

Capitolo 4

I tipi di dati

Dopo tante “lungaggini burocratiche”, entriamo finalmente nel vivo del linguaggio. Entriamo alla grande, esplorando uno degli aspetti più peculiari di Python: i suoi spettacolari tipi di dati.

Cominceremo facendo rapidamente la conoscenza dei comandi *dir* e *type*, utilissimi quando si inizia a utilizzare Python, ma anche dopo! Quindi passeremo alle caratteristiche delle liste, delle stringhe, delle tuple, degli insiemi e dei dizionari (in Python sono rispettivamente i tipi *list*, *str*, *tuple*, *set* e *dict*), dei dati numerici. Infine daremo un’occhiata anche ad alcuni tipi un po’ speciali: *True*, *False* e *None* (che in italiano vogliono dire *Vero*, *Falso* e *Nulla*).



Nei prossimi paragrafi utilizzeremo IDLE per collaudare i vari esempi d’uso delle variabili. Come vedremo, per inizializzare una variabile basta scegliere un nome e poi assegnarle un valore. Questo è vero sia nella modalità interattiva sia nella scrittura di codice sorgente. Attenzione però: anche se non siamo obbligati a dichiarare una variabile, non possiamo usarne una non ancora inizializzata. Python ci lascia la massima libertà, ma cerca sempre di ridurre al minimo ogni possibilità di errore.

Il comando *dir*

Il comando *dir* si utilizza per lo più in modalità interattiva. Ci permette di visualizzare un elenco degli attributi dell’oggetto passato come argomento, qualunque esso sia.

Vediamolo all’opera con un modulo:

```
>>> import smtplib
>>> dir(smtplib)
['CRLF', 'OLDSTYLE_AUTH', 'SMTP', 'SMTPAuthenticationError',
'SMTPConnectError',
'SMTPDataError', 'SMTPException', 'SMTPHeloError',
```

```
'SMTPRecipientsRefused', 'SMTPResponseException',
'SMTPSenderRefused', 'SMTPServerDisconnected',
'SMTP_PORT', 'SSLFakeFile', 'SSLFakeSocket',
'__all__', '__builtins__', '__doc__', '__file__',
'__name__', 'base64', 'email', 'encode_base64',
'hmac', 'quoteaddr', 'quotedata', 're', 'socket',
'stderr']
>>>
```

Tutti gli elementi della lista sono funzioni o costanti del modulo `smtplib` (una possibilità fondamentale quando vorremo inviare un messaggio email da un nostro programma; abbiamo già visto all'opera questa possibilità in un esempio del Capitolo 1).



Spesso, molti degli elementi restituiti da `dir` sono delle stringhe che presentano una sequenza di due caratteri “_” all’inizio e alla fine. Per ora possiamo tranquillamente ignorarli: sono metodi particolari che vengono richiamati dall’interprete per le operazioni standard. Per esempio il metodo `__str__` di un dato oggetto viene richiamato da Python ogni volta che è richiesta la conversione in stringa dell’oggetto o una sua rappresentazione visualizzabile dal comando `print`.

Vediamo ora l’output di `dir` con una stringa:

```
>>> s="Ciao"
>>> dir(s)
['_add_', '__class__', '__contains__',
'_delattr_', '__doc__', '__eq__', '__ge__',
'_getattr_', '__getitem__',
'_getnewargs_', '__getslice__', '__gt__',
'_hash_', '__init__', '__le__', '__len__',
'_lt_', '__mod__', '__mul__', '__ne__',
'_new_', '__reduce__', '__reduce_ex__',
'_repr_', '__rmod__', '__rmul__', '__setattr__',
'_str_', 'capitalize', 'center', 'count',
'decode', 'encode', 'endswith', 'expandtabs',
'find', 'index', 'isalnum', 'isalpha', 'isdigit',
'islower', 'isspace', 'istitle', 'isupper', 'join',
'ljust', 'lower', 'lstrip', 'partition', 'replace',
'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',
'rstrip', 'split', 'splitlines', 'startswith',
'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
>>>
```

Non facciamoci spaventare da questo lungo elenco: Python è un linguaggio potente e i suoi tipi di dati non sono da meno. Nel paragrafo dedicato alle stringhe vedremo in dettaglio il funzionamento della maggior parte di questi metodi.

Il comando `type`

Il comando `type` visualizza il tipo dell'oggetto che viene passato come argomento. Vediamo alcuni rapidi esempi:

```
>>> n=1
>>> type(n)
<type 'int'>
>>> s="ciao"
>>> type(s)
<type 'str'>
>>> import smtplib
>>> type(smtplib)
<type 'module'>
>>>
```

Ogni qualvolta avremo un dubbio su un elemento restituito da `dir` (sarà una costante, una funzione oppure una classe?), `type` ci verrà in soccorso.

Proviamo a vedere come ci può aiutare con alcuni degli elementi restituiti da `dir` per il modulo `smtplib`:

```
>>> import smtplib
>>> dir(smtplib)
['CRLF', 'OLDSTYLE_AUTH', 'SMTP',
'SMTPAuthenticationError', 'SMTPConnectError',
'SMTPDataError', 'SMTPException', 'SMTPHeloError',
'SMTPRecipientsRefused', 'SMTPResponseException',
'SMTPSenderRefused', 'SMTPServerDisconnected',
'SMTP_PORT', 'SSLFakeFile', 'SSLFakeSocket',
'__all__', '__builtins__', '__doc__', '__file__',
'__name__', 'base64', 'email', 'encode_base64',
'hmac', 'quoteaddr', 'quotedata', 're', 'socket',
'stderr']
>>>
```

Cosa sarà mai l'attributo `CRLF`?

```
>>> type(smtplib.CRLF)
<type 'str'>
>>>
```

Una stringa! Va bene, era facile, ci si poteva arrivare dal nome. Infatti è la costante che contiene i codici CR e LF, che stanno per *Carriage Return* (ritorno carrello) e *Line Feed* (nuova riga).



I caratteri CR e LF sono ben conosciuti a tutti coloro che hanno avuto a che fare con il trasferimento di file tra sistemi Unix e sistemi Windows. Unix utilizza come terminatore di una riga dei file di testo (e quindi anche dei file di codice sorgente) il solo codice LF, mentre Windows utilizza la sequenza CR-LF. Questo fatto può essere di per sé fonte di problemi, ma la situazione è a volte peggiorata da alcuni programmi di trasferimento file, che effettuano automaticamente questa conversione. Questa traduzione può essere utile per i file di testo, ma se viene erroneamente effettuata per i file binari (per esempio per i file compressi in formato ZIP) il risultato sarà quasi sempre un discreto mal di testa per il malcapitato che ne è vittima.

SMTP che cosa rappresenta?

```
>>> type(smtplib.SMTP)
<type 'classobj'>
>>>
```

Una classe! Ma quali metodi conterrà questa classe? Riproviamo con `dir`:

```
>>> dir(smtplib.SMTP)
'docmd', 'does_esmtp', 'ehlo', 'ehlo_resp', 'expn',
'file', 'getreply', 'has_extn', 'helo',
'helo_resp', 'help', 'login', 'mail', 'noop',
'putcmd', 'quit', 'rcpt', 'rset', 'send',
'sendmail', 'set_debuglevel', 'starttls', 'verify',
'vrfy']
>>>
```

A questo punto, con un po' di immaginazione, possiamo intuire che `sendmail` è il metodo che serve per inviare un messaggio di posta elettronica.

Spesso ci sarà capitato di dover consultare la documentazione di un linguaggio di programmazione per farci ricordare il nome di un metodo o di una costante: con Python, `dir` e `type` ci eviteranno un sacco di volte questa fatica...



La documentazione di Python è molto ben realizzata ma, purtroppo, è disponibile solo in inglese. Per fortuna esiste anche un sito dedicato alla traduzione in italiano della documentazione ufficiale: <http://www.python.it>. È probabile che non vi troverete l'ultima versione tradotta, ma anche la penultima sarà più che sufficiente per risolvere la stragrande maggioranza dei dubbi.

Le liste

Cosa ci fa venire in mente la parola *lista*? Quasi certamente penseremo alla lista della spesa. Quali sono le sue caratteristiche principali? In estrema sintesi possiamo affermare che una lista è un elenco ordinato che contiene elementi eterogenei, ovvero di diverso tipo.

Bene, in Python una lista è proprio questo: un elenco ordinato di elementi eterogenei.

Proviamo subito a definire una semplice lista contenente qualche valore numerico scelto a caso:

```
>>> elenco = [23, 9, 1964]
>>> elenco
[23, 9, 1964]
>>>
```



In questo esempio dobbiamo prestare attenzione ai caratteri “[” e “]”: Sono proprio queste ultimi, le parentesi quadre, che indicano a Python che vogliamo creare una lista.

Sì, facile; ma non avevamo detto che la lista poteva contenere elementi eterogenei?

```
>>> elenco = [23, 9, 1964]
>>> varie = [1, "pippo", elenco]
>>>
```

Questo esempio è un pochino più interessante: il primo elemento della lista è un numero, il secondo è una stringa e il terzo è un'altra lista. Proviamo ora a visualizzare il contenuto della nuova lista:

```
>>> varie
[1, 'pippo', [23, 9, 1964]]
>>>
```

Abbiamo definito una lista che contiene un'altra lista. Non male.

Non ci basta? Allora esageriamo:

```
>>> import smtplib
>>> varie.append(smtplib)
>>> varie
[1, 'pippo', [23, 9, 1964], <module 'smtplib' from
'C:\python25\lib\smtplib.py'>]
>>>
```

Ora la nostra lista, oltre a contenere dei valori semplici e un'altra lista, contiene addirittura una libreria importata.

Quello di inserire in una lista un modulo importato è un colpo a effetto (probabilmente inutile). Abbiamo però visto il funzionamento del metodo `append`, che aggiunge un elemento in fondo alla lista.



In Python il punto separa un oggetto da un suo attributo o da un suo metodo.

Bene, ora che abbiamo imparato a creare le liste, come possiamo estrarre un elemento a piacere? Semplice:

```
>>> elenco = [23, 9, 1964]
>>> elenco[0]
23
>>>
```

Per estrarre l'elemento nella prima posizione abbiamo utilizzato l'indice 0. Questo perché in Python la numerazione delle posizioni all'interno di una lista parte da 0: data una lista di n elementi, questi sono numerati da 0 a $n - 1$.

Che cosa succede se specifichiamo un indice che non esiste? Viene scatenato un errore o più precisamente un'eccezione, argomento al quale è dedicato un intero capitolo:

```
>>> elenco = [23, 9, 1964]
>>> elenco[4]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
>>>
```

L'errore *"list index out of range"* indica proprio il fatto che l'indice utilizzato si trova al di fuori della "portata" della lista.

Come possiamo estrarre l'elemento che si trova nell'ultima posizione?

In questo caso abbiamo due possibilità:

```
>>> elenco = [23, 9, 1964]
>>> elenco[2]
1964
>>> elenco[-1]
1964
>>>
```

Se conosciamo la lunghezza della lista, possiamo indicare direttamente l'indice dell'ultimo elemento; in alternativa possiamo, assai più comodamente, indicare l'indice -1. Python sa che quando l'indice è negativo deve partire dal fondo. Gli elementi di una lista di lunghezza n possono essere individuati in ordine inverso con un indice che va da -1 a $-n$.

A proposito di dimensioni, la funzione `len` ci permette di scoprire la lunghezza di una lista:

```
>>> elenco = [23, 9, 1964]
>>> len(elenco)
3
>>>
```

Possiamo anche estrarre più elementi in un colpo solo? Ma certo, basta specificare due indici, separati dal simbolo ":":

```
>>> elenco = [23, 9, 1964]
>>> elenco[0:2]
[23, 9]
>>> elenco[1:3]
[9, 1964]
```

Possiamo usare gli indici negativi anche con il simbolo ":":

```
>>> elenco[1:-1]
[9]
>>> elenco[0:-1]
[23, 9]
>>>
```

Possiamo perfino omettere uno dei due valori, a sinistra o a destra di ":"; in tal caso Python raggiunge automaticamente il primo o l'ultimo indice della lista:

```
>>> elenco = [23, 9, 1964]
>>> elenco[1:]
[9, 1964]
>>> elenco[:2]
[23, 9]
```

```
[23, 9]
>>> elenco[:]
[23, 9, 1964]
>>>
```

Quest'ultimo esempio è da tenere a mente in quanto rappresenta il modo più comodo e veloce per duplicare una lista:

```
>>> elenco = [23, 9, 1964]
>>> elenco2 = elenco[:]
>>> elenco2
[23, 9, 1964]
>>>
```

Questo particolare metodo di accesso a uno o più elementi di una lista è detto *slicing* (da *to slice* = affettare); si tratta di uno di quegli aspetti di Python che lo rendono così piacevole da utilizzare.



In Python lo slicing è utilizzabile con tutti i tipi di dati che sono sequenziali. Per esempio è utilizzabile anche con le stringhe che, come vedremo nel prossimo paragrafo, possono essere considerate come una sequenza ordinata di caratteri.

Lo *slicing* può essere usato anche per assegnare in un colpo solo più elementi di una lista:

```
>>> elenco = [23, 9, 1964]
>>> elenco[0:2] = [13, 8]
>>> elenco
[13, 8, 1964]
>>>
```

oppure per inserire nuovi elementi:

```
>>> elenco = ['a', 'b', 'c']
>>> elenco[1:1] = [1, 2, 3]
>>> elenco
['a', 1, 2, 3, 'b', 'c']
>>>
```

o anche per cancellare degli elementi:

```
>>> elenco = ['a', 1, 2, 'b', 'c']
>>> elenco[1:3] = []
>>> elenco
['a', 'b', 'c']
>>>
```

Abbinando fra loro questi ultimi esempi, proviamo a immaginare come sia possibile cancellare gli ultimi due elementi di una lista. Elementare, Watson:

```
>>> elenco = ['a', 'b', 'c', 1, 2]
>>> elenco[-2:] = []
>>> elenco
['a', 'b', 'c']
>>>
```

In questo paragrafo abbiamo già visto all'opera il metodo `append`. Ma quali altri metodi possono essere applicati a una lista? Come facciamo a scoprirlo?

Ebbene sì: con il nostro vecchio amico `dir`:

```
>>> varie = [1, 'marco', [23, 9, 1964] ]
>>> dir(varie)
['_add_', '__class__', '__contains__',
 '__delattr__', '__delitem__', '__delslice__',
 '__doc__', '__eq__', '__ge__', '__getattr__',
 '__getitem__', '__getslice__', '__gt__',
 '__hash__', '__iadd__', '__imul__', '__init__',
 '__iter__', '__le__', '__len__', '__lt__',
 '__mul__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__reversed__',
 '__rmul__', '__setattr__', '__setitem__',
 '__setslice__', '__str__', 'append', 'count',
 'extend', 'index', 'insert', 'pop', 'remove',
 'reverse', 'sort']
>>>
```

Ignoriamo tranquillamente gli attributi che iniziano e terminano con la sequenza “`__`” e vediamo in dettaglio gli altri: `append`, `count`, `extend`, `index`, `insert`, `pop`, `remove`, `reverse` e `sort`.

Sappiamo già usare il primo, `append`: consente di aggiungere un nuovo elemento in fondo alla lista.

`insert` permette di inserire un elemento nella posizione desiderata:

```
>>> varie = ['pluto', 'pippo']
>>> varie.insert(0, 'nuovo')
>>> varie
['nuovo', 'pluto', 'pippo']
>>>
```

Per inserire un elemento in fondo alla lista possiamo usare un numero qualsiasi, uguale o maggiore alla lunghezza della lista.

Quindi possiamo impiegare la seguente forma:

```
>>> varie = ['pluto', 'pippo']
>>> varie.insert(2, 'fondo')
>>> varie
['pluto', 'pippo', 'fondo']
>>>
```

ma anche:

```
>>> varie = ['pluto', 'pippo']
>>> varie.insert(42, 'fondo')
>>> varie
['pluto', 'pippo', 'fondo']
>>>
```

Il metodo `extend` accetta come parametro una seconda lista, che viene “appesa” in fondo alla lista principale:

```
>>> varie = ['pluto', 'pippo']
>>> aggiunta = ['a', 'b', 'c']
>>> varie.extend(aggiunta)
>>> varie
['pluto', 'pippo', 'a', 'b', 'c']
>>>
```

La differenza tra i metodi `append` ed `extend` è chiara: il primo aggiunge in fondo alla lista un elemento semplice, mentre il secondo aggiunge in fondo alla lista principale un'intera lista; in pratica è un po' come usare un `append` per tutti gli elementi della lista da aggiungere.

Siete stanchi di sentir parlare di metodi che allungano la nostra lista? È comprensibile, sicuramente ci capiterà anche di dover rimuovere qualche elemento dalla lista. I metodi `remove` e `pop` servono proprio a questo.

Il metodo `remove` è speculare rispetto a `insert`, in quanto rimuove l'elemento corrispondente all'indice passato come argomento:

```
>>> elenco = ['a', 'b', 1, 'c']
>>> elenco.remove(2)
>>> elenco
['a', 'b', 'c']
>>>
```

Il metodo `pop` (onomatopeico) è molto particolare, in quanto abbina le due funzionalità di estrazione e rimozione di un elemento. Può essere usato con o senza indice; in quest'ultimo caso estrae l'ultimo elemento

della lista:

```
>>> elenco = ['a', 'b', 'c']
>>> elenco.pop()
'c'
>>> elenco
['a', 'b']
>>> elenco.pop(0)
'a'
>>> elenco
['b']
>>>
```

Il metodo `count` conta gli elementi della lista che sono uguali all'argomento passato:

```
>>> varie = [1, 2, 3, 4, 1]
>>> varie.count(1)
2
>>> varie.count(5)
0
>>>
```

Attenzione, perché `count` non esplora ricorsivamente eventuali liste presenti all'interno della lista principale:

```
>>> varie = [1, 'marco', [23, 9, 1964] ]
>>> varie.count(23)
0
>>>
```

L'elemento 23 è, sì, presente, ma solo all'interno della lista che rappresenta il terzo elemento della lista principale, per questo motivo non viene trovato da `count`.

`index` restituisce la posizione all'interno della lista dell'elemento passato come argomento:

```
>>> elenco = ['a', 'b', 'c']
>>> elenco.index('b')
1
>>>
```

Attenzione, perché se l'elemento non esiste viene scatenato un errore (sì, giusto, si chiama eccezione):

```
>>> elenco = ['a', 'b', 'c']
>>> elenco.index('e')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: list.index(x): x not in list
>>>
```

Il metodo `reverse` inverte semplicemente una lista:

```
>>> elenco = ['c', 'b', 'a']
>>> elenco.reverse()
>>> elenco
['a', 'b', 'c']
>>>
```

Il metodo `sort` dispone in ordine alfabetico e/o numerico gli elementi della lista:

```
>>> elenco = [44, 'c', 1, 'b', 'a']
>>> elenco.sort()
>>> elenco
[1, 44, 'a', 'b', 'c']
>>>
```

I numeri vengono disposti in ordine all'inizio della lista, quindi vengono elencati i valori alfanumerici, in ordine alfabetico. Normalmente il metodo `sort` dovrebbe essere applicato a liste contenenti elementi omogenei, ma nulla ci vieta di impiegarlo anche su liste di elementi eterogenei.

Le stringhe

Senz'altro qualcuno si domanderà che cosa ci potrà mai essere di particolare e innovativo nelle stringhe Python. Effettivamente, di solito una stringa non è altro che una sequenza di caratteri. Ma attenzione: in Python tutto ciò che è una sequenza ha accesso alle potentissime funzionalità di *slicing* che abbiamo appena visto all'opera per le liste. Inoltre una stringa, come ogni altro elemento in Python, è un oggetto, dotato di metodi che ne permettono una facile gestione.

Vediamo subito qualche esempio di *slicing* con le stringhe in Python:

```
>>> s = "0123456789"
>>> s[:5] # I primi 5 caratteri
'01234'
>>> s[-5:] # Gli ultimi 5 caratteri
'56789'
```

```
>>> s[1:-1] # Tutti i caratteri tranne il primo e
l'ultimo
'12345678'
>>>
```



In Python il carattere “#” indica l’inizio di un commento. Inserire un commento in modalità interattiva ha senso solo quando occorre fare degli esempi; sarà invece molto più utile durante la stesura di programmi veri e propri. Lo sappiamo tutti che è utile commentare bene i propri file di codice sorgente, vero?

C’è ancora un tipo di *slicing* che non abbiamo esaminato. Possiamo utilizzare un terzo parametro che indica, oltre all’inizio e alla fine della nostra “fetta”, ogni quanti elementi dobbiamo estrarne uno. Forse è meglio chiarire il concetto con un esempio:

```
>>> s = "0123456789"
>>> s[::2] # Prendi un carattere ogni due
'02468'
>>> s[1::2] # Questa volta parti dal secondo
'13579'
>>>
```

Un’importante differenza rispetto alle liste consiste nel fatto che le stringhe sono oggetti immutabili. Non possiamo assegnare un nuovo valore a una parte di una stringa:

```
>>> s = "ab-de"
>>> s[2] = "c"
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    s[2] = "c"
TypeError: 'str' object does not support item
assignment
>>>
```

Per poterlo fare dobbiamo riassegnare l’intera stringa:

```
>>> s = "ab-de"
>>> s = s[:2] + "c" + s[3:]
>>> s
'abcde'
>>>
```

Possiamo delimitare una stringa con doppi apici; in tal caso la stringa può anche contenere degli apici singoli:

```
>>> s = "Mi piace l'uva"
>>>
```

Oppure possiamo utilizzare come delimitatore l'apice singolo e allora all'interno della stringa potremo utilizzare gli apici doppi:

```
>>> s = 'Ho visto i "Monty Python": che risate!'
>>>
```

Per inserire in una stringa un carattere di "a capo" dobbiamo utilizzare il simbolo *backslash* (la barra rovesciata), seguito dal carattere "n", ovvero la sequenza "\n":

```
>>> s = "1\n2\n3\nVia!"
>>> print s
1
2
3
Via!
>>>
```



La sequenza "\n" è un simbolo speciale. Ne esistono diversi altri ma quelli più importanti sono per l'appunto "\n" Line Feed (a capo), "\r" Carriage Return (ritorno carrello), "\b" Backspace (indietro di un carattere), "\t" Tab (carattere di tabulazione) e "\xHH" che permette di inserire il simbolo il cui codice ASCII è il valore esadecimale HH.

Spesso ci capiterà di dover scrivere delle stringhe su più righe, come stringhe di documentazione dei programmi. È possibile farlo utilizzando una sequenza di tre apici doppi o singoli per racchiudere una stringa multiriga.

```
>>> s = """
Facile scrivere stringhe
su più righe...
E posso usare apici: 'spam'
O doppi apici: "egg"
E finisco qui.
"""

>>> print s
```

```
Facile scrivere stringhe
su più righe...
E posso usare apici: 'spam'
O doppi apici: "egg"
E finisco qui.
```

```
>>>
```

Per concatenare tra di loro due stringhe possiamo usare il simbolo +:

```
>>> s = "Precipite"
>>> p = "volissimevolmente"
>>> s + p
'Precipitevolissimevolmente'
>>>
```

Possiamo perfino moltiplicare una stringa, usando il simbolo *:

```
>>> s = "spam"
>>> s * 5
'spamspamspamspamspam'
>>>
```

Anche le stringhe offrono dei metodi che permettono di manipolarle con facilità. Sono davvero molti: `capitalize`, `center`, `count`, `decode`, `encode`, `endswith`, `expandtabs`, `find`, `index`, `isalnum`, `isalpha`, `isdigit`, `islower`, `isspace`, `istitle`, `isupper`, `join`, `ljust`, `lower`, `lstrip`, `partition`, `replace`, `rfind`, `rindex`, `rjust`, `rpartition`, `rsplit`, `rstrip`, `split`, `splitlines`, `startswith`, `strip`, `swapcase`, `title`, `translate`, `upper`, `zfill`.

Esamineremo in dettaglio solo i principali; la documentazione di Python ci può venire in aiuto per scoprire il funzionamento degli altri metodi (ma un pizzico di intuizione e il prompt di Python spesso sarà più che sufficiente).

`find` ricerca un carattere in una stringa:

```
>>> s = "Troviamo la x in questa stringa"
>>> s.find("x")
12
>>>
```

`strip` rimuove gli spazi all'inizio e alla fine di una stringa:

```
>>> s = " egg "
>>> s.strip()
'egg'
>>>
```

replace sostituisce una sottostringa con un'altra:

```
>>> s = "Non mi piacciono i Monty Python"
>>> s.replace("Non", "Ma come")
'Ma come mi piacciono i Monty Python'
>>>
```

split e join vanno di pari passo: il primo spezza una stringa in una lista di più parti, mentre il secondo riunisce una lista per formare un'unica stringa:

```
>>> s = "uno due tre"
>>> s.split(" ")
['uno', 'due', 'tre']
>>>
>>> "/" .join(["12", "10", "1492"])
'12/10/1492'
>>>
```

Forse l'uso del metodo join ci sembrerà un po' strano; in realtà, se ci pensiamo bene, è perfettamente corretto. join non può essere un metodo della lista; deve essere un metodo della stringa utilizzata per unire gli elementi della lista.

Non possiamo usare split specificando come argomento una stringa vuota per separare tutti i caratteri di una stringa. Per ottenere questo risultato possiamo però usare la funzione predefinita list:

```
>>> s = "123"
>>> list(s)
['1', '2', '3']
>>>
```

Le tuple

Le tuple sono sequenze di oggetti eterogenei (in questo senso sono simili alle liste) ma sono immutabili (e in questo senso sono simili alle stringhe). Vediamo la nostra prima tupla:

```
>>> t = ('basta', 'con', 'le', 'liste')
>>> t
('basta', 'con', 'le', 'liste')
>>>
```



Le parentesi tonde "(" e ")" indicano a Python che vogliamo creare una tupla.

Anche con le tuple possiamo usare lo *slicing*:

```
>>> t = (1, 2, 3, 'stella')
>>> t[:3]
(1, 2, 3)
>>> t[-1]
'stella'
>>>
```

Ma non possiamo modificarne alcuna parte:

```
>>> t = (1, 'x', 3, 'stella')
>>> t[1] = 2

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    t[1] = 'x'
TypeError: 'tuple' object does not support item
assignment
>>>
```

Le tuple non hanno alcun metodo e l'unico modo che abbiamo per sapere se un elemento è presente in una tupla consiste nell'impiegare l'operatore `in`:

```
>>> fibonacci = (1, 1, 2, 3, 5, 8, 13, 21)
>>> 8 in fibonacci
True
>>> 9 in fibonacci
False
>>>
```



True e False sono i valori booleani vero e falso in Python: li vedremo più in dettaglio tra qualche paragrafo.

Come possiamo definire una tupla contenente un solo elemento? Con questo accorgimento:

```
>>> singolo = (1) # No, non così...
>>> singolo
1
>>> singolo = (1,) # Così invece sì!
>>> singolo
(1,)
>>>
```



Qualcuno si domanderà a cosa servono le tuple, visto che non offrono niente in più delle liste ma presentano molte funzionalità in meno. È una domanda legittima: il motivo principale sta proprio nella loro immutabilità che gli permette di fungere da indici per i dizionari (argomento che verrà trattato in uno dei prossimi paragrafi).

Gli insiemi

I *set* (insiemi) sono un potente tipo di dati che possiamo usare quando dobbiamo gestire gruppi di elementi non ordinati e senza duplicati.

A differenza dei tipi di dati che abbiamo visto fino a questo momento, per creare un *set* dobbiamo usare una parola chiave specifica invece che un carattere speciale:

```
>>> insieme = set(['pippo', 'pluto', 'paperino'])
>>> insieme
set(['pippo', 'paperino', 'pluto'])
>>>
```

La parola chiave da impiegare per creare un insieme è proprio `set`. L'argomento può essere una sequenza qualsiasi.

Quindi possiamo impiegare una lista come nell'esempio precedente, ma anche una tupla o una stringa:

```
>>> lettere = set("hello")
>>> lettere
set(['h', 'e', 'l', 'o'])
>>>
```



Potete notare come la stringa "hello", pur contenendo 5 caratteri ha generato un insieme di soli quattro caratteri (il carattere "l" appare una volta sola nell'insieme): gli insiemi non possono contenere elementi duplicati.

Le operazioni che possiamo effettuare sugli insiemi sono tutte quelle classiche dell'insiemistica tradizionale.

L'insieme *unione*, con il carattere `|` (la barra verticale):

```
>>> piccoli = set(['topo', 'mosca'])
>>> grandi = set(['orso', 'balena'])
>>> piccoli | grandi
set(['topo', 'orso', 'mosca', 'balena'])
>>>
```

L'insieme *differenza*, con il carattere "-" (il meno):

```
>>> piccoli = set(['topo', 'mosca'])
>>> bianchi = set(['colomba', 'topo'])
>>> piccoli - bianchi
set(['mosca'])
>>>
```

L'insieme *intersezione*, con il carattere "&" (la "e" commerciale):

```
>>> piccoli = set(['topo', 'mosca'])
>>> bianchi = set(['colomba', 'topo'])
>>> piccoli & bianchi
set(['topo'])
>>>
```

L'insieme *differenza* simmetrica, meglio noto come *xor*, costituito da tutti gli elementi che sono presenti solo in uno dei due insiemi ma non in entrambi, con il carattere "^" (l'accento circonflesso):

```
>>> piccoli = set(['topo', 'mosca'])
>>> bianchi = set(['colomba', 'topo'])
>>> piccoli ^ grandi
set(['colomba', 'mosca'])
>>>
```

I dizionari

Se possiamo eleggere il tipo di dati di Python di cui sentiamo di più la mancanza quando (purtroppo) abbiamo a che fare con un altro linguaggio di programmazione che ne è privo, questo è proprio il *dictionary* (dizionario).

La definizione ufficiale di un dizionario è *array associativo*. Ma come quasi tutte le definizioni belle e pompose, non ci dice molto.

Per renderci conto di che cosa stiamo parlando, possiamo esaminare un esempio: dobbiamo creare un dizionario e assegnargli qualche valore:

```
>>> rubrica = dict()
>>> rubrica['marco'] = '333-123123'
>>> rubrica['lucia'] = '345-888555'
>>>
```

Proviamo a visualizzare il dizionario che abbiamo appena creato:

```
>>> rubrica
{'marco': '333-123123', 'lucia': '345-888555'}
>>>
```

Ora estraiamo un valore dal dizionario:

```
>>> rubrica['marco']
'333-123123'
>>>
```

Quindi proviamo ad aggiungere un nuovo valore:

```
>>> rubrica['difra'] = '332-610610'
>>>
```

Adesso cominciamo ad afferrare il concetto? Un dizionario è un insieme di oggetti che possiamo estrarre attraverso una chiave. La chiave in questione è quella che abbiamo utilizzato in fase di assegnamento.

Per elencare tutte le chiavi di un dizionario possiamo utilizzare il metodo *keys*:

```
>>> rubrica.keys()
['difra', 'marco', 'lucia']
>>>
```

Come possiamo sapere se una determinata chiave è presente nel dizionario? Il metodo *has_key* fa al caso nostro:

```
>>> rubrica.has_key('silvio')
False
>>> rubrica.has_key('marco')
True
>>>
```

Ovviamente possiamo anche cancellare un elemento dal dizionario; basta specificare la sua chiave con il comando *del*:

```
>>> del rubrica['difra']
>>> rubrica
{'marco': '333-123123', 'lucia': '345-888555'}
>>>
```

Che cosa possiamo assegnare a un elemento di un dizionario? Qualsiasi oggetto Python, perfino una funzione o una classe (anche se non abbiamo ancora visto come si usano).

E che cosa possiamo utilizzare come chiave di un elemento? Qualsiasi oggetto immutabile, quindi una stringa, un intero ma anche una tupla.



Una tupla può essere utilizzata come chiave in un dizionario solo se non contiene, direttamente o indirettamente, altri oggetti modificabili. Una tupla che contiene una lista non può quindi essere usata come chiave di un dizionario. Non si può usare neppure un insieme, perché è un oggetto modificabile. Esiste però una versione immutabile che si chiama frozenset (letteralmente “insieme congelato”) che è per i set quello che le tuple sono per le liste; è immutabile e quindi può essere usato come chiave di un dizionario.

Anche i dizionari hanno vari metodi (nessuno ne dubitava): `clear`, `copy`, `fromkeys`, `get`, `has_key`, `items`, `iteritems`, `iterkeys`, `itervalues`, `keys`, `pop`, `popitem`, `setdefault`, `update`, `values`.

Alcuni di essi (come `keys` e `has_key`) sono già stati brevemente introdotti negli esempi precedenti. Tra tutti gli altri esamineremo solo i più utili.

`clear` cancella tutti gli elementi di un dizionario.

```
>>> biblio = {'bibbia': 0,
              'peopleware': 42 }
>>> biblio
{'peopleware': 42, 'bibbia': 0}
>>> biblio.clear()
>>> biblio
{}
>>>
```



Un dizionario può essere inizializzato con le parentesi graffe. Un dizionario vuoto è indicato da due parentesi graffe vuote.

`get` permette di estrarre un valore di default qualora la chiave specificata non sia presente:

```
>>> sport = {'corsa': 10,
            'basket': 13 }
>>> sport.get('corsa', 0)
10
>>> sport.get('tennis', 0)
0
>>>
```

`setdefault` permette di estrarre un valore di default, ma aggiungendolo al dizionario qualora la chiave specificata non esista:

```
>>> ricetta = {'uova': 2,
               'farina': 100,
               'burro': 30 }
>>> ricetta.setdefault('zucchero', 200)
200
>>> ricetta
{'zucchero': 200, 'burro': 30, 'farina': 100,
 'uova': 2}
>>>
```

`values` è il gemello speculare di `keys`: invece delle chiavi estrae tutti gli elementi:

```
>>> azioni = {'fiat': 2,
              'link': 99,
              'oracle': 3}
>>> azioni.values()
[2, 3, 99]
>>>
```

Infine `items` è la “somma” di `keys` e `values`, infatti restituisce la lista delle coppie chiave/valore del dizionario (sotto forma di tuple):

```
>>> frutta = {'mele': 123,
              'banane': 5}
>>> frutta.items()
[('banane', 5), ('mele', 123)]
>>>
```

I numeri

Cosa mai potrà aggiungere Python a un tipo di dati così comune come i dati numerici? Cominciamo dagli interi e proviamo a calcolare la decimillesima potenza di 2:

```
>>> 2 ** 10000
```

Nella Figura 4.1 possiamo vedere il risultato dell'operazione.

Proprio così: non c'è limite alle dimensioni di un numero intero. O meglio, il limite è solo quello fisico della memoria del computer su cui stiamo utilizzando Python.

```

Python Shell
File Edit Shell Debug Options Windows Help
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 1.2
>>> 2**10000
19950631168807583488837421626835850838234968318861924548520089498529438830221946631919
96166403619459789933112942320912427155649134941378111759378593209632395785573004679379
45267652465512660598955205500869181933115425086084606181046855090748660896248880904898
94838009253941633257850621568309473902556912388065225096643874441046759871626985453222
86853816169931577562964076283688076073222853509164147618395638145896946389941084096053
62678210646214273333940365255656495306031426802349694003359343166514592567732796657756
06172582031407994198179607378245683762280037302885487251900803446458154650557929601414
8339216157345881392570953797691192778008269577356744441230620187578363255027283278927
07103738028663930314281332414016241956716905740614196543423246388012488561473052074319
92259611796250130992860241708340807605932320161268492288496255841312844061536738951487
11425631511108974551420331382020293164095759646475601040584584156670204496286701651506
1920631004186422275908670900574606417856951911456055068251250406007519842618980592371
18054444788072906395242548339221982707404473162376760846613033778706309803413197133493
6546227005631699374555082417809728109832913144035718775247685098572769379264332159939
9876886660808368837838027643282775172273657527447841122943897338108616074232352919748
13120197604178281965697475898164531258434135959862784130128185406283476649088690521047
5008082615823961985770122407044330580307586903931960460340497315658320867210591330090375
28234155397453943977155274552905102123109473216107534748257407752739863482984983407569
37955466638621874569499279016572103701364433135817214311791398222983845847334440270964
18285100507292774836455057863450110085298781238947392869954083434615880704395911898581
514577917714319698728131459483783202081474988217185801138907122825095058268174362205774
7592141765371568725614904582904992461028630081535583308130101987675856234343538955409
1756234008488875261626435686488351946372037729324009445624692325435040067802727383775
537640672689863624103749141096671855705759098100246789880178271925933812824219540283
02759408448955014676668389697996886241636313376393903373455801407636741877711055384225
7394991101864682196965816514851304942236994771476306915546821768287620036277725772378
13653316111968112807926694818872012986643660768551639605346022978715575179473852463694
46923087894265948217008051120322365496288169035739121368338393591756418733850510970721
6139154395909159815465441733631165693603112224993796999922678173235802311186264457529
91357581750081988392362846152498810889602322443621737716180863570154684840586223297928
5387562348655644053696262201896357102881236156751254333830327002909766865056857157505
5167251889919412971133769014991618131517154400772865057318955745092033018530484711381
83154073240533190384620840364217637039115506397890007428536721962809034779745333204663
68795868580237952218629210080742819551317948157624448298518461509704888027274721574688
131594750409732115080498190455803416826949787141311606321068639151168177430479259670937
6L
>>>

```

Figura 4.1 Un numero intero decisamente inusuale.

Ovviamente per questo motivo utilizzare anche i numeri in virgola mobile (attenzione però, in questo caso il limite c 'è e dipende dalla piattaforma che stiamo utilizzando):

```

>>> 2.3456 / 7.89
0.29728770595690751
>>>

```

Per i palati più esigenti esistono anche i numeri complessi, identificati dal suffisso j o J :

```

>>> a = -1j # unità immaginaria
>>> a.real

```

```
0.0
>>> a.imag
-1.0
>>> a ** 0.5 # radice quadrata di -1
(0.70710678118654757-0.70710678118654746j)
>>>
```

True, False e None

Anche Python, come ogni linguaggio che si rispetti, ha i suoi valori booleani vero e falso, che corrispondono alle stringhe True e False.

```
>>> numeri_pari = (2, 4, 6, 8)
>>> x = 1 in numeri_pari
>>> x
False
>>> y = 4 in numeri_pari
>>> y
True
>>>
```

Se vogliamo inizializzare un valore booleano dobbiamo ricordarci di non usare gli apici come per le stringhe, quindi:

```
>>> flag1 = True # Così va bene
>>> flag2 = "False" # Così non ci siamo proprio
>>>
```

Ed eccoci finalmente arrivati alla degna conclusione di questa lunga passeggiata tra i tipi di dati di Python: il nulla. Il tipo di dati None (nulla) può assumere un solo valore, uguale proprio a None. Viene utilizzato in tutte le situazioni in cui intendiamo indicare la mancanza di un valore definito. Per esempio possiamo utilizzarlo per inizializzare il valore di alcune variabili che devono esistere ma non hanno alcun valore iniziale.



Quando parleremo dei parametri opzionali delle funzioni vedremo come il tipo di dati None può essere utilizzato per inizializzarli in maniera pulita.

Per definire None, come per True e False, non dobbiamo mai usare gli apici:

```
>>> nulla = None # Così va bene
>>> qualcosa = "None" # Questo non è un vero None
>>>
```

Conversioni

Sicuramente ci capiterà di dover trasformare un valore da un tipo di dati a un altro; per esempio potremmo dover trasformare una lista in una tupla o un intero in una stringa e così via. Per fare questo dobbiamo utilizzare le funzioni predefinite `str`, `list`, `tuple`, `int`.

Vediamo qualche esempio:

```
>>> t = ('a', 'b', 'c') # Una tupla
>>> l = [1, 2, 3]       # Una lista
>>> s = "ciao"         # Una stringa
>>> i = 42              # Un intero
>>> str(i)              # Da intero a stringa
"42"
>>> tuple(l)           # Da lista a tupla
(1, 2, 3)
>>> list(t)            # Da tupla a lista
['a', 'b', 'c']
>>> int("1997")        # Da stringa a intero
1997
>>> int("FF", 16)      # Da esadecimale a intero
255
>>>
```