L'ambiente di programmazione IDLE

Due componenti principali:

- **Interprete** (*shell*): è la finestra che si apre all'avvio dell'ambiente di programmazione, e consente la valutazione di espressioni e l'esecuzione di istruzioni in modalità interattiva
- ***Editor* di testi** (una o più finestre che si possono aprire attraverso la voce di menu *File* > *New window*): consente la scrittura in un *file* di testo di un programma, e successivamente di eseguirlo

Tipi di dato, variabili, espressioni

Esecuzione di un programma

I programmi Python sono sequenze d'istruzioni, e devono essere memorizzati in un *file* di testo. Per la scrittura di programmi si consiglia l'uso dell'*editor* di testi dell'ambiente di programmazione scelto

Per convenzione, i *file* che contengono programmi Python hanno estensione `.py`

Un programma può essere eseguito solo dopo averlo memorizzato in un *file*. Nell'ambiente IDLE L'esecuzione viene avviata selezionando dalla finestra dell'*editor* la voce di menu *Run* > *Run module*, o in alternativa premendo il tasto `F5`

Tipi di dato principali

- Numeri interi; es.: `1, -5`
- Numeri reali; es.: `-2.7, 3.14` (notare l'uso del punto, non della virgola, come separatore tra parte intera e parte frazionaria)
- Caratteri, indicati tra apici singoli o doppi; es.: `'a'  "a"  'b'  'P'  '3'  ';'`
- Valori logici (*Booleani*) "vero" e "falso", indicati con i simboli `True` e `False`

Tipi di dato principali

- Sequenze (*stringhe*) di caratteri, indicate tra apici singoli o doppi; es.: `"python"`, `'questa è una stringa'`
  Per inserire caratteri speciali in una stringa (apici singoli o doppi, *newline*, tabulazioni, ecc.), si usano sequenze di caratteri speciali, dette sequenze di *escape*, che iniziano con il carattere `\` (detto *backslash*). Le principali sono:
  - `\'` inserisce `'` in una stringa
  - `\"` inserisce `"`
  - `\n` inserisce un *newline* ("andata a capo")
  - `\t` inserisce una tabulazione (sequenza di spazi)
  - `\\` inserisce `\` (lo stesso carattere *backslash*)

  Esempio: `"Prima riga\nSeconda riga"`

Tipi di dato principali

- *Dizionari*: *insiemi* di elementi (non ordinati) costituiti da coppie *chiave*/*valore*; ogni *valore* è identificato dalla *chiave* associata
  - **chiave**: numero, carattere o stringa (non possono esistere chiavi identiche in un dizionario)
  - **valore**: un valore *qualsiasi*
  - sintassi: `{`c_1`:`v_1`, ... ,` c_n`:`v_n`}`

  Esempi:
  - `{}` (un dizionario vuoto)
  - `{'nome':'Mario', 'età':25}`
  - `{'giorno':'lunedì',`
    `'data':{'g':10, 'm':3, 'a':2014}}`
    (un dizionario che ne contiene un altro)

Tipi di dato principali

- *Liste*: *sequenze ordinate* di zero, uno o più valori *qualsiasi*, indicati tra parentesi quadre e separati da virgole, ciascuno dei quali è identificato dalla propria posizione nella sequenza (*indice*: un intero a partire da 0); es.:
  - `[]` (una lista vuota)
  - `[1, 2, 3]`
  - `['a', -5.3]`
  - `[8, [5, 1], 3]` (una lista che ne contiene un'altra)

Variabili

Le *variabili* sono nomi simbolici scelti dal programmatore (per es.: `x`, `area`), a ciascuno dei quali è possibile associare un valore *qualsiasi*, attraverso l'istruzione di assegnamento (si veda più avanti)

Nei linguaggi di alto livello le variabili svolgono la stessa funzione delle celle di memoria nel linguaggio macchina

Operatori ed espressioni

- Per i diversi tipi di dato sono disponibili *operatori* (per es., operatori aritmetici) che consentono di scrivere *espressioni*, ciascuna delle quali coinvolge uno o più valori di tipi appropriati, e produce un valore come risultato della sua valutazione
- I nomi delle variabili alle quali sia già stato assegnato un valore possono essere usati all'interno di espressioni
- In particolare, gli operatori *logici* consentono di scrivere *espressioni condizionali*, che consistono nel *confronto* tra coppie di valori o espressioni *qualsiasi*, e producono un valore logico (*Booleano*)

Operatori principali su stringhe e liste

È possibile accedere ai singoli elementi di una stringa o di una lista memorizzata in una variabile, attraverso la seguente espressione:

var [indice]

- var indica il nome della variabile
- indice dev'essere una qualsiasi espressione il cui valore sia un numero intero compreso tra zero e la lunghezza della stringa/lista meno uno

Operatori aritmetici principali

`+`	addizione
`-`	sottrazione
`*`	moltiplicazione
`/`	divisione (se entrambi gli operandi sono interi, calcola la parte intera del quoziente)
`//`	divisione (calcola sempre la parte intera del quoziente)
`%`	modulo (resto della divisione)
`**`	elevamento a potenza

Operatori principali su stringhe e liste

Per esempio, assumendo che le variabili `s`, `L` e `i` contengano rispettivamente la stringa `'Python'`, la lista `['a','b','c']` e il numero `1`:

- `s[0]` ⇒ `'P'`
- `s[i]` ⇒ `'y'`
- `L[i+1]` ⇒ `'y'`
- `L[3]` produce un errore (il quarto elemento della lista `L`, cioè l'elemento di indice pari a 3, non esiste)

Operatori principali su stringhe e liste

\+ concatenazione

Esempi:

- `'a' + 'b'` ⇒ `'ab'`
- `'abc' + 'de'` ⇒ `'abcde'`
- `[1,2] + [3]` ⇒ `[1,2,3]`
- `[1,2,3] + [4,5]` ⇒ `[1,2,3,4,5]`

Operatori logici principali su valori numerici

Si applicano a coppie di espressioni, e producono un valore *Booleano* (`True` o `False`):

`==` uguaglianza

`!=` disuguaglianza

`<` minore

`<=` minore o uguale

`>` maggiore

`>=` maggiore o uguale

Operatori principali su stringhe e liste

L'operatore di *slicing* si applica a variabili che contengono stringhe o liste, e restituisce una sottosequenza della sequenza originale

`var[a:b]` restituisce la sottosequenza (stringa o lista) contenuta nella variabile var, avente indici da a a b-1

`var[a:b:s]` restituisce la sottosequenza di indici a, a+s, a+2s, ..., fino all'elmento di indice b escluso

`var[:]` restituisce una *copia* dell'intera sequenza

Es.: se `x` contiene `'abcdefg'`:

- `x[2:5]` restituisce `'cde'`
- `x[0:7:2]` restituisce `'aceg'`
- `x[:]` restituisce `'abcdefg'`

Operatori logici principali su stringhe, liste e dizionari

`==` uguaglianza

`!=` disuguaglianza

`in` verifica la *presenza* di un elemento (una chiave nel caso dei dizionari), e restituisce `True` o `False`; esempi:

```
'z' in 'python'          ⇒  False
1 in [1,2,3]             ⇒  True
'b' in {'a':1, 'b':2}    ⇒  True
```

`not in` verifica l'*assenza* di un elemento; esempi:

```
'z' not in 'python'          ⇒  True
1 not in [1,2,3]             ⇒  False
'b' not in {'a':1, 'b':2}    ⇒  False
```

Operatori logici su valori *Booleani*

Oltre a espressioni logiche *semplici* (consistenti in un confronto tra una coppia di valori) è possibile scrivere espressioni logiche *composte*, combinando espressioni semplici (o altre espressioni composte), attraverso tre operatori logici corrispondenti alle congiunzioni *e* e *o* (*oppure*) e all'avverbio *non* del linguaggio naturale. Anch'esse producono il valore logico `True` o `False`:

- `and` (congiunzione di due espressioni logiche)
- `or` (disgiunzione inclusiva di due espressioni)
- `not` (negazione di una singola espressione)

Istruzioni principali del linguaggio Python

- Assegnamento
- Ingresso (tastiera) / uscita (schermo)
- Istruzioni *composte*
  - condizionale
  - iterativa

Istruzioni

Istruzione di assegnamento

variabile = espressione

Assegna a una variabile, il cui nome è scelto dal programmatore, il valore di un'espressione *qualsiasi*, che può contenere variabili tra gli operandi

I nomi delle variabili possono contenere un numero qualsiasi di caratteri alfabetici, di cifre, e di simboli _ (*underscore*), ma non possono cominciare con una cifra, né coincidere con le parole chiave del linguaggio

Istruzione di assegnamento

Esempi:

- `i = 1`
- `s = 'Python'`
- `L = [1,2,3]`
- `x = L[i]`
- `j = i`
- `i = i + 1`

Nota: l'espressione a destra del simbolo = viene calcolata *prima* dell'assegnamento alla variabile, quindi la conseguenza dell'ultima istruzione è incrementare di una unità il valore associato alla variabile `i` (assumendo che tale valore sia un numero)

Note sull'istruzione di assegnamento

Se a una variabile viene assegnato il valore di un'altra variabile, entrambe avranno lo stesso valore. Per esempio, le due istruzioni seguenti fanno sì che sia `m` che `w` abbiano come valore `5`

```
w = 5
m = w
```

Se successivamente a una delle due variabili viene assegnato un nuovo valore, il valore dell'altra *non cambia*. Proseguendo l'esempio precedente, dopo l'assegnamento

```
w = 10
```

il valore di `w` sarà `10`, ma il valore di `m` sarà ancora `5`

Istruzione di assegnamento

Il valore associato a una variabile può essere modificato da successive istruzioni di assegnamento

In ogni istante, il valore associato a una variabile è quello che le è stato assegnato dall'ultima istruzione di assegnamento in cui tale variabile è stata coinvolta

Note sull'istruzione di assegnamento

Se però una variabile contiene una lista o un dizionario, e il suo valore viene assegnato a un'altra variabile, ogni modifica ai *singoli elementi* della lista o del dizionario attraverso una successiva istruzione di assegnamento a una delle due variabili *modificherà anche il valore dell'altra variabile*. Per esempio, dopo le istruzioni:

```
a = [1,2,3]
b = a
a[0] = 10
```

sia il valore di `a` che quello di `b` saranno `[10,2,3]`

Note sull'istruzione di assegnamento

L'operatore di *slicing* applicato a una lista restituisce una *copia* della stessa lista. Può quindi essere usato per assegnare a una variabile una *copia* di una lista già assegnata a un'altra variabile. In questo caso, qualsiasi modifica degli elementi della lista attraverso un'istruzione di assegnamento a una delle due variabili *non modificherà l'altra variabile*. Esempio:

```
x = [1,2,3,4]
y = x[:]
y[0] = 10
```

Ora il valore di `y` è `[10,2,3,4]`, mentre quello di `x` è ancora `[1,2,3,4]`

Istruzioni di ingresso/uscita

Ingresso:

- `raw_input(`testo`)`
  stampa sullo schermo il messaggio testo, poi resta in attesa che l'utente scriva una *qualsiasi* sequenza di caratteri attraverso la tastiera, e dopo la pressione del tasto "Invio" restituisce tale sequenza all'interno di una stringa
- `raw_input()`
  come sopra, ma non stampa nessun messaggio

Nota: questa istruzione può essere usata per bloccare l'esecuzione di un programma fino alla pressione del tasto "Invio", per es.:

```
raw_input("Premi Invio per continuare")
```

Istruzioni di ingresso/uscita

Ingresso (tastiera):

- `input(`testo`)`
  testo dev'essere un'espressione avente come valore una stringa
  Questa istruzione stampa sullo schermo il messaggio contenuto nella stringa testo, poi resta in attesa che l'utente scriva un'espressione Python attraverso la tastiera, e dopo la pressione del tasto "Invio" ne restituisce il valore
- `input()`
  come sopra, ma non stampa nessun messaggio

Nota: se non viene inserita un'espressione valida, l'interprete produce un messaggio d'errore

Istruzioni di ingresso/uscita

Uscita:

- `print` espressione
  stampa sullo schermo il *valore* di espressione
  (un'espressione *qualsiasi*)
- `print` espressione$_1$, **...** , espressione$_n$
  stampa sullo schermo i *valori* di ciascuna espressione,
  separati da un carattere di spaziatura

Istruzione condizionale

```
if condizione:
    istruzione1
    ...
    istruzionen
```

Se l'espressione logica (*condizionale*) condizione è vera, viene eseguita la sequenza $istruzione_1$, ..., $istruzione_n$

Fare attenzione al rientro delle istruzioni della sequenza: ognuna di esse deve essere scritta con un rientro (di uno o più caratteri) rispetto alla parola chiave `if`, identico a quello delle altre. Se la sequenza fosse composta da una sola istruzione, si potrebbe anche scrivere:

```
if condizione:  istruzione
```

Istruzione condizionale

```
if condizione:
    istruzione1,1
    ...
    istruzione1,n
else:
    istruzione2,1
    ...
    istruzione2,m
```

Se una delle sequenze (o entrambe) fosse composta da una sola istruzione, si potrebbe anche scrivere nella stessa riga della parola chiave (`if` o `else`) corrispondente

Istruzione condizionale

```
if condizione:
    istruzione1,1
    ...
    istruzione1,n
else:
    istruzione2,1
    ...
    istruzione2,m
```

Le parole chiave `if` e `else` devono essere allineate a sinistra

Le istruzioni delle due sequenze devono essere scritte con un rientro di uno o più caratteri rispetto a `if` e `else`, e devono essere allineate a sinistra

Se condizione è vera, viene eseguita la sequenza $istruzione_{1,1}$, ..., $istruzione_{1,n}$, altrimenti viene eseguita la sequenza $istruzione_{2,1}$, ..., $istruzione_{2,m}$

Istruzione iterativa

`while` condizione`:`

istruzione$_1$

...

istruzione$_n$

Esegue ciclicamente la seguente operazione: se l'espressione logica condizione è vera, si esegue la sequenza istruzione$_1$, ..., istruzione$_n$; l'esecuzione termina non appena condizione risulta falsa

Per il rientro e l'allineamento delle istruzioni della sequenza valgono le stesse regole dell'istruzione `if`

Istruzione iterativa

`for` variabile `in` sequenza`:`

istruzione$_1$

...

istruzione$_n$

- sequenza è un'espressione il cui valore dev'essere una stringa o una lista
- variabile è un nome di variabile scelto dal programmatore

Le istruzioni istruzione$_1$, ..., istruzione$_n$ vengono eseguite per un numero di volte pari alla lunghezza di sequenza; nell'iterazione *i*-esima variabile ha come valore l'*i*-esimo elemento di sequenza

Istruzione iterativa

Sia per l'istruzione `while` che per `for`, nel caso in cui la sequenza di istruzioni da ripetere fosse composta da una sola istruzione si potrebbe anche scrivere:

`while` condizione`:` istruzione

`for` variabile `in` sequenza`:` istruzione

Istruzione iterativa

`for` variabile `in` sequenza`:`

    istruzione$_1$

    ...

    istruzione$_n$

Il valore di sequenza può anche essere un dizionario. In questo caso, in ognuna delle iterazioni variabile avrà come valore una delle *chiavi* del dizionario, in un ordine *qualsiasi* (non controllabile dal programmatore)

Le istruzioni `break` e `continue`

Si possono usare solo all'interno di un'istruzione iterativa (`while` o `for`)

- `break` causa l'immediata conclusione dell'esecuzione dell'istruzione iterativa
- `continue` causa il passaggio immediato all'iterazione successiva

Istruzioni *composte*

Le istruzioni condizionale e iterativa sono dette *composte*, perché possono contenere al loro interno istruzioni qualsiasi (anche altre istruzioni condizionali e iterative, che in questo caso si dicono "annidate")

Funzioni

Le funzioni sono sequenze d'istruzioni che svolgono una data operazione su un dato insieme di nessuno, uno o più valori (*argomenti*)

Ogni funzione è identificata da un *nome* simbolico (analogo ai nomi delle variabili)

L'esecuzione di una funzione su un certo insieme di argomenti può essere richiesta ogni volta che lo si desideri all'interno di un qualsiasi programma, attraverso un'opportuna espressione detta *chiamata*

L'esecuzione di una funzione può produrre come risultato un valore, che può essere usato dal programma chiamante

Funzioni

Funzioni

Le funzioni vengono usate nei linguaggi di alto livello per realizzare operazioni di utilità generale all'interno di qualsiasi programma, evitando ai programmatori di dover riscrivere le sequenze di istruzioni corrispondenti ogni volta che se ne presenti la necessità

Ogni linguaggio di programmazione comprende un insieme di funzioni *predefinite* che consentono di realizzare, per es., operazioni aritmetiche non disponibili sotto forma di operatori del linguaggio, come le funzioni trigonometriche o il calcolo dei logaritmi nel linguaggio Python

Chiamata di una funzione

L'esecuzione delle istruzioni di una funzione avviene attraverso un'espressione, detta *chiamata*, che può essere scritta come un'istruzione a sé stante, oppure (se la funzione restituisce un valore) come operando di un'espressione (inclusa un'espressione a destra del simbolo = in un'istruzione di assegnamento, per memorizzare in una variabile il valore restituito)

Sintassi di una chiamata a funzione:

nome_funzione (argomenti)

argomenti è una sequenza di espressioni separate da virgole, il cui numero deve coincidere con quello degli argomenti della funzione

La *libreria standard* di Python

Alcune funzioni predefinite possono essere usate direttamente in un programma Python, per es.:

- `len(`sequenza`)` restituisce il numero di elementi di una stringa, lista o dizionario
- `range`
  - `range(`a`)` restituisce la lista `[0,1, ... ,`a`-1]`, oppure una lista vuota se a < 1
  - `range(`a`,`b`)` restituisce la lista `[`a`,`a`+1, ... ,`b`-1]`, oppure una lista vuota se a ≥ b
  - `range(`a`,`b`,`s`)` restituisce una lista contenente i valori a, a+s, a+2s, ... fino al valore b escluso, oppure: una lista vuota se a = b, oppure a < b e s < 0, oppure a > b e s > 0; produce un messaggio d'errore se s = 0

La *libreria standard* di Python

L'insieme delle funzioni predefinite del linguaggio Python, che fanno parte di qualsiasi ambiente di programmazione, è detto *libreria standard*

La definizione di tali funzioni si trova in *file* aventi estensione `.py`

Un elenco esaustivo delle funzioni di libreria si trova nel documento "La libreria di riferimento di Python" (di Guido van Rossum), disponibile nel sito Web `http://www.python.it`

La *libreria standard* di Python

- `str(`espressione`)` restituisce una stringa rappresentante il valore di espressione
- `int(`stringa`)` e `float(`stringa`)`
  restituiscono il numero intero e reale rappresentato da stringa (se questa è interpretabile come un numero)
- `int(`numero`)`
  restituisce la parte intera di un'espressione numerica
- `float(`numero`)`
  converte in un valore reale un'espressione numerica
- `abs(`numero`)` restituisce il valore assoluto di numero

La *libreria standard* di Python

- `min(`lista`)` e `max(`lista`)` restituiscono il minimo e il massimo valore contenuto in una lista di numeri
- stringa`.split()` restituisce una lista di stringhe ottenute suddividendo i caratteri di stringa in corrispondenza dei caratteri di spaziatura (che vengono rimossi); es.:
  ```
  s = "Questa è una frase."
  s.split() ⇒ ['Questa', 'è', 'una', 'frase.']
  ```
- stringa`.split(`caratteri`)`
  dove caratteri è un'espressione avente per valore una stringa: come sopra, ma la suddivisione avviene in corrispondenza della sequenza caratteri; es.:
  ```
  s = "uno;due;tre"
  s. split(';') ⇒ ['uno', 'due', 'tre']
  ```

Principali funzioni della libreria `math`

- `cos(x), sin(x), tan(x)`
  coseno, seno, tangente
- `acos(x), asin(x), atan(x)`
  arco-seno/-coseno/-tangente
- `radians(x), degrees(x)`
  conversione gradi/radianti
- `exp(x)` e^x
- `log(x)` $\log_e x$
- `log10(x)` $\log_{10} x$
- `log(x,b)` $\log_b x$
- `pow(x,y)` x^y
- `sqrt(x)` $\sqrt{x}$

La *libreria standard* di Python

La maggior parte delle altre funzioni di libreria richiedono un'istruzione particolare prima di poter essere usate. Per usare le funzioni definite in un *file* `nome.py`:

```
from nome import *
```

Per es., i file `math.py` e `random.py` contengono funzioni matematiche e funzioni per la generazione di numeri casuali. Per usare tali funzioni sono necessarie le istruzioni

```
from math import *
```

```
from random import *
```

Principali funzioni della libreria `math`

Nel *file* di libreria `math.py` sono anche definite due variabili che contengono i valori delle costanti matematiche π ed e (numero di Eulero):

- `pi`
- `e`

Principali funzioni della libreria `random`

- `random()`
  genera un numero reale da una distribuzione uniforme nell'intervallo [0,1]
- `uniform(a,b)`
  genera un numero reale da una distribuzione uniforme nell'intervallo [a,b] (a e b sono numeri reali)
- `randint(a,b)`
  genera un numero intero casuale compreso tra a e b (a e b devono essere numeri interi)
- `gauss(m,s)`
  genera un numero casuale da una distribuzione Gaussiana con media m e deviazione standard s

Definizione di nuove funzioni

È possibile definire nuove funzioni, attraverso la seguente istruzione:

```
def nome_funzione (parametri):
    istruzione1
    ...
    istruzionen
```

parametri è una sequenza di nomi di variabili (separati da virgole), alle quali saranno assegnati i valori degli *argomenti* della funzione, quando questa sarà *chiamata*

Tutte le istruzioni della funzione devono essere scritte con un rientro a destra di almeno un carattere, e devono essere allineate tra loro

Principali funzioni della libreria `random`

- `choice(`sequenza`)`
  restituisce un elemento di sequenza (stringa o lista) scelto in modo casuale
- `shuffle(`lista`)`
  permuta in modo casuale gli elementi di una lista (*modificando* la lista originale)
- `sample(`sequenza`,`k`)`
  restituisce una lista contenente k elementi di sequenza (stringa, lista o dizionario) scelti in modo casuale (nel caso di un dizionario, vengono restituite le *chiavi*)

L'istruzione `return`

Dopo l'esecuzione dell'ultima istruzione di una funzione, l'interprete riprende l'esecuzione delle istruzioni del programma chiamante

In un qualsiasi punto di una funzione è anche possibile usare l'istruzione `return`

- `return`
  termina l'esecuzione della funzione e restituisce il controllo al programma chiamante
- `return` espressione
  come sopra, restituendo il valore di espressione al programma chiamante

Funzioni e variabili *locali*

Sia i parametri che le variabili eventualmente definite in una funzione sono *locali*, cioè sono "visibili" solo dalle istruzioni della stessa funzione, e non da altre funzioni o dal programma chiamante

A sua volta, una funzione non può accedere alle variabili usate dal programma chiamante

È quindi sempre possibile usare variabili con lo stesso nome in un programma e in funzioni diverse, senza nessun conflitto o ambiguità

Liste e dizionari come argomenti di funzioni

Esempio:

```
def prova (lista):
  lista[0] = 'a'
```

Se la funzione `prova` viene chiamata passandole come argomento una variabile contenente una lista:

```
x = [1,2,3]
prova(x)
```

dopo la chiamata il valore di `x` sarà `['a',2,3]`

Liste e dizionari come argomenti di funzioni

Quando una funzione viene chiamata, i valori degli argomenti idicati nella chiamata vengono assegnati ai suoi parametri, con le stesse modalità di un'istruzione di assegnamento

Questo implica che se un argomento è una variabile che contiene una lista o un dizionario, la modifica dei suoi elementi da parte delle istruzioni della funzione si rifletterà anche sui valori della corrispondente variabile del programma chiamante

Gestione dei *file*

Gestione dei *file*

- Tipi di *file*:
  - sequenze di *byte*
  - sequenze di caratteri (*file* di testo)
- Operazioni eseguibili sui *file* di testo:
  - lettura
  - scrittura, con tre possibilità:
    - creazione di un nuovo *file*
    - aggiunta di dati al termine di un *file* esistente
    - sostituzione del contenuto di un *file* esistente (sovrascrittura)

Apertura di un *file*

Si usa la dunzione predefinita `open`

`open(`nome_file`,` modalità`)`
restituisce un valore contenente informazioni sul *file*, che dovrà essere assegnato a una variabile, la quale dovrà essere usata per eseguire qualsiasi operazione sul *file*

- nome_file: stringa contenente il nome del *file* (*pathname*)
- modalità:
  - `'r'` lettura da un *file* esistente
  - `'w'` scrittura: crea un nuovo *file* (se non esiste nessun *file* di nome nome_file) o sovrascrive un *file* esistente
  - `'a'` scrittura: aggiunta di dati a un *file* esistente

Esempio:

```
f = open ('dati.txt', 'r')
```

Gestione dei *file*

Procedura per l'accesso ai *file*:

1) apertura di un *file* in modalità di lettura o di scrittura
2) esecuzione di operazioni di lettura *oppure* di scrittura (a seconda della modalità scelta in fase di apertura)
3) chiusura del *file*

Chiusura di un *file*

Si usa la funzione predefinita `close`

variabile_file`.close()`

chiude il *file* associato a variabile_file (non sarà più possibile eseguire operazioni di lettura e scrittura su di esso, a meno di riaprirlo con la funzione `open`)

Esempio:

```
f.close()
```

Lettura da un *file*

Attraverso alcune funzioni predefinite è possibile acquisire da un *file* di testo sequenze di uno o più caratteri, di una o più righe, o il suo intero contenuto

Ogni operazione di lettura viene eseguita:

- a partire dal primo carattere del *file*, se si tratta della prima operazione di lettura dopo l'apertura
- a partire dal carattere successivo all'ultimo carattere acquisito dalla precedente operazione di lettura

I dati acquisiti da ciascuna di tali funzioni vengono di norma assegnati a una variabile, o vengono usati all'interno di un'espressione

Lettura da un *file*: `read`

La funzione `read` consente anche di acquisire un dato numero di caratteri da un file, invece che tutti i caratteri restanti, indicando tale numero come argomento:

variabile_file`.read(`n`)`

dove n è un'espressione il cui valore dev'essere un numero intero positivo

Lettura da un *file*: `read`

La funzione `read` restituisce una stringa contenente:

- l'intero contenuto del *file*, se viene chiamata subito dopo l'apertura
- i caratteri non ancora acquisiti dalle operazioni di lettura precedenti (se tutti i caratteri sono già stati acquisiti, restituisce una stringa vuota)

Sintassi:

variabile_file`.read()`

dove variabile_file è la variabile associata al *file*

Lettura da un *file*: `readline`

La funzione `readline` restituisce una stringa contenente una singola riga di un *file*, cioè una sequenza di caratteri conclusa da un *newline* ("andata a capo", indicata con la sequenza di *escape* `'\n'`), a partire dal caratterere successivo rispetto all'ultimo acquisito da eventuali operazioni di lettura precedenti. Note:

- se una riga è vuota, `readline` restituisce la stringa `'\n'`
- se tutti i caratteri sono già stati acquisiti, `readline` restituisce una stringa vuota

Sintassi:

variabile_file`.readline()`

Lettura da un *file*: `readline`

La funzione `readline` consente anche l'acquisizione di un dato numero di caratteri da una riga di un *file*, indicando tale numero come argomento:

variabile_file`.readline(`n`)`

Scrittura su un *file*: `write`

La funzione `write` consente di scrivere in un *file* una stringa di caratteri, indicata come argomento:

variabile_file`.write(`stringa`)`

- stringa dev'essere un'espressione il cui valore sia una stringa di caratteri
- la stringa viene scritta al termine del *file* (cioè dopo gli eventuali caratteri scritti da precedenti operazioni di scrittura)

Lettura da un *file*: `readlines`

La funzione `readlines` restituisce l'intero contenuto di un *file* (o della parte non ancora acquisita dopo le precedenti operazioni di lettura) memorizzato in una lista di stringhe, ognuna delle quali contiene una singola riga (incluso il carattere di *newline* `'\n'`):

variabile_file`.readlines()`

Tracciamento di grafici: la libreria `matplotlib`

Introduzione

Libreria di funzioni per il tracciamento di grafici (richiede l'installazione della libreria matematica `numpy`)

- Installazione
  - `numpy`
    http://www.numpy.org/ (documentazione)
    http://sourceforge.net/projects/numpy/files/
  - `matplotlib`
    http://matplotlib.org/index.html (documentazione)
    http://matplotlib.org/downloads.html
- Per usare le funzioni di `matplotlib`:
  `from pylab import *`
  (nota: `matplotlib` importa a sua volta `math`)

`matplotlib`: funzioni principali

- `plot(x,y)` costruisce un grafico, unendo con un segmento di retta ogni coppia di punti adiacenti le cui coordinate si trovano nelle liste `x` e `y` (i valori di `x` dovrebbero essere ordinati in senso crescente)

- `show()` mostra il grafico in una finestra dedicata (nota: se questa funzione è chiamata dalla *shell*, può mantenerla bloccata finché non si chiude la finestra del grafico)