Breve introduzione al linguaggio Python

Marco Liverani*

Maggio 2019

1 Introduzione

Il linguaggio Python, inventato da Guido van Rossum a partire dal 1991, è un linguaggio di scripting object oriented, che supporta pienamente la programmazione strutturata, progettato appositamente per garantire una semplicità estrema nella codifica del programma e al tempo stesso una grandissima leggibilità del codice. Python è un linguaggio che si presta molto bene per usi didattici, ma al tempo stesso è un linguaggio potentissimo, con cui è possibile scrivere software estremamente complesso, tanto da essere adottato tra i linguaggi di programmazione standard per l'implementazione di servizi da aziende come Google e la NASA, oltre a moltissime altre sia in Italia che all'estero.

Il nome del linguaggio è un omaggio ai comici britannici Monty Python, di cui Guido van Rossum era un appassionato.

Oggi Python è giunto alla versione 3.x (3.7.2 nel momento in cui sono state scritte queste note), che introduce numerose importanti differenze rispetto alla versione 2.x. Per questo motivo nelle pagine seguenti faremo riferimento esclusivamente alla versione 3.x del linguaggio.

Questa brevissima guida di base al linguaggio non ha la pretesa di sostituire un buon testo sull'argomento, né di essere un manuale esaustivo sulle numerosissime caratteristiche del linguaggio. Si rivolge a chi, con qualche esperienza di programmazione pregressa, vuole avvicinarsi al linguaggio Python ed esplorarne alcune delle caratteristiche che ho ritenuto più utili o rilevanti di altre. È un punto di partenza, che rende poi necessario un approfondimento, ad esempio con i testi e le risorse disponibili in rete, citate nei riferimenti bibliografici al termine di questa guida.

2 L'interprete del linguaggio

Per eseguire un programma in Python è necessario installare l'interprete del linguaggio sul computer su cui si intende eseguire il programma. Esistono numerosissime versioni dell'interprete Python per diversi sistemi operativi; il programma di installazione dell'interprete Python può essere scaricato gratuitamente dal sito `http://www.python.org`.

Nelle pagine seguenti faremo riferimento all'interprete Python utilizzato in ambiente Linux e Apple OS X, ma le differenze con l'ambiente Microsoft Windows sono veramente minime.

*E-mail: liverani@mat.uniroma3.it; ultima modifica: 12 maggio 2019

Versione dell'interprete Python

Il nome del programma eseguibile corrispondente alla versione 3.*x* dell'interprete del linguaggio è tipicamente «`python3`», mentre il nome «`python`» è riservato alla versione 2.*x* dell'interprete. Per fugare ogni dubbio si può eseguire l'interprete con l'opzione «`--version`» per visualizzare il numero di versione:

```
$ python --version
Python 2.7.10
$ python3 --version
Python 3.7.2
```

Editor di testo Atom

Per scrivere un programma in Python è necessario disporre di un editor di testo, come il programma *open source* **Atom**, che è possibile scaricare gratuitamente per numerosi sistemi operativi (Microsoft Windows, Linux, Apple OS X ed altri ancora) dal sito `https://atom.io`.

Esecuzione di un programma in Python

L'interprete Python può essere utilizzato per eseguire dei programmi salvati su un file di testo (tipicamente il nome dei file che contengono programmi scritti in Python, è caratterizzato dall'estensione «`.py`»), lanciando l'interprete seguito, sulla riga di comando, dal nome del programma:

```
$ python3 nomeprogramma.py
```

Esecuzione dell'interprete Python in modalità interattiva

In alternativa è possibile eseguire l'interprete in modalità interattiva: in questo modo viene presentato un "prompt" dell'interprete ed è possibile digitare singole istruzioni che vengono immediatamente eseguite dall'interprete stesso:

```
$ python3
Python 3.7.2 (v3.7.2:9a3ffc0492, Dec 24 2018, 02:44:43)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Ciao a tutti!")
Ciao a tutti!
>>> a=7
>>> b=5
>>> c=a+b
>>> print(c)
12
>>> if a>b:
...     print("a e' maggiore di b")
... else:
...     print("a e' minore o uguale a b")
...
a e' maggiore di b
>>> exit()
$
```

In modalità interattiva è possibile utilizzare la funzione «`help(...)`» per visualizzare delle istruzioni sintetiche relative alle funzioni del linguaggio. La funzione «`dir(...)`» visualizza invece l'elenco degli attributi e delle componenti dell'oggetto passato come argomento alla funzione stessa.

Per uscire dall'interprete dei comandi eseguito in modalità interattiva, si deve richiamare la funzione «`exit()`», oppure si può digitare la sequenza CTRL – D. Questa sequenza di caratteri corrisponde al terminatore di file (EOF, *end of file*).

3 Alcune regole sintattiche

La sintassi del linguaggio Python è molto semplice, ma presenta alcune differenze rispetto agli standard adottati in molti altri linguaggio di programmazione (C, Java, Javascript, ecc.). In particolare è opportuno mettere in evidenza tre aspetti principali.

Python è un linguaggio che supporta pienamente le regole e i costrutti della programmazione strutturata. Per questo motivo è necessario identificare i blocchi che costituiscono le istruzioni che vengono eseguite nell'ambito di una determinata struttura iterativa o nei due rami di una struttura condizionale. Quasi tutti i moderni linguaggi di programmazione fanno uso delle parentesi graffe per circoscrivere l'inizio e la fine di un blocco di istruzioni; le parentesi graffe in questi linguaggi possono essere nidificate, in modo da includere blocchi afferenti ad una determinata struttura algoritmica all'interno di un'altra struttura algoritmica (es.: due cicli nidificati uno dentro l'altro).

Indentazione del codice per identificare i blocchi di istruzioni

In linguaggio Python non si utilizzano le parentesi graffe per delimitare blocchi di istruzioni: si utilizza l'indentazione del codice. Ciò che in altri linguaggi è un elemento facoltativo, utile per lo più per migliorare la leggibilità del codice, in Python, che valorizza moltissimo la leggibilità del codice, diventa un elemento sintattico indispensabile, non opzionale.

La nidificazione di blocchi di istruzioni si ottiene aumentando il livello di indentazione delle istruzioni che fanno parte di un determinato blocco. L'istruzione che precede un determinato blocco di istruzioni (ad esempio una condizione o un controllo per una struttura iterativa) termina con il carattere "due-punti". Ad esempio:

```
1  while i <= n:
2      if a[i] == 1:
3          print(i)
4          cont = cont+1
5          j = i
6          while j <= n:
7              a[j] = 0
8              j = j+i
9      i = i+1
10 print(cont)
```

Nell'esempio precedente sono presenti tre strutture algoritmiche nidificate una dentro l'altra. La prima è una struttura iterativa controllata dall'istruzione «`while`» a riga 1; il blocco interno a questa struttura va da riga 2 a riga 9. All'interno della struttura iterativa è presente una struttura condizionale controllata dall'istruzione «`if`» a riga 2; tale struttura contiene un blocco di istruzioni che va da riga 3 a riga 8; l'istruzione a riga 9 è interna al blocco della struttura iterativa, ma esterno al blocco della struttura condizionale. Infine, all'interno della struttura condizionale è presente un'altra struttura iterativa controllata dall'istruzione «`while`» di riga 6, che contiene le istruzioni alle righe 7 e 8.

Terminatore delle istruzioni

In quasi tutti i moderni linguaggi di programmazione si usa il "punto-e-virgola" come terminatore delle istruzioni; il linguaggio Python non fa eccezione, ma, siccome si privilegia una modalità di scrittura del codice in cui su una stessa riga di programma viene riportata una sola istruzione, il "punto-e-virgola" diventa opzionale: può essere usato, ma è indispensabile soltanto per separare istruzioni diverse riportate sulla stessa riga di programma. Altrimenti, il terminatore delle istruzioni è il semplice carattere di "a-capo".

Funzioni e librerie, metodi, oggetti, classi e moduli

Infine il linguaggio Python è un linguaggio le cui istruzioni sono costituite quasi esclusivamente da quelle necessarie per definire le strutture algoritmiche (iterazioni, condizioni) e quelle utilizzate per definire funzioni. La maggior parte delle istruzioni di un programma Python sono invece costituite da chiamate a *funzioni* definite all'interno di specifiche *librerie* o chiamate a *metodi* di *oggetti*, definiti in appositi *moduli* che contengono le definizioni di *classi* specifiche.

Questo significa che spesso le istruzioni del linguaggio sono seguite da una coppia di parentesi che delimitano i parametri passati come argomento di una funzione, anche nel caso in cui tali parametri non siano presenti affatto (es.: «`print("Ok!")`», «`input(a)`», «`exit()`»).

In altri casi la funzione è un metodo di una particolare classe di oggetti e dovrà essere invocato utilizzando il "punto" per riferirsi all'oggetto a cui si intende applicare il metodo (es.: «`A.insert(17)`»).

Commenti nel codice del programma

In Python, come nella maggior parte dei linguaggi di scripting (es.: Bash, Perl, ecc.) il carattere "`#`" indica un *commento*: i caratteri che seguono il simbolo "`#`" vengono ignorati dall'interprete, fino alla fine della riga. È possibile inserire commenti che si estendono per più di una riga, inserendo il simbolo "`#`" all'inizio di ciascuna riga, oppure utilizzando la sequenza «`"""`» (tre volte le doppie virgolette) all'inizio del commento e un'altra sequenza di tre doppi apici alla fine del commento, anche dopo più di una riga.

4 Variabili, input/output e strutture dati di base

In linguaggio Python non è necessario dichiarare variabili e le variabili sono "non tipate" (ossia non è necessario dichiarare a priori il tipo di dato che contengono). I nomi delle variabili seguono la tipica sintassi di ogni linguaggio di programmazione: sono stringhe alfanumeriche prive di spazi, che non possono iniziare con un numero; ad esempio: `a`, `B`, `nome`, `F24`, `voto_esame`, ecc.

Per assegnare un valore ad una variabile si deve usare l'operatore di assegnazione, rappresentato dal simbolo "=". Ad esempio:

```
a = 5
b = a+1
c = b/a
nome = "Marco"
```

Naturalmente i valori che vengono assegnati ad una variabile sono invece di tipo ben determinato; il tipo del dato assegnato ad una variabile viene utilizzato applicando gli operatori: ad esempio due numeri possono essere sommati tra loro, sottratti l'uno dall'altro, divisi e moltiplicati fra di loro, mentre non è possibile fare altrettanto con due stringhe di caratteri.

Tipo	Descrizione	Tipo	Descrizione
`int`	interi	`long`	interi lunghi
`float`	numeri in virgola mobile	`complex`	numeri complessi
`bool`	booleani	`str`	stringhe di caratteri

Tabella 1: I tipi di dato in Python

I tipi di dato in Python sono analoghi a quelli già visti in molti altri linguaggi di programmazione e per semplicità sono riportati in Tabella 1. Le parole chiave che identificano i tipi possono essere utilizzate per specificare o convertire un dato in un tipo ben preciso (es.: «`a = int(7/2)`» assegna alla variabile `a` il valore 3). In modalità interattiva è possibile utilizzare la funzione «`type(...)`» per visualizzare il tipo di un determinato oggetto o classe di oggetti del linguaggio (variabile, funzione, metodo, oggetto, ecc.).

Operatori aritmetici

Gli operatori aritmetici con cui è possibile operare sulle variabili e le espressioni numeriche sono molto simili a quelle di altri linguaggi: oltre agli operatori di somma (+), sottrazione (-), prodotto (*) e divisione (/) sono anche presenti l'elevamento a potenza indicato con «**» (ad esempio a^n si rappresenta con «`a**n`») e il modulo (resto della divisione intera) indicato con «%» (ad esempio «`5%2`» restituisce il valore 1). È interessante osservare che gli interi lunghi possono assumere valori con un numero di cifre enorme, che dipende solo dalle risorse di calcolo della macchina. Ad esempio:

```
>>> 2**250
1809251394333065553493296640760748560207343510400633813116524
750123642650624
>>> len(str(2**250))
76
```

Nell'esempio precedente, per calcolare il numero di cifre ottenuto abbiamo trasformato il numero in una stringa con la funzione «`str()`» e ne abbiamo contato i caratteri mediante la funzione «`len()`».

Formattazione delle stringhe, il metodo **format**

Attraverso il metodo «`format()`» le stringhe possono essere composte con il valore presente in altre variabili, formando una stringa unica più complessa. Supponiamo di disporre di due variabili, `a` e `b`, che contengono rispettivamente il nome di una squadra di calcio e i punti acquisiti in campionato. Per comporre una stringa utilizzando anche i valori di queste due variabili si può usare la seguente espressione:

```
>>> a = "Genoa"
>>> b = 30
>>> s = "La squadra {0} ha {1} punti".format(a, b)
>>> s
'La squadra Genoa ha 30 punti'
```

Il metodo «`format()`» sostituisce nella stringa la sequenza "`{0}`" con il valore della prima variabile indicata come argomento e la sequenza "`{1}`" con la seconda variabile, e così via. In questo modo alla variabile `s` viene assegnato il valore "`La squadra Genoa ha 30 punti`".

Input e output: **input** e **print**

Le funzioni «`input()`» e «`print()`» consentono, rispettivamente, di acquisire in input dei dati e di memorizzarli nelle variabili del programma e di visualizzare in output un messaggio o il valore di una variabile o di un'espressione.

La funzione «`input()`» permette di acquisire in input un valore inserito dall'utente mediante la tastiera del computer (sarebbe più corretto dire: dal canale di *standard input*) e di memorizzarlo in una variabile.

La funzione «`print()`» consente, al contrario, di visualizzare sullo schermo del computer (o meglio: di inviare sul canale di *standard output*) una frase, il valore di una variabile, ecc. Ad esempio:

```
>>> a = input("Inserisci un numero: ")
Inserisci un numero: 57
>>> print(a)
'57'
>>> a = int(input("Inserisci un numero: "))
Inserisci un numero: 68
>>> print(a)
68
```

Come si può notare dall'esempio precedente, la funzione «`input()`» restituisce sempre una stringa di testo; se si desidera interpretare il valore inserito in input dall'utente come un numero, occorre utilizzare la funzione «`int()`», «`float()`» o «`long()`», per trasformare la stringa nel numero corrispondente.

La funzione «`print()`» aggiunge automaticamente al termine della stringa visualizzata un carattere di "a-capo"; in questo modo la successiva chiamata alla funzione «`print()`» visualizzerà l'output sulla riga seguente dello schermo del computer o del terminale dell'utente. Se invece si preferisce che l'output della successiva chiamata alla funzione «`print()`» venga visualizzato sulla stessa riga, bisogna aggiungere come parametro della funzione anche l'opzione «`end="..."`». Il valore specificato con tale parametro opzionale rappresenta il carattere o la stringa da visualizzare al termine dell'output prodotto dalla funzione «`print()`».

L'opzione **end** della funzione **print**

Se alla funzione «`print()`» vengono passati più parametri come argomento (ad esempio diverse stringhe o variabili), verrà visualizzato automaticamente uno spazio tra un valore e l'altro. Se non si desidera che venga visualizzato lo spazio aggiuntivo, si deve specificare il carattere o la sequenza di caratteri da visualizzare al posto dello spazio, con l'opzione «`sep`». Ad esempio:

L'opzione **sep** della funzione **print**

```
>>> nome="Marco"
>>> print("Ciao", nome, "come stai?")
Ciao Marco come stai?
>>> print("Ciao", nome, "come stai?", sep="###")
Ciao###Marco###come stai?
```

Strutture dati

Oltre alle variabili, in linguaggio Python sono disponibili anche numerose **strutture dati**, riepilogate in Tabella 2. Ciascuna struttura dati è una classe di oggetti che dispone di metodi specifici per operare sulla struttura stessa.

Liste, "**list**"

Le **liste** sono le strutture dati più semplici e consentono di rappresentare delle sequenze di dati. In Python con un oggetto `list` si rappresentano sia le liste che gli array: non esiste una distinzione, dal momento che anche gli elementi di una lista sono indirizzabili direttamente mediante un indice che ne indica la posizione nella

Tipo	Descrizione
`list`	lista
`tuple`	tupla (lista non modificabile)
`set`	insieme
`frozenset`	insieme non modificabile
`dict`	dizionario
`file`	file su memoria di massa

Tabella 2: Strutture dati in Python

struttura dati (iniziando dalla prima posizione che ha indice 0), come avviene con gli array in altri linguaggi di programmazione. Una lista può essere definita esplicitamente elencandone gli elementi tra parentesi quadre, separati da virgole, oppure mediante il "costruttore" `list()`. Ad esempio:

```
a = [10, 56, 11, 43, 49]
b = list()
```

È possibile ottenere l'elemento della lista memorizzato nella posizione di indice *i* utilizzando l'espressione «`a[i]`», così come si usa fare per gli elementi di un array in altri linguaggi di programmazione; ricordiamo che il primo elemento della lista ha indice 0. Sugli oggetti di tipo `list` è possibile operare con i seguenti metodi:

Metodi per operare sulle liste

- `append`: aggiunge un elemento alla lista; es.: «`a.append(17)`»;
- `clear`: svuota completamente la lista, eliminando ogni elemento eventualmente presente; es.: «`a.clear()`»;
- `copy`: copia il contenuto di una lista in un'altra lista; es.: per copiare la lista `a` sulla lista `c`: «`c = a.copy()`»;
- `count`: conta il numero di occorrenze di un determinato valore in una lista (in una lista uno stesso valore può essere memorizzato più volte); es.: «`a.count(17)`»;
- `index`: restituisce l'indice della posizione nella lista della prima occorrenza del valore specificato come argomento; restituisce un errore se il valore non è presente; es.: «`a.index(17)`»;
- `insert`: inserisce un elemento prima della posizione specificata come primo argomento; es.: «`a.insert(2,36)`» inserisce il numero 36 nella lista `a` prima dell'elemento con indice 2, ossia prima del terzo elemento della lista;
- `pop`: estrae dalla lista (e rimuove dalla lista) l'elemento nella posizione specificata; se non si indica nessuna posizione, viene estratto l'ultimo elemento della lista; es.: «`a.pop(2)`» estrae il terzo elemento dalla lista, mentre con «`a.pop()`» si estrae l'ultimo elemento della lista;
- `remove`: rimuove dalla lista la prima occorrenza del valore specificato come argomento; es.: «`a.remove(17)`»;
- `reverse`: inverte l'ordine degli elementi della lista; es.: «`a.reverse()`»;

- `sort`: ordina in ordine crescente (numerico o lessicografico) gli elementi della lista; es.: «`a.sort()`».

Insiemi, "**set**", operazioni sugli insiemi

Molto simili sono i metodi che si applicano ad una struttura dati di tipo `set`, gli **insiemi**; la principale differenza tra un insieme ed una lista è che un insieme non può contenere valori duplicati. Inoltre sugli insiemi è possibile applicare degli operatori di tipo insiemistico: l'operatore «|» rappresenta l'unione tra insiemi, l'operatore «&» l'intersezione, l'operatore «-» rappresenta la differenza tra insiemi e l'operatore «^» la differenza simmetrica. Ad esempio:

```
>>> a = {'cane', 'gatto', 'topo'}
>>> b = set()
>>> b.append('leone')
>>> b.append('gatto')
>>> b.append('tigre')
>>> a | b   # unione
{'leone', 'tigre', 'cane', 'gatto', 'topo'}
>>> a & b   # intersezione
{'gatto'}
>>> a - b   # differenza
{'cane', 'topo'}
>>> a ^ b   # differenza simmetrica
{'tigre', 'cane', 'topo', 'leone'}
```

È possibile verificare se un insieme è contenuto in un altro (è sottoinsieme di un altro insieme) con il metodi «`issubset(...)`» e verificare se contiene un altro insieme con il metodo «`issuperset(...)`»; entrambi i metodi restituiscono un valore booleano. Ad esempio:

```
>>> a = {2, 4, 6, 8}
>>> b = {1, 3, 5}
>>> c = {2, 6}
>>> b.issubset(a)
False
>>> c.issubset(a)
True
>>> a.issuperset(c)
True
```

Dizionari, array associativi

I dizionari (`dict`) sono delle strutture dati molto potenti ed estremamente utili per la rappresentazione di array associativi, ossia delle liste composte da coppie di elementi, di cui uno è la *chiave* della voce del dizionario e l'altro è il valore associato a tale chiave. Per identificare un determinato valore presente nel dizionario si usa la chiave corrispondente. Ad esempio potremmo costruire un dizionario per memorizzare le valutazioni che abbiamo ottenuto agli esami:

```
>>> voto = {'IN110':30, 'AL420':27, 'AM110':22, 'IN440':28}
>>> voto['AL420']
27
```

Con i metodi «`keys()`» e «`values()`» applicati ad un dizionario si ottengono rispettivamente la lista delle chiavi e la lista dei valori associati a tali chiavi.

```
>>> voto.keys()
dict_keys(['IN110', 'AL420', 'AM110', 'IN440'])
>>> voto.values()
dict_values([30, 27, 22, 28])
```

Con la funzione «`del()`» è possibile rimuovere tutti gli elementi di un dizionario corrispondenti ad una determinata chiave; ad esempio:

```
>>> del(voto['IN110'])
>>> voto
{'AL420': 27, 'AM110': 22, 'IN440': 28}
```

5 Strutture algoritmiche

Come abbiamo già detto in precedenza, il linguaggio Python supporta pienamente il paradigma della programmazione strutturata, consentendo di implementare le strutture algoritmiche sequenziale, condizionale e iterativa.

La **struttura condizionale** viene codificata con l'istruzione «`if-else`» che, come in ogni altro linguaggio di programmazione strutturato, permette di valutare una condizione logica e di proseguire con l'esecuzione di determinate istruzioni oppure di altre, in base al valore (vero/falso) della condizione.

Struttura condizionale, l'istruzione "**if**"

La condizione viene espressa di seguito alla parola chiave «`if`», senza doverla delimitare mediante parentesi tonde (come in C o in Java); deve essere utilizzato il carattere due-punti «`:`» per terminare la condizione. Il blocco di istruzioni che deve essere eseguito se la condizione risulta vera, viene riportato nelle righe seguenti, indentando ogni istruzione. Dopo il blocco di istruzioni eseguite nel caso in cui la condizione sia verificata, può essere utilizzata l'istruzione «`else`», pure questa seguita da due-punti, per indicare le istruzioni che dovranno essere eseguite nel caso in cui la condizioni risulti falsa. Anche in questo caso il blocco di istruzioni da eseguire deve essere indentato rispetto alla parola chiave «`else`». Per esprimere le condizioni è possibile utilizzare le parentesi e gli operatori logici «`and`», «`or`» e «`not`». Ad esempio:

```
1 if a>b and (c<17 or c>26):
2     a = a+1
3     b = b-1
4 else:
5     a = a-1
6     b = b+1
7 c = a*b
```

Come si dovrebbe intuire dall'indentazione del codice, le istruzioni a riga 2 e 3 vengono eseguite solo se la condizione espressa a riga 1 è verificata; le istruzioni alle righe 5 e 6 vengono eseguite, al contrario, solo se la condizione è falsa. L'istruzione a riga 7 è fuori dai due blocchi e viene eseguita in ogni caso, a prescindere dal valore logico della condizione.

Per le **strutture iterative** il linguaggio Python ci mette a disposizione l'istruzione «`while`», che permette di esprimere una condizione che, se verificata, fa sì che venga eseguito iterativamente il blocco di istruzioni fino a quando la condizione non risulterà falsa. Anche in questo caso la condizione viene terminata dal carattere due-punti e il blocco di istruzioni da eseguire fintanto che la condizione risulta vera, viene indentato rispetto all'istruzione «`while`». Ad esempio:

L'istruzione "**while**"

```
1 while mx != my :
2     if mx < my :
3         mx = mx+x
4     elif my < mx :
5         my = my+y
6 print(mx)
```

Le istruzioni dalla 2 alla 5 vengono eseguite fintanto che risulta vera la condizione $mx \neq my$; l'istruzione «`print(mx)`» a riga 6 viene eseguita invece una volta soltanto, al termine dell'iterazione, quando la condizione risulta falsa.

Istruzione "**for**"

L'istruzione «`for`» permette di ripetere un determinato blocco di istruzioni, assegnando di volta in volta ad una variabile di controllo un valore diverso preso da un intervallo o da una struttura dati.

Intervalli, "**range**"

Gli **intervalli** in particolare possono essere espressi mediante la funzione «`range()`» con cui è possibile indicare gli estremi dell'intervallo o soltanto l'estremo superiore, assumendo che il valore dell'estremo inferiore sia 0. L'estremo inferiore è contenuto nell'intervallo, mentre l'estremo superiore è escluso. Ad esempio «`range(1,10)`» indica la sequenza di numeri naturali $1,2,\ldots,9$. Con l'istruzione «`range(10)`» si indica invece la sequenza di numeri $0,1,2,\ldots,9$.

L'istruzione `for` può quindi essere utilizzata in questo modo, per produrre, ad esempio, la stampa dei numeri interi da 0 a 9:

```
1 for x in range(10):
2     print(x)
```

La variabile `x` assume ad ogni iterazione un diverso valore nell'intervallo $0,\ldots,9$. La variabile utilizzata per controllare l'esecuzione di un ciclo con l'istruzione `for` può anche assumere come valori i diversi elementi di un insieme o di una lista o di qualsiasi altra struttura dati, tra quelle viste nelle pagine precedenti, come riportato nei seguenti esempi:

```
1  a = {'cane', 'gatto', 'topo'}
2  for animale in a:
3      print(animale)
4  esami = {'IN110':27, 'AM110':30, 'AL110':24}
5  print("Esami superati:", end=" ")
6  for corso in esami.keys():
7      print(corso, end=" ")
8  print()
9  print("Valutazioni per gli esami:")
10 for corso in esami.keys():
11     print("Corso ", corso, " voto ", esami[corso], "/30")
```

Il programma precedente produce il seguente output:

```
$ python3 iterazioni.py
cane
gatto
topo
Esami superati: IN110 AM110 AL110
Valutazioni per gli esami:
Corso  IN110  voto  27 /30
Corso  AM110  voto  30 /30
Corso  AL110  voto  24 /30
```

La variabile `animale` usata a riga 2 e a riga 3, ad ogni iterazione del ciclo, assume un diverso valore tra i vari elementi dell'insieme `a`. La variabile `corso` assume invece, ad ogni iterazione del ciclo controllato dall'istruzione `for` di riga 6, un diverso valore tra le chiavi del dizionario `esami`, restituite dal metodo `keys()`. Lo stesso esito si ottiene nel ciclo `for` alle righe 10 e 11, dove il valore della variabile `corso` viene usato come chiave del dizionario `esami` per estrarre il voto conseguito nell'esame corrispondente.

Da notare l'uso della funzione `print()`, che normalmente produce l'output su una riga e aggiunge automaticamente un carattere di fine-riga al termine dell'output, in modo tale che la successiva chiamata della funzione `print()` visualizzi l'output su una riga differente. Se si vuole evitare che a fine riga venga inserito il carattere di "a-capo", si può usare il parametro opzionale `end` per modificare il carattere aggiunto alla fine della riga. In particolare nell'istruzione a riga 5 e a riga 7 il parametro `end` viene impostato con il carattere spazio, in modo tale che dopo aver visualizzato ogni valore, il successivo venga visualizzato sulla stessa riga separato con uno spazio dal valore precedente.

6 Funzioni

In Python è possibile suddividere un programma in più sottoprogrammi utilizzando la definizione di **funzioni**. Una funzione è caratterizzata da un nome, un argomento costituito da zero o più parametri, un corpo della funzione, costituito dalle istruzioni che implementano l'algoritmo che definisce il comportamento della funzione e un valore restituito dalla funzione stessa.

Definizione di funzioni, l'istruzione "**def**"

Per definire una funzione si deve utilizzare la parola chiave «`def`» seguita dal nome della funzione, l'eventuale elenco di parametri delimitati da parentesi e separati l'uno dall'altro mediante delle virgole; la riga termina con il carattere duepunti. Quindi, nelle righe successive, indentate rispetto alla parola chiave `def`, vengono riportate le istruzioni che costituiscono il corpo della funzione. Con la funzione «`return`» si termina la funzione ed eventualmente si restituisce un valore all'istruzione chiamante. Ad esempio:

```
1 def mcm(x, y):
2     mx = x
3     my = y
4     while mx != my:
5         if mx<my:
6             mx = mx+x
```

```
7         else:
8             my = my+y
9     return(mx)
10
11 a = int(input("Inserisci un valore per a: "))
12 b = int(input("Inserisci un valore per b: "))
13 m = mcm(a,b)
14 print("mcm(", a, ",", b,") = ", m)
```

Il programma, salvato nel file "`mcm.py`" può essere eseguito dando luogo ad un risultato simile al seguente:

```
>>> python3 mcm.py
Inserisci un valore per a: 9
Inserisci un valore per b: 6
mcm( 9 , 6 ) = 18
```

La funzione `mcm` definita da riga 1 a riga 9, calcola il minimo comune multiplo tra i due valori (interi positivi) passati come argomento. Le variabili `x` e `y` sono **parametri** della funzione; assumono i valori indicati nella chiamata della funzione (nell'esempio la chiamata è a riga 13).

Parametri di una funzione

I parametri possono essere passati ad una funzione seguendo la *notazione posizionale* (il primo valore passato come argomento viene assegnato al primo parametro, il secondo al secondo parametro e così via), come in quasi tutti i linguaggi di programmazione, oppure è possibile specificare il nome del parametro a cui si intende assegnare un determinato valore, in modo da rendere la chiamata della funzione indipendenti dalla posizione dei parametri nella dichiarazione della funzione stessa (ma legandola ai nomi dei parametri utilizzati nella definizione della funzione!). Il seguente esempio mette in luce questo aspetto del linguaggio:

```
1 def scrivi(a, b):
2     print("primo parametro a = ", a)
3     print("secondo parametro b = ", b)
4     return
5
6 scrivi('Topolino', 'Pippo')
7 scrivi(b='Pippo', a='Topolino')
```

Eseguendo il programma (salvato nel file "`scrivi.py`") si ottiene il seguente risultato che mette in luce il fatto che entrambe le chiamate della funzione passano gli stessi valori ai due parametri, sebbene l'ordine dei valori nella seconda chiamata a riga 7 sia invertito rispetto all'ordine dei parametri nella chiamata a riga 6:

```
primo parametro a = Topolino
secondo parametro b = Pippo
primo parametro a = Topolino
secondo parametro b = Pippo
```

L'istruzione "**return**"

La funzione `mcm` presentata a pagina 11, termina eseguendo a riga 9 l'istruzione `return`, che restituisce il valore della variabile `mx` all'istruzione che ha richiamato

la funzione: a riga 13 il valore restituito dalla funzione `mcm` viene assegnato alla variabile `m`. È anche possibile definire funzioni che non restituiscono alcun valore; ad esempio:

```
1 def saluta(nome):
2     print("Ciao", nome)
3     return
4
5 x = input("Come ti chiami?")
6 saluta(x)
```

Eseguendo il programma salvato nel file "`saluti.py`", si ottiene il seguente output:

```
$ python3 saluti.py
Come ti chiami? Marco
Ciao Marco
```

Variabili locali e variabili globali

Le variabili `mx` e `my` del programma per il calcolo del minimo comune multiplo di pagina 11, sono **variabili locali**, definite all'interno della funzione `mcm`: questo significa che al termine dell'esecuzione della funzione le due variabili vengono eliminate dalla memoria della macchina. Le variabili definite nelle istruzioni del programma esterne ad una funzione, sono chiamate invece **variabili globali**.

Le variabili globali possono essere utilizzate anche all'interno di una funzione, anche se questa pratica andrebbe assolutamente evitata: è bene scrivere delle funzioni che siano completamente autonome ed indipendenti dal codice esterno alla funzione stessa; le funzioni devono essere completamente autosufficienti e autoconsistenti e la loro esecuzione non deve dipendere dal valore di altre variabili definite all'esterno della funzione stessa. L'unica modalità di "comunicazione" e di scambio di dati tra la funzione e il resto del programma è costituita dai parametri passati come argomento (per portare all'interno della funzione informazioni definite all'esterno) e dal valore restituito dalla funzione mediante l'istruzione `return` (per portare all'esterno della funzione un dato calcolato al suo interno, prima che le variabili definite localmente alla funzione vengano eliminate dalla memoria della macchina).

Le variabili locali definite all'interno di una funzione potrebbero essere denominate con nomi uguali a quelli di altre variabili globali, definite all'esterno della funzione, o di altre variabili locali definite in altre funzioni. Che variabili locali di funzioni diverse abbiano lo stesso nome non deve farci pensare che tali variabili assumeranno gli stessi valori: sono variabili di memoria distinte che avranno un ciclo di vita e l'assegnazione di valori del tutto indipendenti le une dalle altre. Nel caso in cui invece una variabile locale abbia lo stesso nome di una variabile globale, all'interno della funzione sarà considerata solo la variabile locale.

Flusso di esecuzione di un programma Python

Un programma Python viene composto quindi da un insieme di funzioni, definite all'inizio del programma, e da istruzioni esterne a qualsiasi funzione, riportate dopo la definizione delle funzioni. L'interprete Python esegue il programma a partire dalle istruzioni esterne alle funzioni. Le funzioni vengono eseguite solo se richiamate dalle istruzioni del programma esterne alle funzioni stesse; le funzioni possono essere richiamate anche da altre funzioni; si veda a tal proposito il seguente programmino utile a chiarire quanto detto.

```
1  def pluto():
2      pippo()
3      print("Pluto")
4      return
5
6  def pippo():
7      print("Pippo!")
8      return
9
10 print("Qui inizia l'esecuzione del programma.")
11 pluto()
12 print("Fine del programma.")
```

L'output del programma è il seguente: viene per prima eseguita l'istruzione a riga 10, quindi, con l'istruzione a riga 11 viene richiamata la funzione «`pluto()`»; quest'ultima esegue, a riga 2, una chiamata della funzione «`pippo()`». Terminata l'esecuzione di quest'ultima funzione, l'esecuzione ritorna alla funzione «`pluto()`» che esegue l'istruzione a riga 3 e quindi termina, facendo proseguire il programma con l'ultima istruzione a riga 12.

```
$ python3 funzioni.py
Qui inizia l'esecuzione del programma.
Pippo!
Pluto
Fine del programma.
```

È utile osservare che non è rilevante l'ordine con cui vengono definite le funzioni, anche nel caso in cui una funzione esegua la chiamata ad una funzione definita successivamente. È importante, invece, definire le funzioni prima delle istruzioni del programma esterne alle funzioni stesse.

Funzioni ricorsive

Le **funzioni ricorsive** sono funzioni che richiamano se stesse. Python ammette la possibilità di definire funzioni di questo tipo, anche se c'è un limite (dovuto alle risorse fisiche di memoria del sistema su cui viene eseguito il programma) al numero di chiamate ricorsive successive che è possibile effettuare.

Nell'esempio seguente riportiamo un programma in cui è definita la funzione ricorsiva «`fattoriale()`», che consente di calcolare il *fattoriale* di un numero naturale: dato $n > 0$, viene calcolato il valore $n!$ dato dal prodotto $n \cdot (n-1) \cdot (n-2) \cdots \cdot 3 \cdot 2 \cdot 1$. La funzione utilizza una definizione ricorsiva, data dalla seguente espressione:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{se } n \geq 1 \end{cases}$$

A riga 5 del seguente programma viene calcolato il valore della variabile `f` eseguendo una chiamata ricorsiva della funzione «`fattoriale()`».

```
1  def fattoriale(n):
2      if n==0: f = 1
3      else: f = n*fattoriale(n-1)
4      return(f)
5
```

```
6 n = int(input("inserisci un numero intero positivo: "))
7 print(n, "! = ", fattoriale(n), sep="")
```

È forse utile notare che nel programma precedente le istruzioni da eseguire sono state riportate sulla stessa riga dell'istruzione `if` e dell'istruzione `else`. Eseguendo il programma si ottiene un output analogo al seguente:

```
$ python3 fattoriale.py
inserisci un numero intero positivo: 100
100! =  933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182862536979208272237582511852109168640000000000000000000000000
```

7 File

Con Python è facile utilizzare i **file** per salvare dati prodotti dal programma o per caricare in input dati presenti su un file. Per poter utilizzare un file è necessario aprirlo utilizzando la funzione «`open()`» e specificando come argomenti il nome del file, eventualmente comprensivo del *path* e la modalità con cui si intende accedere al file stesso: in scrittura o in lettura. Non è possibile aprire un file ed eseguire sia operazioni di lettura che operazioni di scrittura sul file: quando lo si apre si deve stabilire come si intenda utilizzarlo. Dopo aver eseguito le operazioni di lettura o quelle di scrittura, il file deve essere chiuso utilizzando la funzione «`close()`». A quel punto, se necessario, sarà possibile aprirlo nuovamente cambiando la modalità con cui si intende operare su di esso.

La funzione "**open**" per aprire i file

Per creare un nuovo file lo si deve aprire in **modalità di scrittura** utilizzando l'istruzione «`open("nomedelfile", "w")`»: la lettera «`w`» rappresenta la parola inglese *write* per indicare che il file viene aperto in scrittura. Se il file che viene aperto in modalità di scrittura già esiste, il suo contenuto sarà eliminato sovrascrivendolo con le successive operazioni di scrittura.

Creazione di un nuovo file in modalità di scrittura

Se il file già esiste lo si può aprire in scrittura in **modalità *append***, utilizzando il carattere "`a`" per indicare questa modalità: in questo modo le operazioni di scrittura aggiungeranno nuovi dati al file, senza sovrascrivere eventuali informazioni già presenti nello stesso file. Anche aprendo il file in modalità *append*, se il file non esiste viene creato.

Aggiunta di dati ad un file in modalità append

Per aprire un file già esistente in **modalità di lettura** si deve specificare la lettera «`r`», che rappresenta la parola inglese *read*: «`open("nomedelfile", "r")`».

Apertura del file in modalità di lettura

La funzione «`open()`», se riesce ad aprire il file nella modalità specificata, restituisce un "puntatore" al file: è un riferimento al file che bisogna memorizzare in una variabile per poter fare riferimento allo stesso file nelle successive istruzioni del programma. Ad esempio:

```
1 f = open("voti", "w")
2 f = open("/home/marco/simulaz.dat", "a")
3 f = open("./dati/risultati.txt", "r")
```

La prima istruzione apre in modalità di scrittura il file denominato "`voti`", che viene creato nella stessa directory da cui viene eseguito il programma Python. Con

la seconda istruzione dell'esempio precedente, il file "`simulaz.dat`" viene aperto in modalità *append* nella directory "`/home/marco`". Nel terzo esempio viene aperto in modalità di lettura il file "`risultati.txt`"; il file deve esistere già e si deve trovare nella sottodirectory "`/dati`" della directory corrente, altrimenti si verificherà una condizione di errore che blocca l'esecuzione del programma. In tutti e tre i casi il riferimento al file aperto viene memorizzato nella variabile `f`. Naturalmente in uno stesso programma possono essere aperti più file contemporaneamente, ma il riferimento di ciascun file dovrà essere memorizzato in una variabile diversa.

L'apertura di un file è sempre un'operazione critica, che potrebbe provocare un errore e la conseguente interruzione dell'esecuzione del programma. Il file potrebbe infatti non essere presente oppure l'utente che esegue il programma potrebbe non avere i permessi sufficienti ad aprire il file nella modalità specificata con la funzione «`open()`». Per gestire questo genere di errori ed evitare che l'esecuzione del programma venga interrotta bruscamente, si possono intercettare gli errori generati dall'esecuzione della funzione «`open()`». Per far questo è necessario utilizzare l'istruzione «`try... except...`» Le istruzioni "fragili", che potrebbero provocare degli errori al momento dell'esecuzione del programma («a *run-time*») devono essere incluse nel blocco dell'istruzione `try`; in caso di errori il programma non si blocca, ma esegue le istruzioni presenti nel blocco dell'istruzione `except`. Il seguente esempio aiuterà a capire il funzionamento di questa istruzione:

L'istruzione **try–except** per intercettare e gestire gli errori

```
1 nomefile = input("Inserisci il nome del file: ")
2 try:
3     f = open (nomefile, "r")
4 except:
5     print("Il file", nomefile, "non esiste o non puo' essere aperto")
```

Il metodo **write** per scrivere in un file

Con la funzione (metodo) «`write()`», applicato al file aperto con la funzione «`open()`» in modalità scrittura, è possibile scrivere dei dati nel file. Siccome nel file possono essere scritte solo stringhe di testo, nel caso in cui volessimo scrivere dei numeri, dovremo prima trasformarli in stringhe con la funzione «`str()`».

Il metodo **read** per leggere dati da un file

Analogamente, con la funzione «`read()`» è possibile leggere stringhe di testo da un file aperto in modalità lettura. In particolare la funzione accetta come argomento il numero di caratteri da leggere dal file (es.: «`read(5)`'»; richiamando la funzione «`read()`» senza alcun argomento, viene letto l'intero contenuto del file. Per leggere una sola riga per volta si può usare la funzione «`readline()`». Anche in questo caso, se le stringhe lette in input dal file rappresentano dei numeri, dovremo trasformarli in dati numerici con le funzioni «`int`», «`long`» o «`float`». Sia il metodo «`read()`» che il metodo «`readline()`» restituiscono la stringa vuota ("") quando è stata raggiunta la fine del file.

Il metodo **readline** per leggere un'intera riga da un file

Utilizzare la funzione «`write()`» su un file aperto in lettura o la funzione «`read()`» su un file aperto in scrittura, provoca un errore.

Chiusura di un file, la funzione **close**

Quando sono state completate le operazioni di lettura o di scrittura su un file aperto, è necessario chiuderlo utilizzando la funzione «`close()`». Il seguente programma di esempio, acquisisce in input dall'utente i voti conseguiti nei diversi esami e li salva sul file "`voti.txt`"; per salvare sul file sia il nome del corso che il voto dell'esame con una sola istruzione «`f.write()`», viene utilizzata la funzione «`format()`», presentata a pagina 5, per formattare la stringa passata come argomento al metodo «`write()`».

```
1  voto = dict()
2  n = int(input("Numero di esami superati: ")
3  for i in range(0,n):
4      corso = input("Sigla del corso: ")
5      voto[corso] = int(input("Voto dell'esame: "))
6  nomefile = input("Nome del file: ")
7  f = open(nomefile, "w")
8  for corso in keys(voto):
9      f.write("{0}\n{1}\n".format(corso, voto[corso]))
10 f.close()
```

È possibile scrivere su un file anche attraverso la funzione «`print()`» specificando nell'opzione «`file`» il riferimento al file in cui si intende scrivere. Ad esempio l'istruzione a riga 9 del programma precedente può essere sostituita con la seguente:

Output su file con la funzione **print**

```
9  print(corso, voto[corso], sep="\n", file=f)
```

Il seguente programma Python esegue l'operazione opposta rispetto al programma precedente: apre in lettura il file in cui sono stati salvati i voti degli esami, legge il contenuto del file una riga per volta con il metodo «`readline()`» e lo visualizza sullo schermo del terminale dell'utente.

```
1  voto = dict()
2  nomefile = input("Nome del file: ")
3  f = open(nomefile, "r");
4  x = f.readline()
5  while x != "":
6      y = int(f.readline())
7      voto[x] = y
8      x = f.readline()
9  f.close()
10 s = 0
11 n = 0
12 for corso in voto.keys():
13     print("Corso:", corso, "Voto:", voto[corso])
14     n = n+1
15     s = s+voto[corso]
16 print("Hai sostenuto", n, "esami con la media del", s/n)
```

I dati letti in input dal file sono alternativamente una stringa (il codice del corso) e un numero intero (il voto dell'esame). Vengono sempre letti a coppie, memorizzando il codice del corso nella variabile `x` e il voto nella variabile `y`. Gli stessi dati vengono anche memorizzati nel dizionario `voto`, con l'istruzione a riga 7. Con il ciclo alle righe 12–15, vengono visualizzati tutti i valori letti in input e memorizzati nel dizionario; vengono anche contati gli esami, incrementando di 1 il valore della variabile `n` ad ogni iterazione del ciclo e viene calcolata la somma `s` dei voti degli esami; in questo modo, a riga 16, alla fine del ciclo, è possibile calcolare e visualizzare il numero di esami sostenuti e la media aritmetica dei voti.

8 Grafica con Python

Il linguaggio Python mette a disposizione numerosi *moduli* (librerie di funzioni e classi di oggetti che estendono ed arricchiscono di nuove caratteristiche il linguaggio Python), alcuni dei quali consentono di aprire finestre e visualizzare dei grafici al loro interno (linee, punti, poligoni, grafici di funzione, ecc.).

L'istruzione **import** per il caricamento di un modulo

La parola chiave **as** per definire un alias di un modulo

Per includere gli oggetti, le strutture dati e le funzioni presenti in uno specifico modulo occorre "importarlo" utilizzando l'istruzione «`import`». Ad esempio con la seguente istruzione si può caricare il modulo «`pylab`»; con la parola chiave «`as`» è possibile definire un *alias*, una parola più breve o più significativa, per fare riferimento al modulo che abbiamo importato; nell'esempio potremo fare riferimento ai metodi e alle funzioni del modulo `matplot.pylab` semplicemente con l'alias `pl`:

```
1 import matplotlib.pylab as pl
```

Con l'istruzione «`from`» è possibile caricare solo una parte di un modulo e non tutte le definizioni di oggetti e metodi contenuti al suo interno; ad esempio se volessimo importare dalla libreria `numpy` solo il metodo «`sin()`» per il calcolo della funzione trigonometrica "seno", potremmo utilizzare la seguente istruzione:

```
1 from numpy import sin as seno
2 y = seno(3.14)
```

Il modulo **pylab**

Il modulo «`pylab`» definito nella libreria MatPlotLib offre numerose funzioni e un vero e proprio ambiente grafico per la visualizzazione di grafici e istogrammi bidimensionali su un piano cartesiano.

Partiamo da un esempio molto semplice, definendo le coordinate di alcuni punti che vogliamo tracciare sul piano e unire con dei segmenti di retta, per costruire una spezzata:

```
1 import matplotlib.pylab as pl
2 x = [1, 2, 3, 4, 5, 6]
3 y = [2, 5, 9, 18, 21, 25]
4 pl.plot(x, y)
5 pl.show()
```

L'istruzione al passo 2 definisce una lista di sei numeri interi denominata `x` e l'istruzione al passo 3 definisce la lista `y` con lo stesso numero di elementi. Considerando i valori corrispondenti della lista `x` e della lista `y` come coordinate di punti sul piano cartesiano, l'istruzione «`pl.plot(x,y)`» a riga 4 costruisce un grafico unendo i punti della sequenza con una linea "spezzata": il punto di coordinate $(1,2)$ viene unito da un segmento con il punto di coordinate $(2,5)$, che poi viene unito con un altro segmento con il punto di coordinate $(3,9)$ e così via... Con l'ultima istruzione «`pl.show()`» a riga 5 viene visualizzata la finestra grafica con il disegno all'interno.

L'output prodotto dal programma è rappresentato in Figura 1. Come si può notare la finestra grafica entro cui viene rappresentato il disegno, presenta anche dei bottoni con cui si può eseguire uno zoom sul disegno visualizzato (icona con la lente d'ingrandimento) e quindi, con gli altri bottoni, ci si può muovere nell'area del disegno, tornare alla versione originale del disegno (icona con la casetta), salvare il disegno su un file.

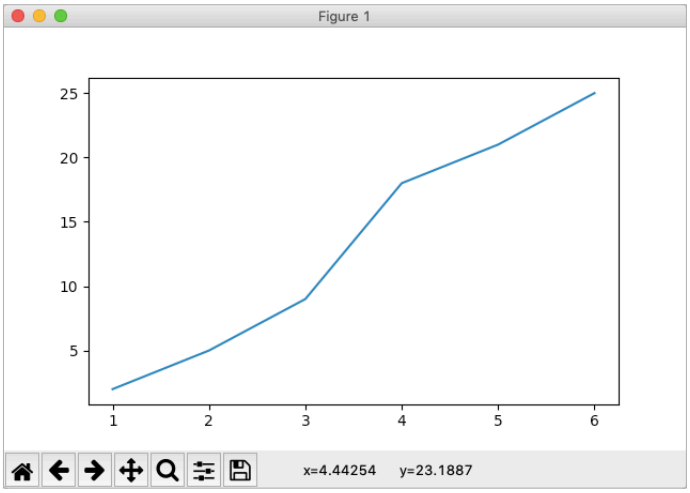

Figura 1: Un output grafico prodotto utilizzando il modulo PyLab

Importando il modulo `pylab` a riga 1 viene anche definito l'alias `pl` che rappresenta la finestra grafica. Su tale oggetto è possibile operare con i metodi «`plot()`», «`axis()`», «`bar()`», «`show()`» ed altri ancora.[1]

In particolare il metodo «`plot()`» ammette un numero arbitrario di parametri. L'uso più comune è quello visto nell'esempio precedente in cui i parametri sono due liste di numeri che, considerati a coppie, rappresentano le coordinate di un insieme di punti sul piano cartesiano.

Definizione dell'aspetto e del colore del grafico

Con l'eventuale terza opzione del metodo «`plot()`» si può definire l'aspetto del grafico visualizzato nella finestra: normalmente, senza specificare alcun parametro, i punti vengono uniti da segmenti di colore blu; in questo caso l'opzione implicita è la stringa "`b-`". Con l'opzione "`ro`" vengono tracciati dei cerchietti rossi in corrispondenza dei punti, senza unirli con dei segmenti.

È possibile invocare più volte il metodo «`plot()`» per tracciare più grafici nella stessa finestra. Ad esempio nel programma precedente si può aggiungere una chiamata al metodo «`plot()`» per evidenziare con dei cerchietti rossi i punti della "spezzata" che compongono il grafico:

```
5 pl.plot(x, y, "ro")
```

Il modulo **numpy**

È interessante utilizzare l'uso del modulo `numpy`, che offre numerose funzioni matematiche e strutture dati per la rappresentazione di matrici multidimensionali, in unione al modulo `pylab`. Nell'esempio seguente vengono visualizzati tre grafici di funzione al variare di x nell'intervallo $(-1, 15)$. In Figura 2 è rappresentato l'output del programma.

```
1 import numpy as np
2 import matplotlib.pylab as pl
3 x = np.arange(-1,15,0.1)
```

[1] A tal proposito si veda uno dei *tutorial* disponibili sul sito ufficiale della libreria MatPlotLib all'indirizzo `https://matplotlib.org`

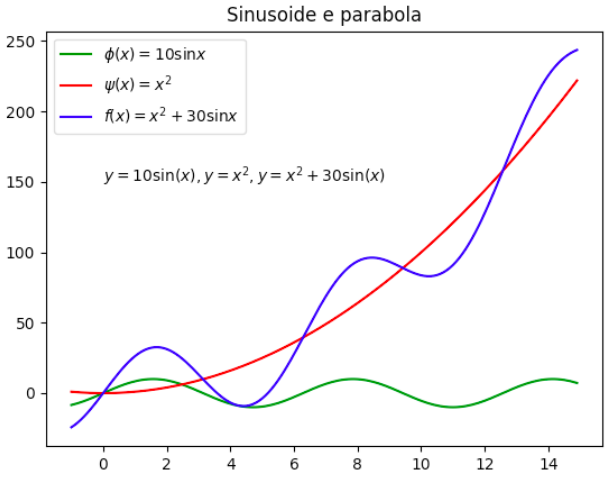

Figura 2: Tra grafici di funzione rappresentati nella stessa finestra

```
4  pl.plot(x, np.sin(x)*10, "g-", label="$\phi(x) = 10 \sin x$")
5  pl.plot(x, x**2, "r-", label="$\psi(x) = x^2$")
6  pl.plot(x, x**2+np.sin(x)*30, "b-", label="$f(x) = x^2 + 30 \sin x$")
7  pl.title("Sinusoide e parabola")
8  pl.text(0, 150, "$y = 10\sin(x), y = x^2, y = x^2+30\sin(x)$")
9  pl.legend()
10 pl.show()
```

Il metodo **arange** per la discretizzazione di un intervallo

In particolare con l'istruzione al passo 3, «`x = np.arange(-1, 15, 0.1)`» si assegna come valori della lista `x` i valori numerici nell'intervallo $(-1,15)$, a partire dal valore -1, a distanza 0.1 l'uno dall'altro: $-1.0, -0.9, -0.8, \ldots, 14.9, 15.0$.

Le successive chiamate del metodo «`plot()`» a riga 4, 5 e 6, visualizzano il grafico ottenuto collegando i punti di coordinate `x` e il valore in `x` della funzione riportata come secondo parametro (rispettivamente $10\sin x$, x^2, $x^2+10\sin x$). La funzione trigonometrica "seno" non è presente tra le funzioni standard del linguaggio Python, ma è definita nella libreria `numpy`; per cui per usarla utilizziamo l'espressione «`np.sin(x)`» (`np` è l'alias di `numpy` definito nell'istruzione `import` a riga 1).

Tra i parametri del metodo «`plot()`» riportiamo anche il parametro opzionale «`label="..."`» che consente di definire un'etichetta associata al grafico che verrà poi visualizzata nella legenda, eseguendo l'istruzione «`pl.legend()`». Il testo della legenda può essere scritto utilizzando la sintassi del linguaggio TEX nell'opzione `label` del metodo `plot`.

I metodi **title** e **text** per la visualizzazione di titoli ed etichette

A riga 7 viene utilizzato il metodo «`title()`» sull'oggetto `pl` per visualizzare un titolo sopra al grafico. Il metodo «`text()`» consente, invece, di visualizzare sul piano cartesiano una stringa di testo, a partire dal punto con le coordinate specificate come primo e secondo parametro. Da notare che, anche in questo caso, la stringa di testo può essere codificata in linguaggio TEX.

La libreria MatPlotLib è dunque uno strumento assai utile per la visualizzazione di grafici di funzione o di curve sul piano cartesiano. Può essere utilizzata, attra-

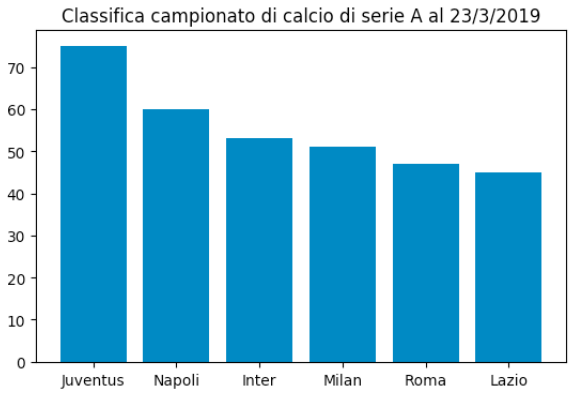

Figura 3: La classifica dei punti conseguiti dalle prime sei squadre nel campionato di calcio di Serie A

verso il modulo `pyplot`, anche per visualizzare dei diagrammi a barre. Ad esempio consideriamo il seguente programma:

Il modulo **pyplot** della libreria MatPlotLib

```
1 import matplotlib.pyplot as plt
2 squadre = ['Juventus', 'Napoli', 'Inter', 'Milan', 'Roma', 'Lazio']
3 punti = [75, 60, 53, 51, 47, 45]
4 plt.bar(squadre, punti)
5 plt.title("Classifica campionato di calcio di serie A al 23/3/2019")
6 plt.show()
```

A riga 2 viene definita la lista `squadre` di stringhe di testo, corrispondenti ai nomi di alcune squadre di calcio; a riga 3 viene definita la lista `punti`, composta da numeri che rappresentano i punti conseguiti dalle corrispondenti squadre di calcio. Con l'istruzione «`plt.bar(squadre, punti)`» viene visualizzato un diagramma a barre con il punteggio di ciascuna squadra (vedi Figura 3).

Il metodo **bar** per la visualizzazione di diagrammi a barre

Per produrre un output grafico è possibile utilizzare, in alternativa a quanto visto nelle pagine precedenti, anche gli oggetti e i metodi offerti dal modulo `graphics`, una libreria grafica molto semplice e potente, che consente di tracciare punti, linee, cerchi e poligoni all'interno di una finestra, definendo un arbitrario sistema di coordinate bidimensionali.

Il modulo **graphics**

La libreria offre una classe di oggetti denominata `GraphWin`, con cui possono essere definite le finestre grafiche su cui possono essere tracciati i disegni; offre inoltre una serie di classi di oggetti grafici che possono essere disegnati all'interno delle finestre grafiche (es.: punti, linee, cerchi, ellissi, rettangoli ed altri poligoni).

Per utilizzare la libreria `graphics` si deve quindi prima definire un oggetto di tipo `GraphWin`, definendone le proprietà (dimensione, titolo, colore di sfondo, sistema di coordinate), poi si possono definire altri oggetti grafici e disegnarli all'interno della finestra.

La finestra è in grado di reagire a degli eventi, come la pressione di tasti da parte dell'utente, o il click sul mouse, restituendo il carattere corrispondente al tasto premuto o le coordinate del puntatore del mouse, nel momento in cui l'utente ha eseguito un click.

Di seguito riportiamo un esempio elementare.

```
1  from graphics import *
2  win = GraphWin("Prova di grafica", 800, 600)
3  win.setBackground("white")
4  win.setCoords(-10, -7, 20, 15)
5  assex = Line(Point(-10, 0), Point(20, 0))
6  assex.setArrow("last")
7  assex.draw(win)
8  assey = Line(Point(0, -7), Point(0, 15))
9  assey.setArrow("last")
10 assey.draw(win)
11 for i in range(0,10):
12     c = Circle(Point(i, i), 2*i)
13     c.setOutline("red")
14     c.draw(win)
15 t = Text(Point(5, 5), "Cerchi sempre più grandi")
16 t.setOutline("blue")
17 t.setSize(20)
18 t.draw(win)
19 win.getMouse()
20 win.close()
```

Definizione della finestra grafica con un oggetto della classe **GraphWin**

Con l'istruzione a riga 1 viene importato il modulo `graphics`, con tutte le sue componenti. A riga 2 viene creato un oggetto denominato `win` della classe `GraphWin`, impostando come titolo la stringa "`Prova di grafica`"; la finestra è larga 800 pixel ed è alta 600 pixel. Con l'istruzione a riga 3 viene impostato come sfondo della finestra il colore bianco e con l'istruzione «`win.setCoords(-10, -7, 20, 15)`», a riga 4, viene definito il sistema di coordinate all'interno della finestra: il punto in basso a sinistra ha coordinate $(-10,-7)$, mentre il punto in alto a destra ha coordinate $(20,15)$.

Definizione di un oggetto della classe **Line** per la rappresentazione di linee (e frecce)

Per dare un'idea del sistema di coordinate definito nella finestra, tracciamo gli assi cartesiani: con l'istruzione a riga 5 viene definito l'oggetto `assex` della classe `Line`, che rappresenta un segmento di retta orizzontale, dal punto di coordinate $(-10,0)$ al punto $(20,0)$. All'estremità finale del segmento, con l'istruzione di riga 6 «`assex.setArrow("last")`» di riga 6, viene visualizzata la punta di una freccia. Quindi l'oggetto `assex` viene visualizzato nella finestra `win` con l'istruzione «`assex.draw(win)`» di riga 7. Le stesse operazioni vengono compiute per visualizzare l'asse y, rappresentato dall'oggetto `assey`, con le istruzioni da riga 8 a riga 10.

Definizione di un oggetto della classe **Circle** per la visualizzazione di circonferenze

Con le istruzioni da riga 11 a riga 14, viene iterato un ciclo, al variare della variabile `i` da 0 a 9, per tracciare dieci cerchi rossi di raggio sempre più grande. Con l'istruzione di riga 12 si definisce l'oggetto `c` della classe `Circle`, definendo il centro della circonferenza («`Point(i,i)`») e il raggio («`2*i`»). Il colore della circonferenza viene definito utilizzando il metodo `setOutline()` a riga 13. Infine, anche in questo caso, con il metodo `draw()` a riga 14, il cerchio viene disegnato nella finestra `win`.

Definizione di un oggetto della classe **Text** per visualizzare testi nella finestra grafica

Con le istruzioni da riga 15 a riga 18 viene visualizzata una stringa di testo nella finestra grafica. La stringa di testo è un oggetto della classe `Text` definito indicando le coordinate del punto centrale della scritta che si vuole visualizzare nella finestra e il testo che la compone: con l'istruzione a riga 15 viene definito l'oggetto

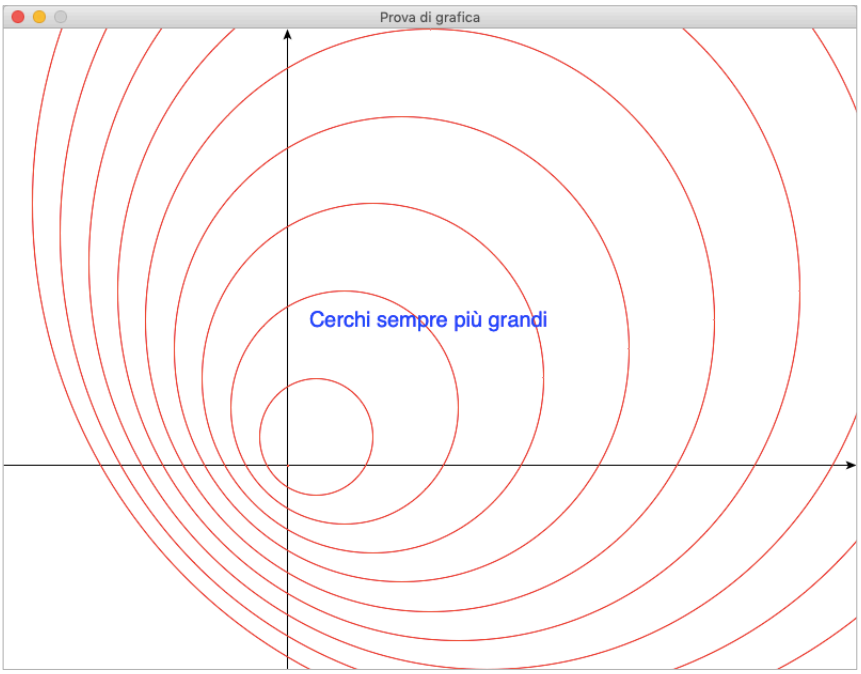

Figura 4: Grafica prodotta utilizzando gli oggetti e i metodi del modulo `graphics`

`t` della classe `Text`. Per visualizzare il testo viene utilizzato un carattere di colore blu (istruzione «`t.setOutline("blue")`» a riga 16) di dimensione 20 (istruzione «`t.setSize(20)`» a riga 17). Infine, con l'istruzione a riga 18, l'oggetto di testo viene visualizzato nella finestra utilizzando il metodo `draw()` come nel caso degli altri oggetti grafici.

Gestione degli eventi e interazione dell'utente con la finestra grafica

Come dicevamo, l'oggetto finestra grafica può rilevare il verificarsi di un evento, determinato da una interazione dell'utente con la finestra utilizzando la tastiera o il mouse. Con l'istruzione «`win.getMouse()`» di riga 19 il programma rimane in attesa che l'utente esegua un click con il mouse.

Con l'istruzione di riga 20 «`win.close()`» viene chiusa la finestra grafica e il programma termina.

9 Matrici

Le matrici a due o più dimensioni possono essere definite come array di array (liste di liste). Gli elementi sono quindi identificati mediante degli indici numerici interi che iniziano tutti dal valore 0. Ad esempio l'elemento nella terza colonna della settima riga della matrice `A` è identificato dall'espressione «`A[6][2]`»: l'indice 6 identifica la settima riga (la prima è identificata dall'indice 0) e l'indice 2 identifica la terza colonna.

Ancora una volta può essere di aiuto utilizzare il package `numpy`, che offre diverse funzioni per definire ed inizializzare delle matrici a più dimensioni:

Funzioni della libreria **numpy** per l'inizializzazione di matrici

- «`numpy.empty((`n_1`, `n_2`, ..., `n_k`))`»: definisce una matrice a k dimensioni senza inizializzare gli elementi con alcun valore; gli elementi della matrice conterranno quindi dei valori imprevedibili, apparentemente casuali;
- «`numpy.full((`n_1`, `n_2`, ..., `n_k`), `x`)`»: definisce una matrice a k dimensioni e assegnata a tutti gli elementi lo stesso valore x specificato come argomento;
- «`numpy.zeros((`n_1`, `n_2`, ..., `n_k`))`»: definisce una matrice a k dimensioni e assegnata a tutti gli elementi il valore zero;
- «`numpy.ones((`n_1`, `n_2`, ..., `n_k`))`»: definisce una matrice a k dimensioni e assegnata a tutti gli elementi il valore 1;

Nel seguente esempio vengono definite due matrici di tre righe e quattro colonne ciascuna:

```
>>> import numpy as np
>>> np.full((3,4), 17)
array([[17, 17, 17, 17],
       [17, 17, 17, 17],
       [17, 17, 17, 17]])
>>> np.zeros((3,4), dtype=int)
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```

Il primo parametro passato come argomento alle funzioni è una lista che la cardinalità (il numero di elementi) della matrice per ogni dimensione: nell'esempio precedente sono state definite due matrici a due dimensioni, con tre righe e quattro colonne ciascuna.

Nella funzione «`full()`» viene passato come argomento anche il valore (17 nell'esempio) che si desidera assegnare a tutti gli elementi della matrice in fase di inizializzazione.

Indicazione del tipo di dato nell'inizializzazione della matrice

È possibile passare come argomento della funzione anche il tipo di dato con cui deve essere definita la matrice: in questo caso l'argomento è composto dall'espressione «`dtype=`*tipo*».

Nell'esempio riportato di seguito viene calcolato e stampato il cosiddetto triangolo di Tartaglia, che, come è noto, riporta i coefficienti dello sviluppo del binomio $(a+b)^n$.

```
1  import numpy as np
2
3  n = int(input("Ordine del triangolo: "))
4  a = np.zeros((n,n), dtype=int)
5
6  a[0][0] = 1
7  for i in range(1, n):
8      a[i][0] = 1
9      for j in range(1, i):
10         a[i][j] = a[i-1][j-1]+a[i-1][j]
```

```
11      a[i][i] = 1
12
13  for i in range(n):
14      for j in range(i+1):
15          print(a[i][j], end=" ")
16      print("")
```

Dopo aver acquisito in input il valore di n (riga 3), viene definita una matrice quadrata di ordine n di interi (riga 4) utilizzando la funzione «`zeros()`». Eseguendo il programma, si programma si ottiene un output analogo al seguente:

```
$ python3 tartaglia.py
Ordine del triangolo: 5
1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
```

10 Numeri casuali

In Python esiste più di un modulo/libreria che fornisce funzioni di vario tipo per la generazione di numeri pseudo-casuali. In particolare il modulo **secrets**, descritto esaustivamente nella documentazione on-line disponibile all'indirizzo `https://docs.python.org/3.7/library/secrets.html`, fornisce, tra le altre, le seguenti funzioni:

Alcune funzioni del modulo **secrets** per la generazione di numeri pseudo-casuali

- «`secrets.randbelow(`x`)`»: restituisce un numero intero casuale positivo, minore del valore passato come argomento;
- «`secrets.choice(`A`)`»: restituisce un elemento casuale della lista di valori (numeri, stringhe, ecc.) passata come argomento;
- «`secrets.randbits(`n`)`»: restituisce un numero intero composto da un numero di bit casuali specificato come argomento (es.: se l'argomento è n, verrà prodotto un numero casuale compreso tra 0 e $2^n - 1$).

Di seguito riportiamo degli esempi sull'uso di alcune delle funzioni definite nel modulo `secrets`:

```
>>> import secrets as sec
>>> sec.randbelow(10)
3
>>> sec.choice(('pari', 'dispari'))
'dispari'
>>> sec.randbits(3)
5
```

Il modulo **random** fornisce altre funzioni per la generazione di numeri pseudo-casuali, in parte analoghe a quelle del modulo `secrets`. Una descrizione completa di questo modulo la si può trovare nella documentazione on-line disponibile al seguente indirizzo: `https://docs.python.org/3.7/library/random.html`.

Alcune funzioni del modulo **random** per la generazione di numeri e sequenze pseudo-casuali

Tra le numerose funzioni messe a disposizione da questo modulo, ci limitiamo, in queste pagine, a citare le seguenti:

- «`random.random()`»: restituisce un numero floating point compreso nell'intervallo $[0,1)$ (quindi compreso il valore 0 ed escluso il valore 1);
- «`random.seed(`n`)`»: imposta un seme (numero intero) specificato come argomento per la generazione dei successivi numeri casuali (impostando lo stesso seme, viene prodotta la stessa successione di numeri casuali);
- «`random.getrandbits(`n`)`»: restituisce un numero intero composto da un numero di bit casuali specificato come argomento (es.: se l'argomento è n, verrà prodotto un numero casuale compreso tra 0 e $2^n - 1$);
- «`random.randint(`x`,`y`)`»: restituisce un numero intero compreso nell'intervallo specificato come argomento, estremi inclusi;
- «`random.choice(`A`)`»: restituisce un elemento casuale della lista di valori passata come argomento;
- «`random.shuffle(`A`)`»: modifica l'ordine degli elementi dell'array passato come argomento, producendo una permutazione casuale degli elementi.

Il seguente esempio genera una matrice di numeri interi di n righe ed n colonne; dopo aver assegnato alla prima riga una sequenza di numeri casuali compresi tra 1 e 9, assegna alle righe successive una permutazione casuale di tali elementi.

```
1  import numpy as np
2  import random as rnd
3
4  n = int(input("Inserisci n: "))
5  a = np.empty((n,n), dtype=int)
6
7  for j in range(n):
8      a[0][j] = rnd.randint(1,9)
9  for i in range(1,n):
10     a[i] = a[0]
11     rnd.shuffle(a[i])
12
13 for i in range(n):
14     for j in range(n):
15         print(a[i][j], end=" ")
16     print("")
```

Con le istruzioni alle righe 1 e 2 vengono caricate le librerie/moduli `numpy` e `random` che ci permettono, rispettivamente, di utilizzare le funzioni per l'inizializzazione delle matrici (`numpy`) e per la generazione di numeri casuali e la generazione di permutazioni casuali (`random`). Con l'istruzione a riga 5 viene inizializzata una matrice quadrata di numeri interi con n righe ed n colonne.

Il ciclo alle righe 7 e 8 assegna, utilizzando la funzione `randint()` della libreria `random`, un numero intero casuale compreso tra 1 e 9 a ciascun elemento della prima riga della matrice. Con le istruzioni del ciclo alle righe 9–11, vengono assegnati dei valori alle righe successive alla prima (indice di riga: $1, 2, \ldots, n-1$) eseguendo

prima una copia della prima riga (riga della matrice di indice 0) sulla riga i-esima (istruzione a riga 10) con una singola operazione di assegnazione (istruzione a riga 9) e poi generando una permutazione casuale degli elementi della riga, mediante la funzione `shuffle()` della libreria `random` di riga 11.

Eseguendo il programma si ottiene un output simile al seguente:

```
$ python3 matriceCasuale.py
Inserisci n: 4
3 4 1 6
3 6 1 4
4 1 6 3
4 6 1 3
```

11 Espressioni regolari

Un'*espressione regolare* è la descrizione di un *pattern* attraverso un meta-linguaggio. Con un'espressione regolare si può descrivere un insieme molto grande di stringhe di testo, senza doverle elencare una per una. Ad esempio si può costruire un'espressione regolare per descrivere il pattern con cui sono costruiti tutti gli indirizzi e-mail utilizzati su Internet, oppure un'espressione regolare per descrivere le URL, o un codice fiscale italiano e molto altro ancora.

Pattern matching e pattern substitution

Le espressioni regolari sono utili per verificare se una determinata stringa (o una sua parte, una sotto-stringa) rispetta o meno un determinato schema: in questo caso si parla di operazione di *pattern matching*. Le espressioni regolari possono essere utilizzate anche per modificare parti di una stringa che corrispondono ad un determinato pattern, con altri caratteri. Questo tipo di operazione prende il nome di *pattern substitution*.

Di fatto, attraverso un insieme di meta-caratteri, con un'espressione regolare è possibile definire un automa a stati finiti per riconoscere l'appartenenza di una stringa al linguaggio (insieme di stringhe) descritto dall'espressione regolare stessa.

Meta-caratteri per costruire un'espressione regolare

Ciascun carattere alfanumerico in un'espressione regolare rappresenta se stesso, salvo i seguenti meta-caratteri che hanno un significato particolare:

- «`.`»: il punto indica un carattere qualsiasi;
- «`\.`»: questo meta carattere indica il punto;
- «`\\`»: back-slash («`\`»);
- «`\w`»: un qualsiasi carattere alfabetico o numerico e il simbolo di sottolineatura «_»;
- «`\W`»: un qualsiasi carattere tranne i caratteri alfabetici, numerici o il simbolo di sottolineatura;
- «`\d`»: un carattere numerico decimale (`0`, `1`, `2`, ..., `9`);
- «`\D`»: un carattere qualsiasi tranne i numeri;
- «`\s`»: un qualsiasi carattere di spaziatura o di fine riga;
- «`\S`»: un carattere qualsiasi tranne i caratteri di spaziatura o di fine riga;

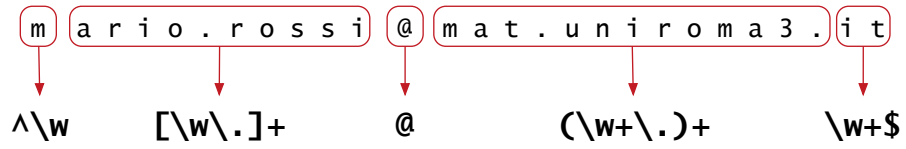

Figura 5: Espressione regolare per descrivere un indirizzo e-mail Internet

- «[...]»: un carattere qualsiasi tra quelli elencati tra parentesi quadrate; possono essere indicate anche sequenze di caratteri come «a-z» per indicare tutte le lettere alfabetiche minuscole, o «A-Z» per indicare tutte le maiuscole;
- «[^...]»: un carattere qualsiasi tranne quelli elencati tra parentesi quadrate;

Ad esempio con l'espressione regolare «^\w[\w\.]+@(\w+\.)+\w+$» descrive tutte le possibili stringhe corrispondenti ad un indirizzo e-mail, come esemplificato in Figura 5.

La stessa espressione regolare può essere descritta con un automa a stati finiti che esegue l'analisi carattere per carattere della stringa per verificare se corrisponde o meno al pattern descritto della stessa espressione regolare (vedi Figura 6).

Altri meta-caratteri e esprimono una "condizione posizionale" per una determinata stringa (es.: tutte le stringe che iniziano con la sequenza..., oppure tutte le stringhe che terminano con la sequenza ...) o servono come "quantificatori" dei caratteri che li precedono:

Meta-caratteri quantificatori e per condizioni posizionali

- «*»: quantificatore: zero o più caratteri dello stesso tipo descritto dall'espressione regolare precedente;
- «?»: quantificatore: zero o un solo carattere dello stesso tipo descritto dall'espressione regolare precedente;
- «+»: quantificatore: uno o più caratteri dello stesso tipo descritto dall'espressione regolare precedente;
- «^»: la stringa deve iniziare con una sequenza di caratteri descritta dall'espressione regolare seguente;

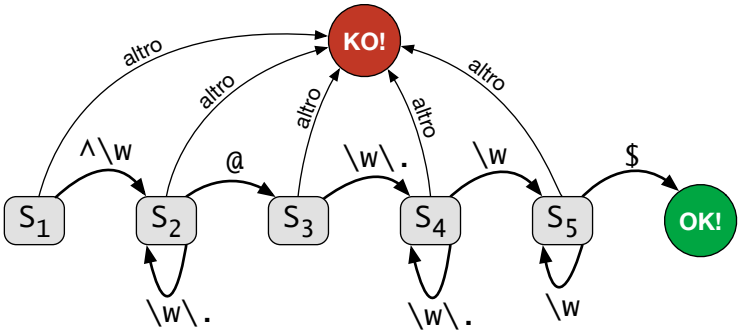

Figura 6: Il grafo di transizione di stato dell'automa a stati finiti che rappresenta l'espressione regolare di Figura 5

- «$»: la stringa deve terminare con una sequenza di caratteri descritta dall'espressione regolare precedente.

È possibile raccogliere tra parentesi tonde una parte dell'espressione regolare per memorizzarla ed utilizzarle in successive operazioni di sostituzione.

Il modulo `re` (*regular expressions*), descritto in modo esaustivo sulla documentazione on-line disponibile all'indirizzo `https://docs.python.org/3.7/library/re.html`, offre un insieme di funzioni che ci permettono di eseguire operazioni di *pattern matching* e di *pattern substitution* utilizzando delle espressioni regolari; tra queste evidenziamo le seguenti:

Funzioni del modulo **re** per le operazioni di **pattern matching** e di **pattern substitution** con espressioni regolari

- «`findall(`*pattern*`,`*string*`)`»: restituisce una lista di sottostringhe che corrispondono al pattern descritto dall'espressione regolare;
- «`search(`*pattern*`,`*string*`)`»: restituisce la prima occorrenza di una sottostringa che corrisponde all'espressione regolare specificata;
- «`split(`*pattern*`,`*string*`)`»: restituisce una lista di sottostringhe che, nella stringa specificata come argomento, sono separate l'una dall'altra da una sequenza di caratteri corrispondete all'espressione regolare;
- «`sub(`*pattern*`,`*subst*`,`*string*`)`»:

Riportiamo di seguito alcuni esempi per l'uso delle precedenti funzioni:

```
>>> re.findall('IN\d\d\d', 'AM220 IN110 FS220 IN440 GE310')
['IN110', 'IN440']
>>> re.search('\w+220', 'AM220 IN110 FS220 IN440 GE310')
<re.Match object; span=(0, 5), match='AM220'>
>>> re.split(';', 'IN440;Aula M5;ore 9-11')
['IN440', 'Aula M5', 'ore 9-11']
>>> re.sub('pippo', 'pluto', 'qui pippo quo pippopippo qua')
'qui pluto quo plutopluto qua'
```

Riferimenti bibliografici

[1] Marco Beri, *Python 3*, ed. Apogeo, Milano, 2010.

[2] Marco Buttu, *Programmare con Python. Guida Completa*, ed. LSWR, 2014.

[3] Mark Lutz, *Imparare Python*, ed. O'Reilly – Tecniche Nuove, 2011.

I precedenti riferimenti bibliografici potrebbero far pensare che occorra chiamarsi Marco per poter scrivere libri o manuali sul linguaggio Python o anche solo per occuparsi di questo linguaggio di programmazione; ebbene, per rassicurare tutti coloro che non si chiamano Marco, possiamo affermare con certezza che questo non è affatto necessario. A dimostrazione di questa affermazione, come controesempio, citiamo il seguente libro disponibile gratuitamente sulla rete Internet:

[4] Allen Downey, *Pensare in Python. Come pensare da informatico*, Green Tea Press, 2015, https://github.com/AllenDowney/ThinkPythonItalian/blob/master/thinkpython_italian.pdf

In rete sono disponibili altre numerosissime risorse utili all'apprendimento del linguaggio Python e dei suoi moduli; tra i molti citiamo i seguenti:

[5] *MatPlotLib*, libreria per la visualizzazione di grafici 2D con i moduli pylab e pyplot: https://matplotlib.org

[6] *NumPy* è una libreria per il calcolo scientifico in Python, che offre un ampio insieme di strutture dati e funzioni matematiche: http://www.numpy.org

[7] John Zelle, *graphics.py*, una libreria estremamente semplice e potente per la visualizzazione di componenti grafiche: https://mcsp.wartburg.edu/zelle/python/graphics/graphics.pdf