione particolare. È il sistema operativo che indirizza i flussi in modo diverso. Ci sono però diversi limiti:

- si può leggere un solo file e si può scrivere un solo file;
- se si usa la redirezione di input, non si possono acquisire dati dalla tastiera;
- se si usa la redirezione di output, non si possono scrivere dati sullo schermo.

Per superare questi limiti, dobbiamo manipolare i file in modo esplicito all'interno del programma. Per quanto riguarda i file di testo, cioè file contenenti caratteri, ciascuno dei quali codificato in binario secondo lo standard Unicode; questi sono file leggibili e modificabili con un editor di testo.
Per leggere un file di testo, cioè acquisire dati che siano memorizzati in esso, bisogna per prima cosa aprire il file. Questa operazione richiede un'interazione con il sistema operativo il quale, a

sua volta, gestisce il file system, cioè la strategia di organizzazione dei file all'interno di un dispositivo di memoria secondaria. Avviene tramite l'invocazione predefinita open, che restituisce un oggetto che rappresenta il file all'interno del programma.

`Leggere file di testo: open e close`

Gli argomenti per la funzione open sono il nome del file nel file system e una stringa che descrive la modalità di apertura ("r" per la lettura, read). Il nome del file può essere relativo o assoluto: è assoluto se indica il percorso per trovare il file a partire dalla radice del file system, altrimenti il percorso è relativo alla cartella in cui si sta eseguendo il programma. In un sistema Windows, i nomi dei percorsi contengono il carattere \ che, ovviamente, va inserito nella stringa come \\ perchè una barra è utilizzata come escape. Il file va chiuso invocando il metodo close.

Per acquisire dati (sempre di tipo stringa) da un file già aperto, ci sono diverse possibilità, solitamente invocando metodi. Il metodo readline è molto simile alla funzione input che usiamo per acquisire stringhe dal flusso di input standard. Diversamente da input, readline termina sempre la stringa restituita con il carattere \n, tranne quando è arrivato alla fine del file: in tal caso restituisce la stringa vuota

```
f = open("input.txt", "r")
line = f.readline()
while line != "" :
  line = f.readline()
f.close()
```

Ogni invocazione di readline restituisce la successiva riga del file di testo, a partire dalla prima (l'oggetto f si ricorda dov'è arrivato). Una riga è definita come una sequenza di caratteri, eventualmente vuota, terminata da un carattere \n.

In un ciclo for, se usiamo come contenitore l'oggetto di tipo "file di testo" restituito dalla funzione open, Python lo considera come una sequenza di stringhe (le singole righe del file, come se le leggesse con readline). La variabile line farà riferimento alle righe di testo del file, una dopo l'altra (ciascuna terminata da un carattere \n).

```
f = open("input.txt", "r")
for line in f :
f.close()
```

Un altro metodo utile può essere readlines: restituisce una lista di stringhe, ciascuna delle quali è una delle righe del file di testo, terminata da un carattere \n. In pratica, legge l'intero file in un solo colpo. E' comodo soprattutto quando si vogliono fare più elaborazioni successive del contenuto di uno stesso file, perchè non c'è bisogno di leggerlo due volte: il contenuto del file rimane sempre disponibile all'interno della lista creata e restituita da readlines.

```
f = open("input.txt", "r")
lines = f.readlines()
f.close()
for line in lines :
i = 0
while i < len(lines) :
   line = lines[i]
   i += 1
```

Il metodo read() restituisce un'unica stringa contenente tutti i caratteri del file (compresi i caratteri \n che separano una riga dalla successiva). E' utile soprattutto quando si vogliono elaborare i caratteri del testo senza tener conto della sua suddivisione in righe perché, in tal caso, la suddivisione in righe sarebbe solamente scomoda. Il metodo read può anche ricevere un parametro di tipo intero (in tal caso restituisce una stringa contenente al massimo quel numero di caratteri (o meno, se termina il file) e invocazioni successive di read ripartono dal punto in cui era arrivato in precedenza (utile per leggere un file a blocchi)).

```
f = open("input.txt", "r")
s = f.read()
count = 0
for c in s :
  if c.isupper() :
      count += 1

f = open("input.txt", "r")
lines = f.readlines()
count = 0
for line in lines :
  for c in line :
      if c.isupper() :
         count += 1
```

Indipendentemente dal metodo utilizzato per acquisire dati da un file di testo, se si vuole "ripartire dall'inizio" nella scansione delle righe è necessario chiudere il file e aprirlo nuovamente, invocando open. Naturalmente, se il contenuto del file è stato acquisito mediante il metodo readlines, la lista di righe che è stata restituita può essere scandita più volte: è una lista di stringhe, il fatto che il suo contenuto provenga da un file è del tutto ininfluente. Analogamente, per una stringa acquisita con read, è possibile avere più file contemporaneamente aperti (ad esempio, leggere righe alternativamente da un file e da un altro file).

Scrivere file di testo

Per scrivere un file bisogna per prima cosa aprire il file in modalità scrittura ("w"). Se il file esiste già, viene "svuotato" prima di iniziare a scrivere. Usando, invece, la modalità append ("a"), se il file esiste già, i nuovi dati vengono aggiunti alla fine (se il file non esiste, questa modalità coincide con quella precedente). Per scrivere una stringa si usa il metodo write, che, diversamente da print, non aggiunge caratteri \n : li dobbiamo inserire noi esplicitamente. Naturalmente, la stringa fornita come argomento a write può anche essere generata con l'operatore di impaginazione %.

Gestione di file: scrivere un programma che legge due file

Esempio 1 - Testo 1: names.txt

```
1763452;Minnie
1122543;Mickey Mouse
1235436;Donald Duck
1324321;Daisy Duck
```

Testo 2: grades.txt

```
1763452;30; 28; 30; 26
1122543;28; 30; 30; 30
1235436;24; 18
```

```
1324321;30; 30; 30; 30
```

Il formato del programma nel testo 1 si chiama CSV (comma-separated values) ed è usato dai programmi spreadsheet o fogli elettronici come Excel. Come separatore si possono usare caratteri diversi (virgola, due punti, ecc.), qui viene usato il punto e virgola. Ogni riga si chiama record e ogni suo elemento si chiama campo o field.

Il programma legge due file e deve di conseguenza generare un file contenente i numeri di matricola, i nomi degli studenti e la media dei voti conseguiti. Il file generato si chiamerà averages.txt. La logica del programma è: "leggi ripetutamente una riga da ciascuno dei due file di input e genera la riga corrispondente nel file di output. In pratica, il file da generare è uguale a names.txt con l'aggiunta della media dei voti al termine di ciascuna riga.

```
def computeAvg(s) :
   # s ha questa forma:
   # "1763452;30;28;30;26\n"
   # il numero di matricola va ignorato
   # il numero di voti non e` predefinito
   return ... # la media dei voti

in1 = open("names.txt", "r")   # read
in2 = open("grades.txt", "r")  # read
out = open("averages.txt", "w")# write
for line1 in in1 : # leggo names.txt
   out.write(line1[:-1]) # toglie \n finale
   out.write(";")
   line2 = in2.readline() # leggo grades.txt
   avg = computeAvg(line2)
   out.write(str(avg) + "\n")
in1.close()
in2.close()
out.close()
```

Modificare un file esistente

Per modificare il contenuto di un file esistente, come risultato di una elaborazione del suo attuale contenuto, la strategia più semplice è quella di:

- aprire il file in lettura;
- acquisire l'intero contenuto del file, memorizzando in una stringa, con read(), oppure in una lista di stringhe, con readlines();
- chiudere il file;
- aprire il file in scrittura, azione che provoca la cancellazione del contenuto del file;
- elaborare il contenuto del file precedentemente acquisito;
- scrivere nel file il suo nuovo contenuto;
- chiudere il file (e preventivamente farne un backup).

File system

Il file system è il sistema con cui sono organizzati file e cartelle nel sistema operativo. Ha una struttura gerarchica (solitamente un albero): ogni cartella ha una cartella padre o meglio una "radice", ed ogni file è contenuto in una di queste. In Linux è "/", in Windows "\" preceduto dal nome del drive, ad esempio " C:\".

La posizione di file e cartelle in un file system viene specificata attraverso i percorsi (o path), che possono essere assoluti o relativi. Il percorso assoluto parte sempre dalla radice:

- Linux: /home/lab8/
- Windows: C:\Users\lab8\

Il percorso relativo, invece parte sempre dalla cartella "corrente", indicata con il carattere ".":

- Linux: ./lab8/
- Windows: .\lab8\

Python è in grado di interagire con il file system: crea/rimuove cartelle, rimuove file, naviga nel file system… ; questo grazie a moduli come os e os.path:

- `os.sep`: variabile che contiene il carattere separatore ("/" in Linux, "\\" in Windows)
- `os.curdir`: variabile che contiene il path relativo della cartella corrente;
- `os.pardir`: variabile che contiene il path relativo della cartella padre;
- `os.getcwd`: funzione per ottenere il path assoluto alla cartella padre;
- `os.listdir()`: ritorna il contenuto della cartella corrente (non indica se si tratta di file o di cartelle!);
  Ad esempio, la cartella home contiene i file m1.mp3, foto.jpg e la cartella lab8. `os.listdir() restituisce ["m1.mp3", "foto.jpg", "lab8"]`
  `os.listdir("/home/lab8/")`: riporta il contenuto della cartella selezionata.

Per verificare se il path indica un file o una cartella usiamo rispettivamente due funzioni che ritornano True o False: `os.path.isfile(path)` e `os.path.isdir(path)`. Python dispone anche di una funzione che fornisce la dimensione di un file in byte, ovvero `os.path.getsize()` che ritorna un numero. `os.path.getsize("/home/foto.jpg")#7985`

GESTIONE DELLE ECCEZIONI

In situazioni di errore, molte funzioni e molti metodi della libreria Python sollevano eccezioni, ovvero azioni che provocano l'interruzione brusca dell'esecuzione del programma. Viene visualizzato uno strano messaggio di errore, significativo per i programmatori che lo sanno interpretare. Se ad una funzione vengono forniti argomenti errati che cosa può fare? La cosa più facile è imparare a prendere contromisure nel momento in cui una funzione solleva un'eccezione. Questa strategia si chiama gestione delle eccezioni (exception handling). Nel momento in cui una funzione solleva un'eccezione, questa viene catturata mediante un apposito costrutto sintattico predisposto dal programmatore e viene eseguito codice "riparatore".

```
try :
   n= int(input("Un numero intero: "))
   print("Numero valido:", n)
except ValueError :
  # contromisura per ovviare all'errore
   n = int(input("Riprova: "))
```

L'esecuzione dell'enunciato composito try/except prevede che normalmente venga eseguito soltanto il corpo del try (che può essere composto da più enunciati), senza eseguire il corpo della clausola except. Se non vengono sollevate eccezioni all'interno del blocco try, l'esecuzione dell'enunciato try/except è terminata e il programma prosegue con gli enunciati dopo il blocco

except. Nel momento in cui, invece, viene sollevata un'eccezione, l'esecuzione del blocco try termina, dopodichè:

- se l'eccezione è diversa da quella dichiarata nella clausola except, tutto va come se la clausola non ci fosse ed il programma terminerà con un messaggio di errore;
- se l'eccezione è proprio quella dichiarata nella clausola except, il suo effetto viene gestito e non provoca la terminazione del programma: viene eseguito il corpo della clausola except. Successivamente, il flusso dell'esecuzione procede con gli enunciati che seguono try/except - NON si torna a terminare l'esecuzione del blocco try interrotto.

L'enunciato try/except può essere inserito anche all'interno di un ciclo:

```
while True :
  try :
     n = int(input("Un numero intero: "))
     print("Numero valido", n)
     break
  except ValueError :
     print("Riprova")
```

Se l'utente fornisce un numero intero, viene eseguito soltanto il blocco try, che esegue break e il ciclo termina dopo aver memorizzato il numero in n. Altrimenti, int lancia l'eccezione ValueError e l'esecuzione del blocco try si interrompe prima dell'esecuzione di break. Più precisamente, l'esecuzione del blocco try si interrompe prima dell'esecuzione dell'assegnazione di un valore alla variabile n (perché la funzione int non ha generato un valore da poter assegnare a n). Viene poi eseguito il corpo della clausola except, dopodiché l'iterazione del ciclo termina e ne viene eseguita un'altra.

La sintassi prevede che la clausola except abbia un corpo, costituito da almeno un enunciato. Se il messaggio "riprova" non è davvero necessario, possiamo usare l'enunciato "pass", che non fa niente: serve soltanto come riempitivo in quelle situazioni, quando deve necessariamente essere presente un enunciato ma non vogliamo eseguire alcuna azione.

```
while True :
   try :
      n = int(input("Un numero intero: "))
      print("Numero valido", n)
      break #da posizionare tra try ed except, altrimenti il ciclo si
            #interrompe in ogni caso!
   except ValueError :
      pass
```

Per sapere qual è il nome dell'eccezione che viene sollevata da una funzione o da un metodo (es. ValueError, FileNotFoundError, …) è necessario consultare la documentazione oppure, scrivendo il codice senza try/except e fornendo dati che innescano l'eccezione, si scopre il nome dell'eccezione sollevata.

Un'altra istruzione di Python utile in questi contesti è l'istruzione finally, traducibile in italiano come alla fine o infine. Il codice definito nel blocco di finally verrà eseguito alla fine del programma qualsiasi cosa succeda, sia che si manifesti un errore oppure no. In generale, è meglio evitare di usare le istruzioni try ed except su tutto il codice di una funzione, ma cercare sempre di racchiudere solamente il codice necessario.

Sollevare/Riportare eccezioni

In Python, con la funzione raise si possono sollevare eccezioni. Con raise, oltre al tipo di eccezione sollevata, possiamo definire anche una stringa che verrà visualizzata dall'interprete in seguito all'interruzione brusca del programma, se l'eccezione non verrà gestita.

```
def strToInt(s) : # funziona come int(...) predefinita
    if s.isdigit():
        print(s + ' è un numero')
    else:
        raise ValueError(s + " non contiene un numero")

strToInt('5') #5 è un numero
strToInt('Ciao') #ValueError: Ciao non contiene un numero
```

Il tipo di eccezione sollevata da una funzione o da un metodo fa parte della loro documentazione, perché gli utilizzatori lo devono conoscere. L'esecuzione della funzione o del metodo si interrompe nel momento in cui viene eseguito l'enunciato raise. Viene ripresa l'esecuzione del codice invocante, come se la funzione o il metodo avessero eseguito return, ma non c'è valore restituito. Se l'esecuzione riprende all'interno di un blocco try che contiene una clausola except dedicata all'eccezione sollevata, l'eccezione viene catturata e gestita. Altrimenti, anche la funzione o il metodo invocante si interrompono e viene ripresa l'esecuzione del codice che li aveva invocati. Se si arriva a interrompere il codice principale, il programma termina con la visualizzazione del messaggio di errore che conosciamo.

Alcuni esempi di errori:

`NameError`: variabile non definita;

`ZeroDivisionError`: divisione per 0;

`ValueError`: errore di valore

Errori di arrotondamento nel calcolo in virgola mobile

Per oggetti di tipo float Python usa lo standard IEEE 754 a 64 bit che descrive come codificare un numero reale mediante una sequenza binaria di 64 bit. Gli errori di arrotondamento sono un fenomeno inevitabile nel calcolo in virgola mobile eseguito con un numero finito di cifre significative, anche in base decimale. Siamo abituati a valutare questi errori pensando alla rappresentazione dei numeri in base decimale, ma i computer rappresentano i numeri in base binaria.

```
f = 4.35 # tipo float n = int(100 * f)
print(n) # visualizza 434
```

In questo caso l'errore inatteso è dovuto al fatto che 4.35 non ha una rappresentazione esatta nel sistema binario usando un numero finito di cifre. In generale, è meglio invocare round.

I calcoli con numeri in virgola mobile possono introdurre errori di arrotondamento e/o troncamento. Tali errori sono inevitabili e bisogna fare molta attenzione nella formulazione di verifiche che coinvolgono numeri in virgola mobile

```
r = 2**(1/2) # radice quadrata di 2: 1.4142135623730951
x = r**2 # 2.0000000000000004
if x != 2 : print("Help") # Help
```

Per fare in modo che gli errori di arrotondamento non influenzino la logica del programma, i confronti per uguaglianza/diversità tra numeri di tipo float devono prevedere una tolleranza. Il

modulo math in Python mette a disposizione una funzione, `isclose(x, y)`, che restituisce True o False, che gestisce proprio la necessaria tolleranza.

LA LIBRERIA STANDARD DI PYTHON

Oltre a round, abs, … ci sono moltissime funzioni matematiche utili, che non sono predefinite nel linguaggio, ma fanno parte della libreria standard di Python. Nella programmazione, una libreria è una raccolta di codice, generalmente scritto da altri programmatori e pronto per essere utilizzato. La libreria standard di uno specifico linguaggio di programmazione viene considerata parte integrante del linguaggio stesso e deve essere presente in qualsiasi pacchetto software che dichiari di essere un ambiente di sviluppo per quel linguaggio.

Moduli in Python

Per creare un modulo di funzioni importabili, è sufficiente inserire in un normale file sorgente Python le definizioni delle funzioni che fanno parte del modulo, SENZA inserire un programma eseguibile (NB: il file si deve trovare nella stessa cartella!).
`from myinput import *` (il file myinput.py deve essere nella stessa cartella).

La libreria standard di Python è organizzata in moduli. Ciascun modulo contiene la definizione di classi, funzioni, variabili e costanti che riguardano uno specifico campo applicativo:

- math: modulo contenente funzioni e costanti matematiche;
- string: modulo contenente funzioni che elaborano stringhe;
- datetime: modulo contenente funzioni e costanti che aiutano a elaborare orari, date e intervalli temporali.

Per poter utilizzare in un programma le risorse messe a disposizione da un modulo, bisogna importare tale modulo (o una sua parte). La stessa sintassi si usa per i moduli della libreria standard sia per moduli che provengono da altre librerie “scaricate” a parte o per moduli progettati da noi). Ad esempio, math contiene la definizione del valore π con la massima precisione consentita dalle variabili float del linguaggio Python..
`from math import pi, print(pi) # visualizza 3.14159265358979`. Senza l’importazione, l’azione `print(pi)` darebbe errore: `NameError: name ‘pi’ is not defined`

Per importare:

- import modulo va bene quando abbiamo bisogno di accedere a diversi nomi di un modulo senza doverli importare individualmente e a quando vogliamo sapere esplicitamente a che modulo appartiene ogni funzione che chiamiamo);

```
import math
help(math)  # possiamo usare help() per vedere la documentazione
dir(math)  # possiamo usare dir() per vedere il contenuto del modulo
```

- from modulo import nome va bene quando abbiamo bisogno di accedere a un numero limitato di nomi, e quando questi nomi sono abbastanza chiari da evitare ambiguità (ad esempio una funzione search potrebbe cercare diverse cose a seconda del modulo, mentre la funzione sqrt serve chiaramente a calcolare la radice quadrata);

```
from math import pi, sqrt  # importa solo i nomi pi e sqrt
print(pi)  # è ora possibile accedere a pi senza usare math.pi
```

- import modulo as nuovonome e from modulo import nome as nuovonome va bene quando i nomi che vogliamo importare sono ambigui o particolarmente lunghi;

```
import math as matematica
# possiamo accedere agli oggetti di math facendo matematica.nome
#oppure
from math import sqrt as radice_quadrata
# radice_quadrata si riferisce lo stesso a sqrt
```

- from modulo import * può andare bene dall'interprete interattivo (se ad esempio vogliamo usarlo come una potente calcolatrice facendo from math import *) o quando conosciamo esattamente il contenuto del modulo, abbiamo bisogno di tutti i nomi, e siamo sicuri che questi nomi non creano conflitti con nomi esistenti (se ad esempio un modulo contiene una funzione chiamata open, l'esecuzione di from modulo import * andrà a sovrascrivere la funzione builtin open).

```
from math import *  # importa tutti i nomi definiti nel modulo math
```

Creazione di nuovi moduli

In Python, non esiste una vera distinzione tra i moduli/librerie e il file principale che viene eseguito (anche conosciuto in altri linguaggi come main). Qualsiasi file con estensione .py può essere sia eseguito che importato. Per esempio, possiamo creare un file che definisce le quattro operazioni aritmetiche di base:

```
# aritmetica.py
def add(a, b):
    return a + b
def sub(a, b):
    return a - b
def mul(a, b):
    return a * b
def div(a, b):
    return a / b
```

Dopo aver salvato questo file come aritmetica.py, possiamo importarlo dall'interprete interattivo o da un altro modulo:

```
import aritmetica  # importiamo il modulo
# possiamo accedere alle funzioni definite all'interno del modulo
aritmetica.add(3, 5)
```

Se proviamo a eseguire il file, non otteniamo nessun output ma neanche nessun errore. Questo accade perché aritmetica.py non esegue nessuna operazione che produce output.

ESPRESSIONI REGOLARI O CANONICHE

A volte si vogliono suddividere stringhe usando separatori più complessi rispetto ad un'unica stringa separatrice. Si può a tal fine pensare ad una versione della funzione split che accetti una lista di separatori. Più in generale, ci serve un linguaggio per descrivere insiemi di stringhe senza doverle elencare esplicitamente.

Modulo re: funzione split e findall

Nel modulo re di Python è definita un funzione split che accetta un'espressione regolare e una stringa, da suddividere usando qualunque separatore che sia descritto dall'espressione regolare.

```
from re import split
s = "http://python.org"
regex = "[^a-zA-Z]+" #vedi dopo per significato
words = split(regex, s)
```

```
print(words) #['http', 'python', 'org']
```

Il linguaggio usato per descrivere insiemi di stringhe è abbastanza complesso. Innanzitutto, dal punto di vista sintattico, in Python un'espressione regolare è una stringa. La regola base è che un'espressione regolare costituita soltanto da caratteri non speciali descrive l'insieme contenente soltanto la stringa stessa. Ad esempio, la stringa regolare "abcz" descrive l'insieme di stringhe costituito dalla sola stringa "abcz". Quindi, se alla funzione split forniamo come espressione regolare una stringa "normale", il suo comportamento è come quello del metodo split: c'è solo un separatore. Un carattere speciale, invece, preceduto dal carattere \ perde le sue caratteristiche speciali e rappresenta se stesso (regola "opposta" a quella usata nelle sequenze di "escape").
La funzione findall del modulo re restituisce una lista con le sottostringhe che corrispondono all'espressione regolare. Nell'ordine in cui le trova dall'inizio alla fine della stringa (lista vuota se non trova corrispondenze), senza sovrapposizioni (cioè inizia a cercare la seconda stringa a partire dal carattere successivo all'ultimo della prima stringa trovata, e così via...)

```
from re import findall
s = "http://python.org"
regex = "[a-zA-Z]+"
words = findall(regex, s) # words = ['http', 'python', 'org']
regex = "[^a-zA-Z]+"
seps = findall(regex, s) # cerca i separatori # seps = ['://', '.']
```

A volte è più comodo descrivere i separatori, altre volte è più comodo descrivere ciò che si cerca (in questo caso è sostanzialmente equivalente)

Caratteri speciali nelle espressioni regolari

- il carattere | posto tra due sotto-espressioni regolari significa "oppure"
  `"ab|cd|z"` descrive l'insieme costituito dalle stringhe "ab", "cd" e "z";
  `"ac\|cd"` descrive invece l'insieme costituito dall'unica stringa "ac\|cd"
- il carattere . significa "un carattere qualsiasi tranne \n";
  `"a."` descrive l'insieme costituito da tutte le stringhe di due caratteri (non necessariamente solo lettere) che iniziano con a, tranne la stringa a\n
  `"b.\."` descrive l'insieme costituito da tutte le stringhe di quattro caratteri che hanno b come secondo carattere e terminano con il carattere punto;
- la coppia di parentesi quadre [ ] significa "un carattere qualsiasi tra quelli indicati tra parentesi". All'interno delle parentesi, gli insiemi di caratteri possono anche essere espressi come intervalli di caratteri, indicando il primo e l'ultimo, separati da un trattino. Per identificare quali siano i caratteri interni all'intervallo si usa la codifica Unicode.
  `"a[bcd]"` descrive l'insieme costituito dalle stringhe "ab", "ac" e "ad" (che è equivalente alla regex "ab|ac|ad")
  `"a[a-z]"` descrive l'insieme costituito da stringhe che iniziano per a e proseguono con una lettera minuscola;
  `"a[a-zA-Z0-9;]"` descrive l'insieme costituito dalle stringhe di due caratteri che iniziano con a e hanno come secondo carattere una lettera (minuscola o maiuscola) oppure una cifra numerica o un "punto e virgola";
- se il primo carattere all'interno di una coppia di parentesi quadre è ^, questo significa "un carattere qualsiasi che non sia tra quelli indicati tra parentesi", ovvero fa il contrario di quanto espresso al punto precedente.

`Caratteri speciali come postfissi`

Se la descrizione di un carattere è seguita da:

- \+ significa “una o più volte”;
  `"ab+"` descrive l’insieme infinito delle stringhe “ab”, “abab”, “ababab”, …
  `"[0-9]+a"` descrive l’insieme (infinito) costituito dalle stringhe che terminano con a e iniziano con un numero qualsiasi di cifre (ma almeno una cifra), come "504a" e "9a" ma non "a"
- * significa “zero o più volte”, ovvero una presenza opzionale;
  `"ab*"` descrive l’insieme infinito costituito da "a", "ab", "abb", "abbb", "abbbb", ecc
- ? significa “zero o una” volta, cioè una presenza opzionale ma non ripetuta;
  `"a[0-3]?"` descrive l’insieme costituito dalle stringhe "a", "a0", "a1", "a2", "a3"
- {m} , dove m è un numero intero, significa "il carattere precedente esattamente m volte"
  `"a[0-9]{3}"` descrive l’insieme costituito dalle stringhe di 4 caratteri, il primo dei quali è a e gli altri tre sono cifre qualsiasi (anche diverse tra loro), equivalente a "a[0-9][0-9][0-9]"
- {m,n} , dove m e n sono numeri interi, significa "il carattere precedente almeno m e al massimo n volte". Se manca n (ma c’è la virgola), si intende "n infinito".
  `"a[0-9]{2,4}"` descrive l’insieme costituito dalle stringhe aventi a come primo carattere, seguito da 2, 3 o 4 cifre (anche diverse tra loro), equivalente a `"a[0-9][0-9]|a[0-9][0-9][0-9]|a[0-9][0-9][0-9][0-9]"`
  `"a[0-9]{4,}"` descrive l’insieme (infinito) costituito dalle stringhe aventi a come primo carattere, seguito da almeno 4 cifre (anche diverse tra loro), equivalente a `"a[0-9][0-9][0-9][0-9]+"`

NB: Nelle espressioni regolari, con l’uso dei moltiplicatori con molteplicità indefinita (es. +, *, {qualcosa,}) si possono verificare ambiguità, ovviamente da eliminare

```
from re import findall
s = "cabbbx"
regex = "ab+" # a seguita da "almeno una b"
words = findall(regex, s)
# trova "abbb", ma anche "ab" oppure "abb" sono stringhe che corrispondono alla descrizione della regex…
```

Nell’utilizzo delle espressioni regolari, si risolvono queste ambiguità usando un approccio greedy ("goloso"): viene identificata la stringa più lunga tra quelle che sono descritte dall’espressione regolare.

`Alcuni esempi di suffissi e postfissi`

- l'espressione canonica `[^p]*p[^p]*p.*` descrive l'insieme di stringhe composto da tutte e sole le stringhe che contengono almeno due lettere p;
- l'espressione canonica `[^p]*p[^p]*p[^p]*` descrive l'insieme di stringhe composto da tutte e sole le stringhe che contengono esattamente due lettere p, anche consecutive;
- l'espressione canonica `[^p]*p[^p]+p[^p]*` descrive l'insieme di stringhe composto da tutte e sole le stringhe che contengono esattamente due lettere p non consecutive.

`Caratteri speciali con linguaggio più articolato`

Una coppia di parentesi tonde può essere usata per raggruppare una sotto-espressione regolare, mentre i "moltiplicatori" postfissi ({m,n} e simili) che agiscono sul singolo carattere che li precede, agiscono invece su un’intera sotto-espressione se questa è racchiusa tra parentesi tonde.
`"a[0-9]{3}"` equivale a `"a[0-9][0-9][0-9]"`, quindi descrive l’insieme costituito dalle stringhe di 4 caratteri, il primo dei quali è a e gli altri tre sono cifre qualsiasi

"(a[0-9]){3}" equivale a "a[0-9]a[0-9]a[0-9]", quindi descrive l'insieme costituito dalle stringhe di 6 caratteri, costituite da tre sottostringhe di due caratteri: in ciascuna coppia, il primo carattere è a e il secondo è una cifra qualsiasi. Ad esempio: a0a2a1, a0a0a0, a4a3a7

INSIEMI IN PYTHON

Un insieme è un contenitore di dati che memorizza una raccolta di valori univoci, ovvero senza duplicati. Diversamente da quanto accade in una tupla, gli elementi in un insieme non vengono memorizzati in alcun ordine specifico e non vi si può accedere tramite un indice associato alla loro posizione. Per ottenere prestazioni così eccellenti, gli insiemi sono realizzati con una struttura di memorizzazione dei dati chiamata tabella hash (una "lista di liste" organizzate in modo speciale...). Ma perché non usiamo sempre gli insiemi? Semplicemente perché memorizzano informazioni diverse da quelle memorizzate dalle liste! Gli insiemi non memorizzano una sequenza: i dati non appartengono a una relazione precedente/successivo. Gli insiemi non memorizzano la molteplicità (in una sequenza, invece, il fatto che uno stesso dato sia presente più volte può essere rilevante).

Creare un insieme

Per creare un insieme si usa una coppia di parentesi graffe contenente i dati separati da virgole

```
s = {"pippo", "pluto", "topolino"}
t = {1, 4, 2, 6}
x = {-2, "minnie", (3, 4)} # un elemento dell'insieme x e` una tupla
```

Non è necessario che gli elementi di un insieme siano omogenei, ma devono essere oggetti immutabili: non si può costruire un insieme di liste né un insieme di insiemi.

La funzione set genera un insieme a partire da un contenitore qualsiasi di oggetti immutabili. Per generare un insieme vuoto non si scriverà s = { } ma `s = set()`

```
lst = [1, 2, 4]
s = set(lst)
```

Visualizzare un insieme

E' possibile chiedere alla funzione print di visualizzare un insieme, l'ordine in cui i suoi elementi vengono visualizzati è praticamente "imprevedibile". Per scandire gli elementi di un insieme non possiamo usare un ciclo con indice. L'accesso agli elementi usando un indice tra parentesi quadre non funziona con gli insiemi, perchè non sono sequenze. E' necessario usare un ciclo for che usi direttamente l'insieme come contenitore.

Funzioni e operatori per insiemi

```
t = {"pippo", "pluto", "topolino"}
print(len(t)) # 3
```

verifica la dimensione di un insieme

```
if "pippo" in t :
  print("OK") # OK
if "paperino" not in t :
  print("OK") # OK
```

verifica se un elemento è presente in un insieme

La funzione frozenset crea un insieme immutabile a partire da qualsiasi contenitore, anche un insieme. Gli insiemi immutabili sono più efficienti degli insiemi

```
t = {2, 4, 1}
s = frozenset(t)
```

Metodi per gli insiemi

- `add`: per aggiungere un elemento ad un insieme. Se è già presente, non viene aggiunto e la cardinalità dell'insieme non cambia;

```
t = {2, 5, 4}
print(len(t)) # 3
t.add(1), print(len(t)) # 4
t.add(2), print(len(t)) # sempre 4, l'elemento era già presente
```

- `discard/remove()`: eliminano dall'insieme l'elemento ricevuto come argomento. Se l'argomento non è presente, discard non fa nulla, mentre remove solleva KeyError;
- `clear()`: rende vuoto l'insieme su cui agisce ed è più rapido di uno svuotamento mediante ripetute eliminazioni;
- `issubset()` verifica se un insieme è un sottoinsieme di un altro insieme. Lo stesso risultato si ottiene confrontando gli insiemi con l'operatore <=;
- `s.union` restituisce l'insieme unione;
- `s.intersection` restituisce l'insieme intersezione;
- `s.difference` restituisce l'insieme differenza;
- `confronto tra insiemi`: due insiemi sono uguali se contengono gli stessi elementi (l'ordine è ininfluente). Per il confronto si usano gli operatori == e !=. A differenza di liste e tuple che non sono mai uguali, set e frozenset lo possono essere.

DIZIONARI IN PYTHON

Un dizionario (o mappa) è un contenitore di dati che memorizza una raccolta di coppie di tipo chiave/valore, cioè ogni dato è, in realtà, una coppia di due elementi che hanno ruoli diversi:

- la chiave (key) è l'elemento che identifica univocamente la coppia all'interno del dizionario (cioè non ci possono essere due coppie con la stessa chiave) e tecnicamente deve essere un oggetto immutabile (numero, stringa, frozenset, tupla di oggetti immutabili...);
- il valore (value) è un dato di qualsiasi tipo, anche modificabile, associato alla chiave all'interno di una coppia (possono esistere più coppie aventi lo stesso valore, purché abbiano chiavi diverse).

Creare un dizionario

Per creare un dizionario contenente alcune coppie iniziali, si usa una coppia di parentesi graffe. Le coppie sono separate da virgole (come in ogni contenitore) e ciascuna coppia ha il formato chiave : valore (i due punti distinguono un dizionario da un insieme). Le chiavi di un dizionario devono essere oggetti immutabili, sui valori non ci sono vincoli. Le chiavi nei dizionari sono proprio il corrispettivo degli indici nelle liste: le usiamo per richiamare i valori ad esse associati.

```
d = {123432:"Mickey Mouse",\
     112232:"Daisy Duck"}
```

La funzione `dict()` crea un dizionario vuoto. La funzione dict crea anche dizionari con contenuto specificato, ma in tal caso è preferibile utilizzare delle parentesi graffe. La funzione è, però, molto comoda per clonare un dizionario.

```
d = {..:.., ..:..}
e = dict(d)
```

Visualizzare un dizionario

L'ordine in cui le coppie vengono visualizzate è praticamente imprevedibile. Si può chiedere alla funzione print di visualizzare. `print(d)`.

Metodi e operatori per dizionari

- `metodo dict.keys()` consente di ottenere una lista di tutte le chiavi presenti;
- `metodo dict.values()` permette di ottenere una lista di tutti i valori;
- `metodo items()` permette di ottenere una lista di tutte le coppie chiavi-valore presenti;
- `len()`: la lunghezza di un dizionario è il numero di coppie che contiene;
- `in e not in`: verificano se esiste una coppia avente una determinata chiave;

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
if 112232 in d : # chiavi, non valori o coppie
  print("OK") # OK
```

- `[]`: all'interno di questo operatore si trova una chiave, non un indice. Se nel dizionario non è presente una coppia con la chiave indicata viene sollevata l'eccezione KeyError. Spesso si condiziona l'accesso usando una verifica con operatore in.

```
if 112232 in d :
   name = d[112232]
```

Questo operatore può essere usato anche, come nelle liste, per modificare un valore. Se il dizionario non contiene una coppia con la chiave specificata non è un errore. Viene aggiunta una nuova coppia con quella chiave e quel valore.

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
d[123432] = "Mr. Mickey Mouse" #modifica il nome della chiave
print(d[123432])#"Mr. Mickey Mouse"
```

- `metodo get`: riceve una chiave da cercare e un valore da restituire se la chiave è assente (altrimenti restituisce il valore associato alla chiave). Se il metodo get riceve solo la chiave e tale chiave è assente, restituisce l'oggetto speciale None;

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
key = int(input("Matricola da ispezionare: "))
if key in d :
  name = d[key]
else :
  name = "Sconosciuto"
print(name)
```

oppure più velocemente:

```
name = d.get(key, "Sconosciuto")
print(name) #se 112232 -> Daisy Duck, se 112132 -> Sconosciuto
```

- `metodo pop`: per eliminare una coppia da un dizionario, fornendo la chiave come argomento. Restituisce il valore presente nella coppia eliminata. Solleva l'eccezione KeyError se non esiste una coppia con la chiave specificata;

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
key = int(input("Matricola da eliminare: "))
if key in d :
  name = d.pop(key)
```

- `sorted`: restituisce una lista ordinata avente lo stesso contenuto del contenitore che riceve come argomento. Nel caso di un dizionario restituisce una lista ordinata contenente soltanto le chiavi. A differenza del metodo per le liste sort(), che ha lo stesso scopo, può

essere usato anche per le tuple e i set perchè non modifica l'ordine degli elementi all'interno della sequenza originale ma la lascia invariata restituendo una nuova lista.

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
print(sorted(d)) # [112232, 123432]
```

Scandire un dizionario

La funzione sorted è anche utile per scandire un dizionario. Ricordiamo sempre che tale funzione non modifica il contenitore che riceve, ma restituisce una lista con il suo contenuto ordinato (se il contenitore è un dizionario, restituisce una lista con le chiavi ordinate).

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
# visualizza i nomi in ordine di matricola crescente
for key in sorted(d) :
  print(d[key])
#visualizza matricole e nomi, in ordine di matricola crescente
for key in sorted(d) :
  print(key, d[key])
```

Più tecnicamente, quando si usa un dizionario in una posizione in cui è richiesto un contenitore, il contenitore che viene effettivamente utilizzato è "il contenitore delle chiavi del dizionario" che viene ottenuto dall'interprete invocando implicitamente il metodo keys() del dizionario.
Il contenitore restituito da keys() è simile a una lista, è di tipo dict_keys (è una sequenza). Passandolo a list, lo si trasforma in una vera lista.

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
for key in d.keys() : # come for key in d :
   print(key)
```

Volendo invece visualizzare il contenuto di un dizionario seguendo l'ordine dei valori, usiamo in un modo analogo il contenitore restituito dal metodo values(). Il suo contenuto è ordinabile.

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
for val in sorted(d.values()) :
   print(val)
```

Infine, c'è un altro contenitore che può essere richiesto a un dizionario, ovvero il contenitore delle coppie, restituito dal metodo items(s). I singoli elementi di tale contenitore sono tuple di dimensione 2, dove il primo elemento è la chiave. Anche in questo caso, il suo contenuto è ordinabile. Viene usato il confronto tra tuple che, con l'operatore <, restituisce True se il primo elemento della tupla di sinistra è minore del primo elemento della tupla di destra.

```
d = {123432:"Mickey Mouse", 112232:"Daisy Duck"}
for item in sorted(d.items()) :
   print(item[0], item[1])
#oppure
for (key, val) in sorted(d.items()) :
  print(key, val)
# oppure
for key in sorted(d) :
  print(key, d[key])
```

UTILIZZO DI LISTE, INSIEMI E DIZIONARI

Spesso capita di elaborare entità aventi molteplici attributi. Ad esempio, uno studente ha un numero di matricola, un nome, cognome e lista di voti degli esami superati… un tipo di informazione che nell'ambito della gestione delle basi di dati si chiama record. Può essere comodo memorizzare tutte queste caratteristiche all'interno di un unico oggetto che rappresenti lo studente. In questo caso il dizionario è perfetto: basta solo dare un "nome" a ciascun attributo, che sarà la chiave di una coppia che descrive l'attributo.

```
stud = {"id":112233,
        "name":"Mickey",
        "surname":"Mouse",
        "grades":[18,20,19]
       }
```

Nel caso in cui si volesse utilizzare la lista sarebbe necessario doversi ricordare la posizione di ciascun singolo attributo. Se poi decidessimo di eliminare un attributo bisognerebbe modificare tutto il codice che usa il record.

```
stud = [112233,"Mickey","Mouse",[18,20,19]]
print(stud[1]) # il nome e` in posizione 1
```

Intestazione e lettura di un file CSV

Spesso i file CSV (con campi separati da ;) hanno una prima riga di intestazione, dove vengono dati nomi alle colonne: possiamo sfruttarli come chiavi in un dizionario.

```
ID;Name;Surname;Enrollment
1122543;Mickey;Mouse;Law
1235436;Donald;Duck;Maths
1324321;Daisy;Duck;Surgery
1122110;Scrooge;McDuck;Economics
```

A questo punto, creiamo una lista di dizionari, uno per ciascuno studente

```
f = open("csv.txt")
lines = f.readlines() # leggo tutto il file
f.close()
keys = lines[0][:-1].split(";") # chiavi
studs = list() # sara` lista di dizionari
for r in range(1, len(lines)):  #altre righe...
    stud = dict() # dizionario di uno studente
    fields = lines[r].strip("\n").split(";")
    for k in range(len(keys)) : # ogni chiave
    stud[keys[k]] = fields[k] # colonna k
studs.append(stud) # lista di dizionari...
```

PROGETTAZIONE DI OGGETTI E DI CLASSI

Può far comodo memorizzare in un unico oggetto composito tutti gli attributi o proprietà di un'entità elaborata da un programma. In particolare è possibile rappresentare un dato composito (record) mediante un oggetto (es. dizionario o lista). Un record contiene tutti gli attributi di un'entità, ma è soltanto una struttura di memorizzazione, un componente software "passivo". Se il componente diventa attivo è possibile invocare un metodo. Questo ha un grande vantaggio. Infatti, quando usiamo oggetti della libreria di Python o di altri moduli, invochiamo metodi specifici per quegli oggetti. Tali metodi si trovano in uno spazio di nomi distinto da quello delle funzioni. Il fatto che i metodi applicabili a un determinato tipo di oggetti siano effettivamente specifici per quella categoria lascia intendere che dati e metodi siano molto correlati tra loro e sarebbe meglio

poterli definire "insieme", anche dal punto di vista sintattico. Tutte queste considerazioni si traducono nella definizione di una classe, un'azione sintattica che caratterizza la programmazione orientata agli oggetti. Una classe costituisce lo schema progettuale per la costruzione degli oggetti di un determinato tipo e definisce un nuovo tipo di dato. In tale schema progettuale ci sono:

- la descrizione degli attributi di ogni oggetto di quel tipo, cioè le proprietà o caratteristiche;
- la descrizione della procedura di "costruzione" di un oggetto che sia un esemplare di tale classe, cioè che ne segua lo schema progettuale;
- le definizioni dei metodi che sono invocabili con tali oggetti e che li elaborano in vario modo, determinandone l'evoluzione all'interno del programma che li usa.

Definizione di una classe

Il progetto di un nuovo tipo di oggetti si concretizza sintatticamente nella definizione di una classe. Innanzitutto occorre decidere il nome di tale nuovo tipo di dato. Spesso la definizione di una classe viene scritta in un file sorgente che diventa un modulo, anche se solitamente contiene codice dedicato al collaudo della classe stessa (eseguibile condizionatamente usando __name__). Il progetto di un nuovo tipo di oggetti si concretizza sintatticamente nella definizione di una classe

```
# student.py (modulo)
## Uno studente ha un numero di matricola e una lista di voti.
class Student :
    # codice che definisce le
    # caratteristiche e le funzionalita`
    # degli oggetti di tipo Student
```

Class è una nuova parola riservata al linguaggio. Per convenzione (= regola), i nomi delle classi definite al di fuori della libreria standard hanno l'iniziale maiuscola. Nel progetto di una classe, la prima cosa da decidere è la sua interfaccia pubblica, ovvero l'insieme dei metodi invocabili con oggetti che siano esemplari di tale classe. L'interfaccia pubblica di una classe definisce le funzionalità dei suoi esemplari. Viene anche chiamata API (Application Programming Interface) È costituita dai suoi metodi e, per ciascun metodo si inducano nome; elenco delle eventuali variabili parametro e del loro significato; eventuali eccezioni, specificando nome e condizioni che ne provocano il sollevamento; eventuale valore restituito e suo significato; descrizione della funzionalità svolta (non necessariamente il codice del metodo).

```
# student.py (modulo)
## Uno studente ha un numero di matricola e una lista di voti.
class Student :
   ## Aggiunge un voto allo studente.
    # @param grade il voto da aggiungere
    def addGrade(self, grade) : ... # codice del metodo
    ## Ispeziona la media dei voti.
    # @return la media dei voti
```

Parametro self

Tutti i metodi definiti all'interno di una classe devono avere, come primo parametro, la variabile self (nome predefinito, con un significato speciale). L'elaborazione svolta da un metodo può coinvolgere i dati forniti come argomenti nell'invocazione (all'interno delle parentesi tonde che seguono il nome), oppure l'oggetto con cui si invoca il metodo, cioè quello che figura alla sinistra del nome del metodo, separato da un punto. Tale oggetto, passato come gli altri parametri, viene

memorizzato automaticamente dall'interprete nella variabile self (che è un alias per l'oggetto usato nell'invocazione) e viene anche detto "parametro implicito" (gli altri "espliciti"). Purtroppo, se dimentichiamo la variabile parametro self nella definizione di un metodo, non viene segnalato un errore di sintassi. Si verifica un errore non appena proviamo ad invocare tale metodo. La funzione self all'interno di un metodo fa sempre riferimento a un oggetto che sia esemplare della classe in cui è definito il metodo stesso.

`Definizione dello stato degli oggetti`

Dopo aver definito l'interfaccia pubblica di una classe (cioè le funzionalità dei suoi esemplari), mediante l'elenco dei suoi metodi, occorre dedicarsi alla definizione dello stato degli oggetti. Per meglio dire, delle informazioni di stato, ovvero l'insieme delle informazioni che ne riassumono il passato e ne determinano il futuro, in funzione dei metodi che verranno invocati con quell'oggetto
In pratica, dobbiamo chiederci: tra due invocazioni di metodi qualsiasi effettuate con uno stesso oggetto, cosa si deve ricordare tale oggetto per poter assolvere ai propri compiti futuri, sulla base del suo passato? Dopo aver individuato quali informazioni devono essere memorizzate all'interno di ciascun oggetto per consentirne il funzionamento (cioè il funzionamento dei metodi dell'interfaccia pubblica della classe di cui è esemplare), dobbiamo tradurre tali informazioni in codice Python.

Gli elementi sintattici che consentono la memorizzazione dello stato di un oggetto al suo interno fanno parte della definizione della classe di cui l'oggetto stesso è esemplare. Si chiamano variabili di esemplare (o di stato) e, per convenzione che seguiremo, hanno nomi che iniziano con _ . Se decidiamo, ad esempio, di rappresentare lo stato di uno studente con una lista che ne memorizzi i voti, ci basta un'unica variabile di esemplare, una lista. In generale, in un programma si potranno creare più esemplari di una stessa classe. Ciascun esemplare di una classe ha le proprie informazioni di stato. All'interno della zona di memoria dedicata a ciascun singolo oggetto trovano posto le sue variabili di esemplare (che non hanno nessuna relazione con le variabili di esemplare di un altro oggetto dello stesso tipo, anche se hanno gli stessi nomi). Per accedere ad una variabile di esemplare, si utilizza il nome della variabile che fa riferimento all'oggetto a cui ci stiamo riferendo, seguita da un punto e dal nome della variabile di esemplare. Questo tipo di accesso è ok, ma sconsigliato.

```
Ad esempio per stud = Student(...)
v = len(stud._grades)
```

Lo stato iniziale degli esemplari di una classe viene definito mediante un metodo speciale detto "costruttore". Il costruttore, in una classe, è un metodo che si chiama sempre __init__ (doppio underscore prima e dopo). Di norma, il suo compito è quello di assegnare un valore a tutte le variabili di esemplare dell'oggetto appena creato. Viene invocato automaticamente dall'interprete dopo aver creato un oggetto: non va mai invocato esplicitamente.

```
class Student :
   def __init__(self) :
      self._grades = list()
```

Nel costruttore, il parametro self fa riferimento all'oggetto che è in corso di costruzione e che verrà restituito dal costruttore! È tutto gestito dall'interprete, che costruisce l'oggetto assegnandogli uno spazio in memoria e, poi, ne esegue immediatamente il costruttore. Il costruttore ha il compito di assegnare un valore iniziale a tutte le variabili di esemplare.

```
# crea un nuovo oggetto e ne invoca il costruttore
stud = Student() # ora l'oggetto stud e` pronto a funzionare
```

```
print(stud.getAvgGrade())# 0
stud.addGrade(18)
stud.addGrade(20)
print(stud.getAvgGrade())# 19
```

Definizioni dei metodi

La definizione di un metodo è molto simile alla definizione di funzione. I metodi sono definiti all'interno di una classe nel suo corpo; devono avere almeno una variabile parametro e la sua prima variabile parametro deve chiamarsi self. Metodi di solo accesso consentono di ispezionare alcune proprietà degli oggetti con cui vengono invocati; metodi modificatori consentono di modificare lo stato degli oggetti con cui vengono invocati.

Costanti di classe

A volte sorge la necessità di condividere informazioni tra tutti gli esemplari di una stessa classe. Ad esempio, le costanti. Una variabile definita all'interno di una classe ma al di fuori dei suoi metodi si chiama variabile di classe. Ne esiste un'unica copia in una zona di variabile dedicata alla classe, non ai suoi singoli esemplari. Dall'esterno della classe, si accede alle variabili di classe usando il nome della classe seguito dal punto

```
print(Student.PASSING GRADE) #18
```

Metodi che elaborano due esemplari della classe stessa

Per elaborare due esemplari della stessa classe, bisogna definire una relazione d'ordine per gli oggetti di quel tipo. Per gli oggetti non sempre possiamo usare gli operatori di confronto (>, < …), ma serve usare metodi come .isLessThan. Come facciamo a convincere l'interprete a usarli anche con "nostri" oggetti? Definire in una classe il metodo __it__ è condizione necessaria e sufficiente perché oggetti di quel tipo siano confrontabili usando l'operatore <. Altrimenti generiamo un TypeError.

```
class Student:
   def __it__(self, other) :
      return self.getAvgGrade() < other.getAvgGrade()
```

Esiste un metodo anche per eseguire l'operatore ==, ovvero __eq__. Se questo metodo non viene definito ne esiste una definizione "standard" che restituisce True se e solo se i due riferimenti puntano allo stesso oggetto, cioè sono alias dello stesso oggetto.

Per comprendere se due variabili (ad esempio i parametri self e other) fanno riferimento al medesimo oggetto? Si usa l'operatore is, il cui valore è True se e solo se i suoi due operandi sono riferimenti al medesimo oggetto (cioè tecnicamente, solo lo stesso indirizzo). Esiste anche l'analogo operatore is not.

```
class Student :
   def __eq__(self, other) :
       if self is other:
          return True
          return self.getAvgGrade() == other.getAvgGrade()
```

Operatori di confronto tra oggetti

Gli altri quattro metodi di confronto disponibili si possono sempre definire in modo che invochino __eq__ e/o __lt_. Solitamente questi sei metodi si definiscono in tutte le classi per le quali le operazioni di confronto abbiano senso.

```
class Student :
   def __ne__(self, other) : #not equal
       return not (self == other) #invoca __eq__
   def __le__(self, other) : #less than or equal
       return self == other or self < other
   def __gt__(self, other) : #greater than
       return not (self <= other) #invoca __le__
   def __ge__(self, other) : #greater than or equal
       return not (self < other) #invoca __lt__
```

Ereditarietà

L'ereditarietà è uno dei principi basilari della programmazione orientata agli oggetti. L'ereditarietà è il paradigma che consente, tra le altre cose, il riutilizzo del codice. Si usa quando si deve realizzare una classe ed è già disponibile un'altra classe che rappresenta un concetto più generale, meno specifico oppure quando due o più classi condividono un comportamento comune. Distinguiamo sottoclassi e superclassi. La sottoclasse è la derivata della superclasse. In Python esiste anche l'ereditarietà multipla, ovvero una sottoclasse può avere due o più superclassi, in modo che erediti l'unione dei loro comportamenti. Tuttavia, questa caratteristica è di difficile utilizzo. Possiamo usare, invece, l'ereditarietà su più livelli: una classe B che sia sottoclasse di A può diventare la superclasse di un'altra classe, C, la quale erediterà quindi il comportamento di B (e, di conseguenza, anche il comportamento di A, superclasse di B).

Quando una sottoclasse definisce un metodo che ha lo stesso nome di un metodo della superclasse, quello della sottoclasse prevale. Quindi, costruendo un esemplare di una sottoclasse, viene invocato ed eseguito soltanto il metodo __init__ della sottoclasse stessa. Il compito del costruttore è quello di assegnare un valore iniziale alle variabili di esemplare dell'oggetto in fase di costruzione.

Progettazione di eccezioni

L'enunciato raise vuole un oggetto che rappresenti un'eccezione. L'oggetto deve essere un esemplare di BaseException o di una sua sottoclasse. Quindi, per progettare un'eccezione di nostra scelta, dobbiamo progettare una sottoclasse di BaseException con il nome che vogliamo.

```
class MyException(BaseException) :
   pass #se il corpo della classe viene lasciato vuoto,
        #viene segnalato un errore di sintassi
raise MyException
```

Variabili e costanti di classe

Le variabili di classe vengono memorizzate in uno spazio condiviso. Di solito è bene che siano "provate" come le variabili di esemplare. Quando una variabile di classe è costante, non è necessario renderla privata, possiamo darle un nome che non inizi con _; usiamo un nome composto da lettere maiuscole, come per tutte le costanti. Variabili e costanti di classe vengono inizializzate all'esterno dei metodi. L'interprete garantisce che la loro inizializzazione avvenga prima della creazione di qualsiasi esemplare della classe, in modo che possano essere utilizzate all'interno di metodi e costruttore, avendo già un valore.

```
# student.py (modulo)
class Student :
  PASSING_GRADE = 18
  _lastAssignedID = 0
```

```
  def __init__(self) :
     Student._lastAssignedID += 1
     self._ID = Student._lastAssignedID
     self._grades = list()

  def getID(self) :
     return self._ID

  def addGrade(self, grade) :
     if grade < Student.PASSING_GRADE :
        raise ValueError
     self._grades.append(grade)

  def getAvgGrade(self) :
     if len(self._grades) == 0 : return 0
     return sum(self._grades)/len(self._grades)
```

`Metodi di classe`

In pratica è una funzione. Non ha un parametro implicito (non viene invocato usando un esemplare della classe, ma usando il nome della classe), e viene inserito nella definizione della classe perché svolge una elaborazione attinente ai compiti attribuiti agli esemplari della classe. Spesso i metodi di classe svolgono elaborazioni che riguardano le variabili di classe.

`Funzione isinstance()`

Quando una funzione o un metodo/costruttore dichiara di ricevere un parametro, solitamente richiede che il valore effettivamente ricevuto come argomento in ciascuna invocazione sia di un determinato tipo (es. numero intero, stringa, lista). Questa verifica si può fare confrontando il valore restituito dalla funzione type con il nome del tipo di dato che vogliamo

```
def func(x) : # x deve essere un numero intero
   if type(x) != int :
     raise ValueError(str(x) + "non e` int")
```

`isinstance(object, type).` Object è l'oggetto da verificare e type è il tipo o la classe da confrontare. Questa funzione viene utilizzata per verificare se un oggetto è di un certo tipo o se è una sottoclasse di quel tipo.

```
x = 5, isinstance(x, int) # output True
```

La funzione type si usa (soltanto) per ottenere informazioni più specifiche in fase di debugging. Ad esempio, se voglio sapere se la funzione ha ricevuto specificatamente un esemplare di una classe o di una sua sottoclasse. isinstance() è molto utile quando è necessario verificare il tipo di un oggetto prima di eseguire un'operazione su di esso: ad esempio si potrebbe utilizzare per verificare se un valore inserito dall'utente è di tipo numerico prima di eseguire un calcolo.

`RICERCA PER BISEZIONE ( O RICERCA "BINARIA")`

Diciamo che una lista è ordinata in senso crescente se il valore dei suoi elementi aumenta all'aumentare dell'indice. Se ci sono elementi replicati (consecutivi) diciamo che la lista è ordinata in senso "non decrescente" o "crescente in senso lato". In generale, la ricerca in una lista è molto più efficace se la lista è ordinata perché consente di dimezzare le dimensioni di un problema.

La ricerca per bisezione si chiama così perché ad ogni passo divide la lista rimasta in due parti aventi circa le stesse dimensioni. Questo algoritmo funziona soltanto se la lista è ordinata: se è disordinata l'algoritmo fornisce una risposta errata. questo metodo di ricerca è molto più veloce di quello sequenziale.

Attenzione: se le ricerche da fare sono poche, bisogna valutare se valga la pena ordinare la sequenza, perchè anche l'azione di ordinamento ha un costo in termini di elaborazione. Se le ricerche sono poche può convenire usare la ricerca sequenziale su un lista anche non ordinata.

Il modulo bisect contiene la funzione bisect che consente di fare ricerche per bisezione in liste/tuple ordinate. Se la lista non è ordinata, si ottengono risposte, in generale, errate. L'obiettivo della funzione è quello di trovare la posizione di inserimento di un nuovo elemento in una lista ordinata in modo che questa continui a essere ordinata.

```
from bisect import bisect
target = … #elemento cercato
lst = [ ] # lista di elementi ordinabili
lst.sort() #lista ordinata in senso crescente
i = bisect(lst, target)
```

i è l'indice non negativo della posizione in cui andrebbe inserito target in lst per lasciarla in ordine, cioè l'indice che si potrebbe fornire al metodo insert. Se in lst sono già presenti valori uguali a target (tra loro consecutivi), la funzione bisect restituisce il valore massimo tra quelli che restituiscono il requisito specificato.

```
if i != 0 and lst[i-1] == target:
… # target trovato
else:
… # target non trovato
```

Conviene definire una funzione di ricerca per bisezione che invochi bisect

```
from bisect import bisect ## Ricerca per bisezione in lista ordinata.
# @param a lista ordinata in cui cercare
# @param trgt elemento da cercare
# @return (trgt in a)
def binarySearch(a, trgt) :
  i = bisect(a, trgt)
   return i != 0 and a[i-1] == trgt

target = ... # elemento cercato
lst = [...] # lista di elementi ordinabili
lst.sort()  # lista ordinata in senso crescente
if binarySearch(lst, target) :
  ... # target trovato
else :
... # target non trovato
```

Nei problemi di ordinamento di sequenze di parla di:

- ordinamento sul posto: quando la funzione di ordinamento riceve una sequenza e ne ordina il contenuto. Solitamente non restituisce nulla: al termine dell'esecuzione della funzione di ordinamento, il codice invocante si ritrova il contenitore ordinato;
- ordinamento non sul posto: quando la funzione di ordinamento riceve una sequenza e ne restituisce un'altra contenente gli stessi elementi, ma in ordine; il contenitore ricevuto non viene modificato.

ORDINAMENTI PER SELEZIONE (SelectionSort)

Per prima cosa bisogna trovare la posizione dell'elemento minimo presente nell'intera lista. La sua posizione corretta nella lista sarà [0], scambio con una variabile temporanea l'elemento presente in data posizione con quello nuovo. Si procede con il medesimo algoritmo, tenendo conto che la parte sinistra della lista risulta progressivamente ordinata.

```
def selectionSort(a) :
   for i in range(len(a) - 1) :
# cerco l'elemento che deve andare in posizione
# i minPos = findMinPosFrom(a, i)
# l'elemento cercato si trova in posizione minPos
if minPos != i :
swap(a, minPos, i)

def swap(a, i, j) :
   temp = a[i]
   a[i] = a[j]
   a[j] = temp

def findMinPosFrom(a, frm) :
   pos = frm
   i = pos + 1
   while i < len(a) :
     if a[i] < a[pos] :
```

per ordinare liste di altri tipi di dati non c'è bisogno di alcuna modifica, è sufficiente che con tali dati funzioni l'operatore <, cioè che sia definito il metodo __lt__

```
         pos = i
     i += 1
   return pos
```

L'algoritmo di ordinamento per selezione è corretto ed è in grado di ordinare qualsiasi lista. Allora perché si studiano altri algoritmi di ordinamento? Esistono algoritmi che, a parità di dimensione della lista da ordinare, vengono eseguiti più velocemente. Ci occupiamo del tempo di esecuzione degli algoritmi di ordinamento ma, in generale, tra le prestazioni degli algoritmi interessa anche l'occupazione di memoria durante l'esecuzione. Gli algoritmi di ordinamento sul posto spesso non usano memoria aggiuntiva rispetto alla lista da ordinare (interessa principalmente il tempo di esecuzione).

Rilevazione delle prestazioni

Per valutare l'efficienza temporale di un algoritmo si misura il tempo impiegato per la sua esecuzione su insiemi di dati di dimensioni diverse. Il tempo di esecuzione di un algoritmo va misurato all'interno del programma stesso. Si usa la funzione time del modulo time che, a ogni

invocazione, restituisce un valore di tipo float che rappresenta il numero di secondi trascorsi da un evento di riferimento.

```
from time import time
a = list()
. . . # fase di input dei dati
beginTime = time()
selectionSort(a)
elapsedTime = time() - beginTime
print("%f secondi" % elapsedTime)
. . . # fase di output dei dati
```

Le prestazioni dell'algoritmo di ordinamento per selezione hanno un andamento quadratico (o parabolico) in funzione delle dimensioni della lista.

Analisi teorica delle prestazioni

Il tempo di esecuzione di un algoritmo dipende dal numero e dal tipo di istruzioni in linguaggio macchina eseguite dal processore. Per fare un'analisi teorica senza fare esperimenti di esecuzione, dobbiamo fare delle semplificazioni. Nell'esecuzione del codice dell'algoritmo, contiamo soltanto gli accessi in lettura o scrittura a singoli elementi della lista, ipotizzando che tali accessi siano le operazioni elementari più lente durante l'esecuzione del programma. Sperimentalmente, si osserva che questo modello è valido per algoritmi che non modificano la lunghezza della lista.

Spesso ci si trova di fronte a un compromesso tra tempo d'esecuzione e occupazione di memoria: per ridurre il tempo, si può decidere di occupare più memoria (o viceversa).

Esempio. Inserire un elemento in una lista ordinata, in modo che rimanga ordinata

Si parla di "ordinamento mediante inserimento", utile nelle situazioni in cui i dati vengono conservati in una lista ordinata e "ogni tanto" viene aggiunto un dato alla lista:

- per prima cosa bisogna trovare la posizione x di inserimento, mediante una ricerca per bisezione (ricordando che la lista è ordinata): si può usare la funzione bisect del modulo bisect;
- poi, tutti gli elementi della lista, dalla posizione x inclusa in poi, vanno spostati nella posizione immediatamente successiva a quella che occupano, aumentando in questo modo la lunghezza della lista di un'unità: si può usare insert che agisce su una lista, fornendo il parametro x ottenuto al passo precedente;
- in alternativa, si inserisce il nuovo elemento in fondo alla lista e si ordina di nuovo (ignorando il fatto che fosse ordinata).