Python 3

01/02/2014 release #1.00

Premessa del traduttore:

Perché Python secondo me:

Perché racchiude in se il meglio di tutti gli altri linguaggi di programmazione, ponendo così il programmatore nello stato di grazia di non dover saltare da un linguaggio ad un' altro per poter sfruttare le rispettive potenzialità. Questo è il vantaggio che ha un nuovo linguaggio che si affaccia nel mondo dell'informatica, ma che dura poco in quanto verrà prima o poi superato a sua volta da altri linguaggi che implementeranno il meglio di quelli fatti in precedenza.

Anche in questo meccanismo Python ha una marcia in più. Infatti è passato dalla versione 2.x alla 3.x che pur simili, sono fra loro incompatibili. Questo può suonare strano mentre tutti si preoccupano di mantenere rigorosamente la compatibilità! Se Python non avesse spezzato il legame tra la 2 e la 3 per garantire la compatibilità, avrebbe di fatto legato la il suo destino alla versione 2. Invece la versione 3, include tante nuove migliorie e correzioni di difetti della 2. Qualcuno potrebbe obiettare che questo passaggio costa caro! La risposta è: meglio un po' di tempo per adeguarsi senza uscire da Python che imparare un linguaggio nuovo o pseudo tale. E' una scelta che può far discutere, ma sicuramente la meno invasiva e dispendiosa in termini di tempo-programmatore, e forse sarà così anche per una ipotetica versione 4.

Inoltre esso rappresenta una caratteristica speciale. Non gli importa nulla dell'ambiente grafico! O meglio, lascia agli altri il modo di implementare l'ambiente grafico non ponendo limiti di piattaforma o toolkit grafico. Infatti sono presenti moltissimi ambienti grafici a cui associare Python, da QTx, GTKx, TINKER, WxPython, JavaSwing ecc. ecc. Il tutto disponibile su praticamente tutte le piattaforme tenendo divisa la parte del codice da quella della grafica (più o meno). Per finire, ultimo ma non ultimo il suo codice estremamente compatto, che riduce il tempo di scrittura e la gestione degli errori minimizzate da robuste funzioni precostituite. Se questo non vi è sufficiente, allora potete evitare di leggere questa guida e aspettare che qualcosa vi cada dal cielo.

Sulla traduzione:

Questa traduzione di Python 3.x.x è resa libera per tutti gli usi e a tutti purché rispetti la licenza del documento originale essa è tratta dalla pagina http://docs.python.org/3/tutorial/index.html. E' nella forma in cui è, e non pretende di essere perfetta nella consapevolezza che di errori ve ne sono anche se sono stato piuttosto attento, ma avendo eseguito il tutto da solo senza alcun aiuto esterno, resta la consapevolezza che non potrà mai essere sia pur minimamente perfetta. Inoltre anche nella versione originale di questa guida, vi sono dei passaggi trattati in modo stringato che inducono ad una traduzione che può non esplicare quel concetto nella sua interezza. Inoltre ho eseguito anche la traduzione di quasi tutti gli esempi (cosa generalmente sconsigliata) per facilitare la comprensione in quanto molte istruzioni mescolate con il testo con l'inglese possono confondere il lettore. Mentre tradotte in italiano rendono il codice più comprensibile anche se esposto maggiormente a potenziali errori. Ciò detto, sono sicuro che sarò graziato per tutto quello che non va, ma dentro di me ho sentito il bisogno di fare questo lavoro in quanto ritengo Python veramente superbo in tutti i suoi aspetti con una documentazione si completa in inglese, ma carente in italiano specialmente per la versione 3.x. La dove qualcuno noti degli errori delle incongruenze o altro che possa essere migliorato può liberamente modificare questa guida oppure contattarmi e informarmi sul quanto a questo indirizzo di posta elettronica: python@mamex.it . Sarà comunque mio compito aggiornarla al meglio. Declino ogni responsabilità su eventuali malfunzionamenti di programmi tratti da questa guida e/o da cattiva traduzione. Chi vuole copiare diffondere questa guida è libero di farlo, sarebbe comunque gradita una citazione in tal caso. Spero che vi divertiate come mi sto divertendo io! Python è veramente grande! Buona lettura.

Il traduttore

Maurizio Da Lio

La Traduzione:

Guida introduttiva a Python 3

Python è un potente linguaggio di programmazione facile e divertente da imparare. Possiede strutture di dati di alto livello molto potenti e semplici da maneggiare utilizzando un' approccio *object-oriented* cioè **orientato agli oggetti**. Inoltre la sua sintassi è elegante ed ha la gestione dei tipi dinamica unitamente alla sua natura di tipo interprete, tutto questo rende Python un linguaggio ideale per lo scripting e lo sviluppo rapido di applicazioni in molte aree sulla maggior parte delle piattaforme.

L'interprete Python possiede un'ampia libreria standard liberamente disponibile in formato sorgente o in forma binaria per tutte le principali piattaforme disponibili sul sito Web di Python, http://www.python.org/, che possono essere utilizzate e distribuite liberamente. Lo stesso sito contiene anche collegamenti a molti moduli liberi prodotti da terzi per Python, programmi strumenti e documentazione aggiuntiva.

L'interprete di Python è facilmente estendibile permette di aggiungere nuove funzioni e tipi di dati implementati in C o C + + (o altri linguaggi richiamabili da C). Esso è adatto anche come linguaggio di estensione per applicazioni personalizzabili.
Questa guida, introduce informalmente il lettore ai concetti base e alle caratteristiche del linguaggio e di sistema. Aiuta ad utilizzare l'interprete in modo pratico per esperienza alla mano con tutti gli esempi pratici descritti in modo da poterla utilizzare anche senza l'utilizzo di una connessione.

Per una descrizione degli oggetti standard e dei suoi moduli, consultare la libreria standard di Python. *The Python Standard Library* fornisce una definizione più formale e dettagliata del linguaggio. Per scrivere estensioni in C o C + +, fare riferimento a *The Python Language Reference*. Inoltre ci sono moltissimi libri in lingua inglese ed alcuni in lingua italiana di ottimo livello per poter approfondire la sua conoscenza con tecniche altamente professionali. Altri importanti riferimenti si trovano a questi link: *Extending and Embedding the Python Interpreter* e *Python/C API Reference Manual*.

Questa guida, non intende essere esaustiva e coprire ogni singola caratteristica, o anche tutte le funzionalità di uso comune. Tende invece a introdurre molte fra le caratteristiche più notevoli di Python e vi darà una buona idea sul gusto e stile del linguaggio. Dopo averla letta, sarete in grado di leggere e scrivere moduli e programmi e sarete pronti ad imparare di più sui vari moduli della libreria Python descritta in in *The Python Standard Library* . Vale inoltre la pena dare un'occhiata al *Glossary*.

INDICE

1. Tanto per stimolarci un po.

Se lavorate molto al computer, vi sarete resi conto che molte delle operazioni che facciamo, potrebbero essere automatizzate alla fine di risparmiare tempo ed evitare errori, rendendo inoltre il lavoro più piacevole. Ad esempio, si potrebbe desiderare di eseguire una ricerca e sostituzione su un gran numero di file di testo, o rinominare e/o riorganizzare un gruppo di file di foto in un modo complicato. Forse vuoi scrivere un piccolo database personalizzato, o un'applicazione grafica GUI (**G**rafics **U**ser **I**nterface) specializzata oppure un semplice gioco.

Se sei uno sviluppatore di software professionale, potresti avere l'esigenza di lavorare con parecchie librerie C / C + + / Java, ma utilizzando linguaggi compilati come è noto, il ciclo usuale di scrittura / compilazione / test / ricompilazione, è molto lento e tedioso. Forse stai scrivendo una suite di un test per una libreria e scrivere e provare il codice diventa un compito noioso, o forse hai scritto una libreria che potrebbe essere usata come estensione per altri utenti potenziando così il linguaggio. Se questo è vero anche in parte, allora Python è quello di cui ai bisogno.
Si potrebbe scrivere uno script con una shell Unix o un file batch di Windows per alcuni di questi compiti, ma gli script di shell, sono più adatti per la rimozione o la modifica di file di testo ma non per le applicazioni GUI (**G**rafics **U**ser **I**nterface) giochi o per programmi interattivi. Si potrebbe in alternativa scrivere un programma in C + + / C o Java, ma tutto questo richiede un notevole tempo di sviluppo anche solo per un primo e semplice programma embrionale funzionante. Python viceversa è semplice da usare, e cosa da non sottovalutare **disponibile per Windows, Mac OS X Unix e quindi Linux.** Questo ti aiuterà a ottenere un lavoro fatto più velocemente e probabilmente migliore oltre che multi piattaforma.

Python è semplice da usare, ma non significa per questo che sia un linguaggio di programmazione dalle scarse potenzialità. Python per esempio offre un supporto per gli script veramente notevole assai migliore di qualsiasi semplice script di shell o batch. Questo non deve però trarre in inganno in quanto è solo uno dei lati positivi. Infatti esso è un linguaggio di programmazione vero e proprio di livello molto alto che racchiude in se tutte le migliori caratteristiche di altri linguaggi aggiungendo funzionalità non presenti in alcuni ed eliminando molti dei difetti.

Per esempio Python offre molto più controllo degli errori di C, e, essendo un linguaggio di livello molto alto, possiede tipi di dati di alto livello integrati, come array (matrici di dati) flessibili, dizionari, liste tuple e la possibilità di combinare queste strutture fra di loro (veramente potente). Queste implementazioni, sollevano il programmatore dal dover reinventare l'acqua calda e dalla gran parte degli errori che possono essere commessi in fase di programmazione a causa della complessità degli argomenti. Questo ha fatto di Python un linguaggio molto popolare, molto usato e molto diffuso.

Esso permette di suddividere il programma in moduli che possono essere riutilizzati in altri programmi Python. Viene fornito con una grande collezione di moduli standard e di una sterminata collezione di librerie prodotte da molti appassionati, che spaziano fra gli ambiti più disparati che potete usare come base dei vostri programmi e/o come esempi per iniziare ad imparare a programmare. Alcuni di questi moduli offrono cose come l' I/O (l'accesso ai file), chiamate di sistema, connessioni di rete, ambienti per interfacce grafiche utente come TK QT GTK ecc.

Python è un linguaggio interpretato, ciò significa che è possibile risparmiare molto tempo durante lo sviluppo del programma, perché non è necessaria la continua compilazione e conseguente linkaggio. L'interprete può essere utilizzato interattivamente il che lo rende facile per sperimentare le caratteristiche proprie, scrivere programmi usa e getta per testare funzioni durante lo sviluppo di un programma principale. Inoltre è anche un'utile calcolatrice.

Python consente ai programmi di essere scritti in modo compatto e di essere di facile lettura in quanto obbliga a scriverli in un certo modo. I programmi scritti in Python sono in genere molto più brevi dei programmi equivalenti C C + + o Java, per diverse ragioni fra cui:

- I tipi di dati di alto livello consentono di esprimere operazioni complesse anche con una singola istruzione;
- Le istruzioni vengono raggruppate tramite indentazione anziché usare parentesi iniziale e finale o blocchi tipo INIZIO BLOCCO – FINE BLOCCO;
- Non è necessario dichiarare alcuna variabile.

Nota:
Quest'ultima caratteristica, farà storcere il naso ad alcuni programmatori specialmente di provenienza C C++ e Java. Ma come vedremo più avanti, la richiesta di scrittura di codice è talmente minore rispetto ad altri linguaggi per esempio da minimizzare la questione. Inoltre essendo fortemente dinamico, usa un' approccio totalmente diverso da altri linguaggi (argomento complesso che sarà trattato più avanti)

Python è estendibile: se sapete programmare in C è facile aggiungere una nuova funzione o un modulo pre costruito per l'interprete, sia per eseguire operazioni critiche alla massima velocità, o per collegare programmi Python a librerie che possono essere disponibili solo in forma binaria (ad esempio librerie grafiche proprietarie o di sistema). Una volta che siete diventati davvero pratici, è possibile collegare l'interprete Python in un'applicazione scritta in C e usarlo come linguaggio di estensione o di comando per tale applicazione.

Molti si domanderanno perché si chiami Python! Non ha nulla a che vedere con i serpenti, il suo autore Guido Van Rossum è un'appassionato della serie televisiva di sketch “Monty Python". Fare riferimenti alle comiche dei Monty Python nella documentazione è permesso, non solo ma è incoraggiato! (l'uso della parola spam ad esempio è veramente ossessiva (sarà perché la carne in scatola non mi fa impazzire, ma tant'è...)

Ora che sarete senz'altro entusiasti di Python, vi consigliamo di esaminare in maggior dettaglio il linguaggio e il suo ambiente. Dal momento che il modo migliore per imparare una linguaggio è quello di usarlo, questa guida vi invita a giocare con l'interprete Python ed ad eseguirne gli esempi.

Nel prossimo capitolo, verranno spiegati i meccanismi per utilizzare l'interprete. Si tratta di informazioni piuttosto banali, ma essenziali per provare gli esempi mostrati successivamente. Il resto della guida, introduce varie caratteristiche di Python attraverso esempi iniziando con semplici espressioni, istruzioni e tipi di dati, attraverso funzioni e moduli, ed infine, toccando concetti avanzati come le eccezioni e le classi definite dall'utente ecc.

2. Usare l'interprete di Python.

2.1. Invocare l'interprete.

Su sistemi UNIX/LINUX:

L'interprete Python su una macchina Unix/Linux, è generalmente installato in `/usr/local/bin/python3.x` (la x specifica il particolare numero di versione) su quelle macchine dove è disponibile; mettere `/usr/local/bin` nel percorso di ricerca della shell Unix/Linux rende possibile avviarlo digitando il comando:

```
python3
```

Poiché la scelta della directory in cui risiede l'interprete è un'opzione di installazione, è possibile installare Python in altre posizioni che potete decidere da soli o con l'amministratore del sistema. (Ad esempio, `/usr/local/python` è una scelta piuttosto popolare.)

su sistemi WINDOWS:

Su macchine Windows, l'installazione di Python è di solito collocata in `C:\Python3x`, anche se è possibile modificare la posizione in fase di installazione. Per aggiungere questa directory al file di path, è possibile digitare il seguente comando nel prompt dei comandi in un terminale DOS:

```
set path=%path%;C:\python3x
```

Digitando un carattere di end-of-file (`Control-D` su Unix/Linux, `Control-Z` su Windows) al prompt primario fa sì che l'interprete esca con uno stato di uscita uguale zero. Se questo non funziona, si può uscire l'interprete digitando `quit()`.

E' possibile inoltre far convivere quante versioni di Python si desidera senza che entrino in conflitto far di loro.

Gli editor di linea per l'interprete di solito non hanno funzionalità molto sofisticate. Su Unix/Linux, chiunque abbia installato l'interprete può avere abilitato il supporto per la libreria GNU readline, che aggiunge più elaborate funzioni di editing e una cronologia interattiva. Forse il controllo più veloce per vedere se la modifica della riga di comando che si sta scrivendo è supportata è Control-P quando si ottiene il prompt di Python. Se si ottiene un beep, allora si ha l'editing della riga di comando, fare riferimento all'appendice *Interactive Input Editing and History Substitution* per un'introduzione sull'uso dei tasti. Se ciò non sembra funzionare o se viene visualizzato `^P` allora l'editing della riga di comando non è disponibile, sarete solo in grado di utilizzare backspace per rimuovere i caratteri dalla riga corrente.

L'interprete di Python, opera all'incirca come una shell Unix/Linux, quando viene chiamato con lo standard input connesso a un dispositivo terminale (la tastiera) legge ed esegue i comandi in modo interattivo, se Python viene chiamato e come argomento, viene passato un file di script, allora Python si avvia eseguendo lo script del file passato.

Un secondo modo di lanciare l'interprete è `python -c command [arg] ...`, che esegue l'istruzione/i al comando, analogo a *-C* dell'opzione della shell. Poiché istruzioni Python contengono spesso spazi o altri caratteri che sono speciali per la shell, di solito è consigliato l'uso preventivo di apici

al cui interno è racchiuso il comando nella sua interezza. Alcuni moduli Python sono utili anche come script. Questi possono essere richiamati utilizzando `python -m module [arg] ...`, che esegue il file di origine per il modulo, come se si fosse scritto il suo nome completo sulla riga di comando. Quando si utilizza un file di script, a volte è utile essere in grado di eseguire lo script ed entrare in modalità interattiva successivamente. Questo può essere fatto passando *-I* prima dello script.

2.1.1. Passaggio di argomenti.

Una volta attivato l'interprete con il nome dello script e gli eventuali argomenti aggiuntivi, essi vengono trasformati in una stringa il cui interno contiene i parametri passati separati da degli spazi. Questa stringa così composta, viene passata alla variabile `argv gestita dal` nel modulo `sys`. Quindi per poter utilizzare i parametri eventualmente passati, è necessario importare il suddetto modulo digitando `import sys`. Per poter processare gli argomenti passati, la lunghezza della lista deve contenere almeno un' argomento, se non contiene argomenti, significa che la stringa è vuota e `sys.argv[0]`. L'argomento pur non essendo difficile, richiede una spiegazione ampia, per cui si rimanda alle varie guide presenti in rete.

2.1.2. Modo Interattivo.

Quando l'interprete legge i comandi da un terminale (**tty**) tipicamente la tastiera, si dice che è in modalità interattiva. In questa modalità viene richiesto per il comando successivo un prompt primario, di solito tre segni di maggiore (**>>>**), per le righe che richiedono una continuazione di comandi, viene attivato un prompt secondario costituito da tre punti (**...**). All'avvio l'interprete stampa un messaggio di benvenuto che indica il numero di versione e un avviso di copyright prima di attivare il prompt come indicato sotto.

```
$ python3
Python 3.4.0 (default, Jan 13 2014, 23:33:01)
[GCC 4.4.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Le linee di continuazione, come già detto sono composte da tre punti ad esempio al seguito di un'istruzione `if`

```
>>> il_mondo_e_piatto = 1
>>> if il_mondo_e_piatto:
...     print("Attento a non cadere giù!")
...
Attento a non cadere giù!
>>>
```

2.2. L'interprete e il suo ambiente.

2.2.1. Trattamento degli errori.

Quando si verifica un errore, l'interprete stampa un messaggio di errore e una traccia dello stack. In

modalità interattiva ritorna al prompt primario. Quando invece l'input è venuto da un file, esso esce con uno stato diverso da zero dopo aver stampato la traccia dello stack. (Eccezioni gestite da una clausola `except` in un'istruzione `try` non sono errori in questo contesto.) Alcuni errori sono incondizionatamente fatali e provocano un'uscita forzata con un valore diverso da zero, questo vale per le incoerenze interne e alcuni casi tipo esaurimento della memoria disco ecc. Tutti i messaggi di errore vengono scritti nel flusso di errore standard-output tipicamente il monitor o dove specificato. Digitando il carattere di interruzione (di solito **Ctrl-C** o **CANC**) al prompt primario o secondario si cancella l'input e ritorna al prompt primario (>>>).

[2] Provocare un'interruzione forzata mentre un comando è in esecuzione solleva l'eccezione `KeyboardInterrupt`, che può essere gestita tramite un'istruzione `try`.

2.2.2. Scripts Python Eseguibili.

Sui sistemi Unix/Linux in stile BSD, gli script Python possono essere direttamente eseguibili, come script di shell inserendo la riga:

```
#!/usr/bin/env python3.x
```

(supponendo che l'interprete si trovi nel PATH dell'utente) all'inizio dello script e dando al file il permesso di esecuzione. I caratteri **`#!`** devono essere i primi due caratteri del file. Su alcune piattaforme, questa prima riga deve terminare con una linea finale in stile Unix ('**`\n`**' *cioè il terminatore di linea significante 'nuova linea*'), non tipo Windows ('**`\r\n`**' *cioè il terminatore di linea significante ' ritorno del carrello più nuova linea*'). Va notato che il carattere **`'#'`** o hash, o carattere cancelletto, viene utilizzato anche per descrivere un commento in Python. Uno script per poter essere eseguito, deve avere un permesso come tale. E possibile renderlo eseguibile utilizzando il comando **chmod**:

```
$ chmod +x myscript.py
```

Sui sistemi Windows, non vi è alcuna nozione di "***modo eseguibile***". Il programma di installazione di Python associa automaticamente l'estensione **.Py** con python.exe in modo che un doppio clic su un file Python verrà eseguito come uno script. L'estensione può anche essere. **PYW**, in questo caso, la finestra di console che appare normalmente viene soppressa.

2.2.3. Codifica del codice sorgente.

Per impostazione predefinita, i file sorgenti Python vengono trattati come codificati in **UTF-8**. Con tale codifica, è possibile rappresentare la maggior parte delle lingue del mondo. Essi possono essere utilizzati contemporaneamente nelle stringhe letterali come identificatori e commenti. Si consiglia comunque di utilizzare solo caratteri **ASCII** per gli identificatori, ed è buona norma utilizzare solo questo tipo di caratteri se vogliamo che il nostro codice sia veramente portabile e di utilizzare **UTF-8** solo per stringhe di testo per i messaggi nella specifica lingua. Comunque per visualizzare correttamente tutti i caratteri, il vostro editor deve riconoscere che il file è **UTF-8**, e deve utilizzare un **font** che supporta tutti i caratteri utilizzati dal file. È anche possibile specificare una codifica differente per i file sorgente. Per fare questo, mettere ancora una riga di commento speciale subito dopo la linea **#!** per definire la codifica del file di

origine:

```
# -*- coding: encoding -*-
```

Con questa dichiarazione, tutto nel file sorgente verrà considerata come avente la codifica ***encoding*** invece di **UTF-8**. L'elenco dei possibili codifiche può essere trovata nella Python Library Reference, nella sezione `codecs`. Ad esempio, se il vostro editor preferito non supporta file codificati **UTF-8** e insiste sull'utilizzo di qualche altra codifica, è possibile per esempio scrivere Windows-1252, per ottenere il risultato:

```
# -*- coding: cp-1252 -*-
```

Ciò utilizzerà tutti i caratteri di caratteri nei file sorgente in Windows-1252. **Il commento speciale codifica deve essere nella prima riga se è da solo o nella seconda linea se presente quella di path all'interno del file.**

2.2.4. Il File Di Avvio In Modalità Interattiva.

Quando si usa Python interattivamente, è spesso comodo che alcuni comandi standard vengano eseguiti ogni volta che l'interprete viene avviato. È possibile farlo impostando una variabile di ambiente denominata `PYTHONSTARTUP` essa può contenere il nome di un file contenente i comandi di start-up. Questo è simile alla funzione `.profile` delle shell di Unix/Linux.

Questo file viene letto solo nelle sessioni interattive, non quando Python legge comandi da uno script, e non quando `/dev/tty` è impostata come sorgente dei comandi (altrimenti si comporterebbe come una sessione interattiva). Esso viene eseguito nello stesso spazio dei nomi in cui vengono eseguiti i comandi interattivi, cosicché gli oggetti che definisce o importa possono essere utilizzati senza riserve nella sessione interattiva. È inoltre possibile modificare in questo filele istruzioni `sys.ps1` e `sys.ps2` che permettono di cambiare i prompt di Python.

Se volete leggere un file di avvio addizionale dalla directory corrente, è possibile programmare questo nel file globale di avvio utilizzando il codice tipo `os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. Se si desidera utilizzare il file di avvio in uno script, deve essere necessariamente nello script:

```
import os
filename = os.environ.get('PYTHONSTARTUP')
if filename and os.path.isfile(filename):
    exec(open(filename).read())
```

Queste impostazioni possono essere utili ma non è necessario che le approfondiate adesso.

2.2.5. La Personalizzazione Dei Moduli.

Python fornisce due modi che consentono di personalizzarli: `sitecustomize` e `usercustomize`. Per vedere come funziona, è necessario prima di trovare il percorso della directory **site-packages** utente.

Avviare Python ed eseguire questo codice:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python3.4/site-packages'
```

Ora è possibile creare un file denominato `usercustomize.py` in quella directory e mettere tutto quello che volete in essa. Essa interesserà ogni invocazione di Python, a meno che non venga avviato con l'opzione -s per disattivare l'importazione automatica. `sitecustomize` funziona allo stesso modo, ma viene in genere creato da un amministratore del computer nella directory globale **site-packages**, `usercustomize` se esiste, e viene importata per prima. Vedere la documentazione del modulo `site` per maggiori dettagli.

1] Su Unix/Linux, l'interprete Python 3.x, non è installato di default ma viene installata ad oggi la versione 2.x in quanto numerosi programmi e script la utilizzano. Inoltre digitando semplicemente `python`, si avvia la versione 2.x. Per far avviare la versione 3.x bisogne digitare se installato `python3`. Questo a causa di alcune incompatibilità dovute ai miglioramenti apportati alla versione 3.x rispetto alla 2,x.

[2] Un problema con il pacchetto GNU Readline può impedire questo.

3. Un' Informale Introduzione A Python.

Nei seguenti esempi, **input** e **output** si distinguono per la presenza o l'assenza di richieste (**>>>** e **...**): per ripetere l'esempio, è necessario digitare tutto dopo il prompt, quando esso appare. Righe che non iniziano con un prompt sono output dell'interprete. Si noti che un prompt secondario su una riga in un esempio significa che è necessario digitare una riga vuota, questo è usato per terminare un comando multi-linea.

Molti degli esempi in questa guida, anche quelli immessi al prompt interattivo, includono commenti. In Python i commenti iniziano con il carattere cancelletto, # e si estendono fino alla fine fisica della linea. Un commento può comparire all'inizio di una riga o in seguito al codice, ma non all'interno di una stringa letterale. Un carattere hash (#)all'interno di una stringa letterale è solo un carattere qualsiasi e viene rappresentato così com'è cioè privo di valore. Dal momento che i commenti servono per chiarire il codice e non sono interpretati da Python, possono essere omessi durante la digitazione negli esempi.

Qualche esempio:

```
# Questo è il primo commento.
spam = 1  # e questo il secondo
          # ... e ora il terzo
text = "# Questo non è un commento perché inserito in una stringa."
```

3.1. Usare Python Come Un Calcolatore.

Proviamo con qualche semplice comando Python. Avviare l'interprete e attendere il prompt primario, **>>>**. (Non dovrebbe impiegarci troppo tempo)

3.1.1. Numeri.

L'interprete si comporta come una semplice calcolatrice: si può digitare un'espressione ed esso fornirà il risultato. La sintassi delle espressioni è semplice: gli operatori +, -, * e / funzionano come nella maggior parte degli altri linguaggi (ad esempio, Pascal o C); parentesi anche nidificate (()) possono essere utilizzati per il raggruppamento e la risoluzione. Per esempio:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5*6) / 4
5.0
>>> 8 / 5  # la divisione restituisce sempre un numero in virgola
mobile (diversamente da Python 2.x)
1.6
```

I numeri interi (es. `2`, `4`, `20`) sono di tipo `int`, quelli con una parte frazionaria (es. `5.0`, `1.6`) sono di tipo `float`. Vedremo di più su tipi numerici più avanti nella guida.

Divisione (`/`) restituisce sempre un `float`. Per ottenere un risultato intero (scartando qualsiasi risultato frazionario) è possibile utilizzare l'operatore `//` vedere *floor division*, per calcolare il resto è possibile utilizzare `%`:

```
>>> 17/3   #classica divisione restituisce un numero in virgola
mobile
5.666666666666667
>>>
>>> 17//3  # divisione che scarta la parte decimale
5
>>> 17%3 # l'operatore % invece restituisce la rimanenza della
divisione
2
>>> 5 * 3 + 2
17
```

Con Python è possibile usare l'operatore **** per calcolare la potenza di un numero [1]:

```
>>> 5 ** 2  # 5 il quadrato
25
>>> 2 ** 7  # 2 elevato alla potenza di 7
128
```

Il segno di uguale (=) viene utilizzato per assegnare un valore a una variabile e non come operatore di confronto (==). Successivamente, nessun risultato viene visualizzato prima del successivo prompt interattivo:

```
>>> altezza = 20
>>> larghezza = 5 * 9
>>> altezza * larghezza
900
```

Se si cercherà di utilizzare una variabile a cui non si sia precedentemente assegnato un valore, si genererà un'errore.

```
>>> n  # variabile senza assegnazione
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Python ha un potente supporto per i dati in virgola mobile e operatori di tipo con operandi misti, convertono l' operando intero a virgola mobile:

```
>>> 3 * 3.75 / 1.5
7.5
>>> 7.0 / 2
3.5
```

In modo interattivo, l'ultima espressione stampata viene assegnata alla variabile `_`. Ciò significa che quando si utilizza Python come una calcolatrice da tavolo, è un po 'più facile continuare calcoli, ad esempio:

```
>>> tassa = 12.5 / 100
>>> prezzo = 100.50
>>> prezzo * tassa
12.5625
>>> prezzo + _
113.0625
>>> round(_, 2)
113.06
```

Questa variabile deve essere considerata come a sola lettura da parte dell'utente. Non assegnare esplicitamente un valore ad essa per utilizzare il suo contenuto, bisogna creare una variabile locale indipendente per mascherare la variabile precostituita con il suo comportamento magico.

Oltre a `int` e `float`, Python supporta altri tipi di numeri, come `Decimal` e `Fraction`. Python ha anche il supporto integrato per i numeri complessi *complex numbers*, e utilizza il suffisso `j` o `J` per indicare la parte immaginaria (es. `3+5j`).

3.1.2. Le Stringhe.

Oltre ai numeri, Python può anche manipolare stringhe, che possono essere espresse in vari modi. Esse possono essere racchiuse tra singoli apici (`'...'`) o doppi apici (`"..."`) con lo stesso risultato [2]. il

carattere '\' Può essere usato per rappresentare le virgolette all'interno di una stringa:

```
>>> 'spam uova'            # virgolette singole
'spam uova'
>>> 'all\'aria aperta'     # \ permettere la stampa dell'apice.
"all'aria aperta"
>>> "all'aria aperta"      #...oppure utilizzando le virgolette doppie
"all'aria aperta"
>>> '"Si," ha detto si.'
'"Si," ha detto si.'
>>> "\"Si,\" ha detto si."
'"Si," ha detto si.'
```

Nel interprete interattivo, la stringa di output è racchiusa tra virgolette e caratteri speciali sono stampati usando la barra rovesciata o backslashes. Anche se questo potrebbe talvolta portare un' aspetto diverso per l' input (le virgolette che racchiudono potrebbero cambiare), le due stringhe sono equivalenti. La stringa è racchiusa tra virgolette doppie se la stringa contiene una singola senza virgolette, altrimenti viene racchiuso tra virgolette singole. La funzione `print()` produce un output più leggibile, omettendo le virgolette che racchiudono la stringa evitando i caratteri speciali:

```
>>> print('"Si," ha detto si.')
"Si," ha detto si.
>>> s = 'linea uno.\nlinea due.'  # \n significa nuova linea
>>> s                             # senza print(), \n viene stampata
'linea uno.\nlinea due.'
>>> print(s)                      # qui produce una nuova linea
linea uno.
linea due.
```

Se non si desidera che i caratteri preceduti da \ siano interpretati come caratteri speciali, è possibile utilizzare il *raw strings* aggiungendo una **r** prima delle virgolette con questo sistema, l'interprete non valuterà nessuna elaborazione ma semplicemente presenta il tutto così com'è:

```
>>> print('C:\some\nome')      # qui \n significa una nuova linea!
C:\some
ome
>>> print(r'C:\some\nome')     # notare la r prima della virgoletta
C:\some\nome
```

I letterali stringa possono estendersi su più righe. Un modo per farlo, è usare le triple virgolette:`"""..."""` . I fine linea vengono inclusi automaticamente nella stringa, ma è possibile impedire questo comportamento aggiungendo un **** alla fine della prima riga. In questo esempio, viene prodotto il seguente output :

```
print("""\
```

```
Usare: Queste [OPZIONI]       Visualizza queste righe
     -h                        ...
     -H hostname               ...
""")
```

Le stringhe possono essere concatenate (incollate assieme sequenzialmente) con l'operatore +, e ripetuto con *:

```
>>> # 3 volte 'Pippo ', seguito da ' si!'
>>> 3 * 'Pippo ' + ' si!'
'Pippo Pippo Pippo si!'
```

Due o più stringhe letterali (cioè quelle racchiuse tra virgolette) l'una accanto all'altra vengono automaticamente concatenate.

```
>>> 'Py' 'thon'
'Python'
```

Questo funziona solo con i letterali ma non con le variabili o espressioni:

```
>>> prefix = 'Py'
>>> prefix 'thon'  #non è possibile concatenare una variabile con una
stringa.
  ...
SyntaxError: invalid syntax

>>> ('un' * 3) 'ium'
  ...
SyntaxError: invalid syntax
```

Se lo scopo è quello di concatenare due variabili, allora dovete usare il segno +:

```
>>> prefix = 'thon'
'Python'
```

Questa caratteristica e spesso usata quando si cerca di spezzare stringhe troppo lunghe:

```
>>> text = ('Mettere più stringhe all'interno di'
            'parentesi per unirle.')
>>> text
'Mettere più stringhe all'interno di parentesi per unirle.'
```

Le stringhe possono essere indicizzate per cui il primo carattere ha come indice lo 0. Non vi è alcuna distinzione fra caratteri! Semplicemente ognuno rappresenta se stesso:

```
>>> word = 'Python'
>>> word[0]  # carattere alla posizione 0
'P'
>>> word[5]  # carattere alla posizione 5
'n'
```

Gli indici possono essere anche numeri negativi, in questo caso il conteggio inizia da destra:

```
>>> word[-1]  # l'ultimo carattere
'n'
>>> word[-2]  # penultimo carattere
'o'
>>> word[-6]  # il primo carattere
'P'
```

Si noti che poiché -0 è lo stesso di 0, gli indici negativi partono da -1.

Oltre all'indicizzazione, è supportato anche la divisione o slicing (letteralmente 'affettamento'). Mentre l'indicizzazione viene utilizzata per ottenere singoli caratteri, la divisione viene utilizzata per ottenere delle sotto stringhe:

```
>>> word[0:2]  # caratteri dalla posizione 0 alla 2 incluse
'Py'
>>> word[2:5]  # caratteri dalla posizione 0 inclusa a 5 esclusa
'tho'
```

Si noti come l'inizio è sempre incluso, e la fine sempre esclusa. Questo fa in modo che **s [: i] + s [i:]** è sempre uguale a **s**:

```
>>> word[:2] + word[2:]
'Python'
>>> word[:4] + word[4:]
'Python'
```

Gli indici delle sotto stringhe, hanno valori predefiniti utili; se omesso il primo indice è uguale a zero, così se omesso il secondo, viene definito dalla lunghezza della sotto stringa.

```
>>> word[:2]  # carattere dall'inizio alla posizione 2
'Py'
>>> word[4:]  # carattere dalla posizione 4 alla fine
'on'
>>> word[-2:] # carattere dalla penultima posizione alla fine
'on'
```

Indici fuori intervallo, sono trattati ignorandoli evitando così errori inutili:

```
>>> word[4:42]
```

```
'on'
>>> word[42:]
''
```

Le stringhe in Python non possono essere modificate, sono di tipo *immutable* cioè immutabili. Pertanto, l'assegnamento ad un indice di una stringa costituisce un errore:

```
>>> word[0] = 'J'
  ...
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
  ...
TypeError: 'str' object does not support item assignment
```

Se la modifica di una stringa, si rende necessaria, dovete crearne una nuova:

```
>>> 'J' + word[1:]
'Jython'
>>> word[:2] + 'py'
'Pypy'
```

La funzione precostituita `len()` restituisce la lunghezza della stringa passatagli:

```
>>> s = 'supercalifragilistiespiralitoso'
>>> len(s)
31
```

Vedere anche:

Text Sequence Type — str
Esempi di sequenza di tipi sulle stringhe, supportano le comuni operazioni supportate da tali tipi.
String Methods
Stringhe supportano un gran numero di metodi per le trasformazioni di base e la ricerca.
String Formatting
Informazioni sulla formattazione delle stringhe con `str.format()` .
printf-style String Formatting
Le vecchie operazioni di formattazione invocate con le stringhe Unicode utilizzano l'operando % alla sua sinistra. Sono qui descritte in maggiore dettaglio.

3.1.3. Le Liste.

Python possiede un certo numero di tipi di dati composti molto potenti, usati per raggruppare altri valori fra cui Il più versatile è la lista, che può essere scritta come un elenco di valori separati da virgole (articoli) tra parentesi quadre. Le liste possono contenere elementi di tipo diverso, ma di solito gli elementi sono tutti lo stesso tipo.

```
>>> quadrati = [1, 2, 4, 9, 16, 25]
>>> quadrati
[1, 2, 4, 9, 16, 25]
```

Come per le stringhe (e tutti gli altri tipi di sequenze precostituite), le liste possono essere indicizzate e suddivise:

```
>>> quadrati[0]    # restituisce il primo elemento
1
>>> quadrati[-1]
25
>>> quadrati[-3:]  # suddivide gli elementi in una nuova lista
[9, 16, 25]
```

Tutte le operazioni di suddivisione, restituiscono un nuovo elenco contenente gli elementi richiesti. Ciò significa che la seguente suddivisione restituisce una nuova (superficiale) copia della lista:

```
>>> quadrati[:]
[1, 2, 4, 9, 16, 25]
```

Le liste supportano anche le operazioni di concatenamento:

```
>>> quadrati + [36, 49, 64, 81, 100]
[1, 2, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Diversamente dalle stringhe, che sono immutabili, liste sono un tipo **mutabile**, cioè è possibile modificare il loro contenuto:

```
>>> cubi = [1, 8, 27, 65, 125]  # qui c'è qualcosa che non va
>>> 4 ** 3                      # il cubo di 4 è 64, non 65!
64
>>> cubi[3] = 64                # sostituisce il valore errato
>>> cubi
[1, 8, 27, 64, 125]
```

È inoltre possibile aggiungere nuovi elementi alla fine della lista, utilizzando il metodo `append()` (vedremo di più sui metodi in seguito):

```
>>> cubi.append(216)     # aggiunge il cubo di 6
>>> cubi.append(7 ** 3)  # e il cubo di 7
>>> cubi
[1, 8, 27, 64, 125, 216, 343]
```

E' anche possibile l'assegnamento a sotto liste. Questo può cambiare la dimensione della lista o cancellarla interamente:

```
>>> lettere = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> lettere
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # sostituisce alcuni valori
>>> lettere[2:5] = ['C', 'D', 'E']
>>> lettere
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # ora ne rimuove alcuni
>>> lettere[2:5] = []
>>> lettere
['a', 'b', 'f', 'g']
>>> # cancella la lista sostituendola con una vuota
>>> lettere[:] = []
>>> lettere
[]
```

La funzione precostituita `len()` applicata anche alle liste:

```
>>> lettere = ['a', 'b', 'c', 'd']
>>> len(lettere)
4
```

E' inoltre possibile nidificarle creare cioè delle liste al cui interno sono presenti altre liste e non solo per esempio:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

3.2. Primi Passi Di Programmazione.

Naturalmente, possiamo usare Python per compiti più complessi piuttosto che aggiungere due più due. Ad esempio, possiamo scrivere una sub-sequenza iniziale della serie di Fibonacci come segue:

```
>>> # serie di Fibonacci:
... # la somma di due elementi, definisce l'elemento successivo
```

```
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Quest'esempio, introduce nuove ed importanti caratteristiche.

- La prima riga utile contiene un assegnamento multiplo: le variabili **a** e **b**, ottengono contemporaneamente i valori **0** e **1**. Lo stesso dicasi per l'ultima riga, a dimostrazione che le espressioni sul lato destro vengono tutte valutate prima di un' assegnazione. Le espressioni laterali vengono valutate da sinistra a destra.

- Il ciclo `while` viene eseguito fino a quando la condizione (qui: `b < 10`) rimane vera. In Python, come in C, qualsiasi valore intero non-zero è vero, nulla è falso. La condizione può anche essere un valore stringa o una lista, di fatto qualsiasi sequenza, qualsiasi cosa con una lunghezza diversa da zero è vero, le sequenze vuote sono false. Il test utilizzato nell'esempio è un semplice confronto. Gli operatori standard di confronto sono scritti come in C: **<** (minore di), **>** (maggiore di), **==** (uguale a), **<=** (minore o uguale a), **>=** (maggiore o uguale a) e **!=** (non uguale a).

- Il corpo del ciclo è indentato: l'identazione è il modo con cui Python raggruppa le istruzioni. A differenza di altri linguaggi, Python non utilizza parentesi graffe **{}** come in C o marcatori di blocco come in Pascal. Ma usa gli spazi. Al prompt interattivo, è necessario digitare una tabulazione o uno spazio per ogni linea identata cioè costituita da 2 so più spazi. In pratica se non si possiede un editor di testo con autoidentazione, la possibilità di errori aumenta in quanto è il solo modo con cui Python riconosce le diverse righe di codice. La cosa positiva è che non richiede caratteri terminatori come negli altri linguaggi tipo la virgola il punto e virgola, il ritorno del carrello ecc. Inoltre costringe il programmatore ad uniformarsi a quello stile piacente o non piacente a beneficio di chiunque legga il programma. Quando viene immessa in modo interattivo un'istruzione composta, deve essere seguita da una riga vuota per indicare il suo completamento (dato che il parser (l'analizzatore di linee di programma) non può sapere quando si è digitato l'ultima riga). Si noti che ogni riga all'interno di un blocco di base deve essere rientrata nello stesso modo.

- La funzione `print()` scrive il valore dell'argomento/i dati. Si differenzia dal semplice scrivere l'espressione che volete (come abbiamo fatto in precedenza negli esempi calcolatrice) nel modo in cui gestisce più argomenti, numeri e stringhe. Le stringhe vengono stampate senza virgolette, e viene inserito uno spazio tra gli elementi, in modo da presentare gli elementi in modo ordinato, come questo esempio:

```
>>> i = 256*256
>>> print('Il valore di i é', i)
Il valore di i é 65536
```

La parola chiave ***end*** può essere usata per evitare la nuova riga dopo l'output, o terminare l'output con una stringa differente:

```
>>> a, b = 0, 1
>>> while b < 1000:
...     print(b, end=',')
...     a, b = b, a+b
...
1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

Note:

[1] Poiché `**` ha precedenza maggiore rispetto a `-3**2` saranno interpretati come `(3**2)` e quindi produrre il risultato di `-9`. Per evitare questo ed ottenere `9`, è possibile usare `(-3)**2`.

[2] A differenza di altri linguaggi, caratteri speciali come `\n` hanno lo stesso significato sia con singoli (`'...'`) e doppi (`"..."`) apici. L'unica differenza tra i due è che tra singoli apici non c'è bisogno di usare il carattere di escape `\` e viceversa.

4. Strumenti Per Controllare il Flusso.

Oltre all'istruzione `while` appena introdotta, Python riconosce le solite istruzioni di controllo del flusso che possiedono anche altri linguaggi, con alcune migliorie.

4.1. L'Istruzione `if`.

E' forse il tipo di istruzione più conosciuta. Per esempio:

```
>>> x = int(input("immetti un intero: "))
immetti un intero: 42
>>> if x < 0:
...     x = 0
...     print('Negativo cambiato in zero')
... elif x == 0:
...     print('Zero')
... elif x == 1:
...     print('Singolo')
... else:
...     print('Altro')
...
Altro
```

Ci possono essere o nessuna o più `elif`, la parte `else` è opzionale. La parola chiave '`elif`' è la contrazione di 'else if', ed è utile essendo più corta per evitare un' eccessiva identazione. Una sequenza di ... `elif` ... `elif` ...`if` è un sostituto per l' istruzione `switch` o `case` che si trova in altri linguaggi.

4.2. L'Istruzione `for`.

L'istruzione `for` in Python differisce un po 'da quello che si può essere abituati in C o Pascal. Piuttosto che iterare sempre su una progressione aritmetica di numeri (come in Pascal), o dare all'utente la possibilità di definire sia il passo di iterazione e la condizione di arresto (come C), l'istruzione `for` di Python, itera sugli elementi di una sequenza tipo una lista o una stringa, nell'ordine in cui appaiono nella sequenza. Ad esempio:

```
>>> # Misura la lunghezza di alcune stringhe in una lista:
... parole = ['gatto', 'finestra', 'defenestrato']
>>> for w in parole:
...     print(w, len(w))
...
gatto 5
finestra 8
defenestrato 12
```

Notare la notevole semplicità e compattezza dell'istruzione. Se è necessario modificare la sequenza di iterazione all'interno del ciclo (ad esempio per duplicare gli oggetti selezionati), si consiglia di eseguire una copia. L'iterazione di una sequenza non implicita il fare una copia. La notazione **slice** lo rende particolarmente conveniente:

```
>>> for w in parole[:]:  # Cicla sulla lista e la ricopia sopra.
...     if len(w) > 8:
...         parole.insert(0, w)
...
>>> parole
['defenestrato', 'gatto', 'finestra', 'defenestrato']
```

4.3. La Funzione `range()`.

Se si necessita d'iterare su una sequenza di numeri, la funzione precostituita `range()`, viene in aiuto. Essa genera progressioni aritmetiche:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

Notare anche in questo caso, la compattezza del codice.

Il dato finale non è mai parte della sequenza generata; `range(10)` genera 10 valori, gli indici leciti per gli elementi di una sequenza di lunghezza 10. E' possibile partire da un altro numero, o specificare un incremento diverso (persino negativo, a volte questo è chiamato il ' passo '):

```
range(5, 10)
   5 verso 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

Per scorrere gli indici di una sequenza, è possibile combinare `range()` e `len()` come segue:

```
>>> a = ['Maria', 'ha', 'un', 'piccolo', 'agnello']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Maria
1 ha
2 un
3 piccolo
4 agnello
```

Nella maggior parte dei casi, tuttavia, è conveniente utilizzare la funzione `enumerate()`, vedere Tecniche dei cicli *Looping Techniques*.

Una cosa strana succede solo se si stampa un intervallo:

```
>>> print(range(10))
range(0, 10)
```

In molti casi l'oggetto restituito da `range()` si comporta come se fosse una lista, ma in realtà non lo è. E' un oggetto che restituisce gli elementi successivi nella sequenza desiderata quando si scorre su di esso, ma in realtà non fa realmente una lista, risparmiando così spazio in quanto non conservato in memoria.

Diciamo un tale oggetto è iterabile, cioè, adatto come obiettivo per le funzioni e costrutti che si aspettano qualcosa in cui si possono ottenere elementi successivi fino ad esaurimento dei dati. Abbiamo visto che l'istruzione `for` è un iteratore. La funzione `list()` ne è un' altro, crea liste di iterabili:

```
>>> list(range(5))
[0, 1, 2, 3, 4]
```

Più avanti vedremo altre funzioni che restituiscono iterabili e prendono iterabili come argomento.

4.4. L'Istruzione `break` e `continue` e La Clausola `else` Sui Cicli.

L'istruzione `break`, come in C, esce dal ciclo `for` o `while` che lo contiene. Istruzioni di ciclo possono avere una clausola `else`, che viene eseguita quando il ciclo termina per esaurimento della lista (con `for`) o quando la condizione diventa falsa (con `while`), ma non quando il ciclo è terminato da un'istruzione `break`. Ciò è esemplificato nel ciclo seguente, che ricerca numeri primi:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'uguale', x, '*', n//x)
...             break
...     else:
...         # il ciclo termina senza aver trovato un fattore
...         print(n, 'è un numero primo')
...
2 è un numero primo
3 è un numero primo
4 uguale 2 * 2
5 è un numero primo
6 uguale 2 * 3
7 è un numero primo
8 uguale 2 * 4
9 uguale 3 * 3
```

Sì, questo è il codice corretto guardate con attenzione... La clausola `else` appartiene al ciclo `for`, non l'istruzione `if`).
Quando viene utilizzato con un ciclo, la clausola `else` assomiglia più alla clausola `else` di un'istruzione `try` di quanto non faccia quella dell'istruzione `if`: La clausola else di un'istruzione `try` viene eseguita quando non si verifica alcuna eccezione, e la clausola `else` di un ciclo viene eseguita quando si verifica nessun `break` . Per maggiori informazioni su `try` e le eccezioni, vedere:
Handling Exceptions.

La dichiarazione `continue`, anch'essa presa in prestito dal C, continua con la prossima iterazione del ciclo:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Trovato un numero pari", num)
...         continue
...     print("Trovato un numero", num)
Trovato un numero numero pari 2
Trovato un numero 3
Trovato un numero numero pari 4
Trovato un numero 5
Trovato un numero numero pari 6
Trovato un numero 7
Trovato un numero numero pari 8
```

```
Trovato un numero 9
```

4.5. L'Istruzione `pass`.

L'istruzione `pass` non fa nulla. Può essere utilizzata quando è richiesta una dichiarazione sintatticamente corretta ma il programma non richiede alcuna azione. Per esempio:

```
>>> while True:
...     pass  #Entra in un ciclo infinito da interrompere con
(Ctrl+C)
...
```

Questa istruzione, viene usata tipicamente per creare classi minimali da gestire successivamente:

```
>>> class MiaClasseVuota:
...     pass
...
```

Un altro modo di utilizzare `pass`, è come segnaposto per una funzione o di un blocco condizionale quando si lavora su del nuovo codice, che consente di gestire il codice pensando ad un livello più astratto. L'istruzione `pass` viene ignorata senza generare alcun che:

```
>>> def log_iniziale(*args):
...     pass   # Ricordarsi d'implementare qualcosa qui!
...
```

4.6. Definizioni Di Funzioni.

Abbiamo visto precedentemente come implementare del codice per ottenere una sequenza di Fibonacci. Possiamo ora invece creare una funzione fino ad un limite arbitrario da poter utilizzare ovunque possa servire senza dover riscriverla:

```
>>> def fib(n):    # scrive una sequenza Fibonacci fino a n
...     """stampa una lista contenente la serie Fibonacci fino a n"""
...     a, b = 0, 1
...     while a < n:
...         print(a, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # Ora chiamiamo la nostra funzione:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

La parola chiave `def` introduce una definizione di funzione. Essa deve essere seguita dal nome della funzione e l'elenco tra parentesi dei parametri formali che se sono in numero maggiore di uno, devono essere separati da virgole.. Le dichiarazioni che formano il corpo della funzione iniziano alla riga successiva, e devono essere identate.

La prima istruzione del corpo della funzione può opzionalmente essere una stringa letterale, questa stringa è la stringa di documentazione della funzione o ***docstring***. (Ulteriori informazioni su ***docstring*** si trovano nella documentazione *Documentation Strings*). Python possiede degli strumenti per la creazione di documentazione che usano le ***docstring*** per produrre automaticamente documentazione in linea o stampata, o per permettere all'utente una navigazione interattiva attraverso il codice: è buona pratica includere ***docstring*** nel codice che si scrive, in modo da prendere l'abitudine ad usarla.

L'esecuzione di una funzione introduce ad una nuova classificazione delle variabili in questo caso locali alla funzione. Più precisamente, tutte le assegnazioni di variabili della funzione, vengono memorizzate in una tabella locale propria della funzione dei simboli locali; quando ci si riferisce ad una variabile, Python prima guarda dentro questa tabella, poi nelle tabelle dei simboli locali di funzioni che la racchiudono a sua volte (se esiste), poi nella tabella dei simboli globale, e infine nella tabella di nomi precostituiti. Pertanto, le variabili globali non possono essere assegnate direttamente ad un valore all'interno di una funzione (a meno che non denominato in una dichiarazione globale), anche se possono essere referenziate.

I parametri (argomenti) di una chiamata a una funzione vengono introdotti nella tabella dei simboli locale della funzione chiamata quando essa viene invocata, quindi, gli argomenti sono passati usando una chiamata per valore (dove il valore è sempre un riferimento a un oggetto, non il valore dell'oggetto). [1] Quando una funzione chiama un'altra funzione, viene creata una nuova tabella locale dei simboli per tale chiamata e via via così.

Una definizione di funzione introduce il nome della funzione nella tabella dei simboli corrente. Il valore del nome della funzione ha un tipo riconosciuto dall'interprete come funzione definita dall'utente. Questo valore può essere assegnato ad un altro nome che può quindi essere utilizzato come funzione. Questo serve come un meccanismo generale di ridenominazione:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Provenendo da altri linguaggi,, si potrebbe obiettare che `fib` non è una funzione, ma una procedura in quanto non restituisce un valore. Va ricordato che Python sinteticamente semplifica la stesura del codice eliminando molti degli orpelli che altri linguaggi hanno! In realtà, anche le funzioni senza istruzione `return` restituiscono un valore. Questo valore è chiamato `None` (è un nome predefinito). La scrittura del valore `None` è di norma soppressa dall'interprete se fosse l'unico valore scritto. Lo si può vedere se davvero si vuole usando `print()`:

```
>>> fib(0)
>>> print(fib(0))
None
```

E 'semplice scrivere una funzione che restituisce un elenco dei numeri della serie di Fibonacci, invece di stamparli:

```
>>> def fib2(n):    # scrive una sequenza Fibonacci fino a n
...     """stampa una lista contenente la serie Fibonacci fino a n"""
...     risultato = []
...     a, b = 0, 1
...     while a < n:
...         risultato.append(a)    # vedi sotto
...         a, b = b, a+b
...     return risultato
...
>>> f100 = fib2(100)    # la chiama
>>> f100                # scrive il risultato
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Questo esempio, come al solito, mette in luce alcune nuove funzionalità di Python:

- L'istruzione `return` restituisce un valore da una funzione. `return` senza argomenti espressione, restituisce `None`. Anche il fallimento di una funzione restituisce `None.`

- L'istruzione `risultato.append(a)` chiama un metodo dall'elenco dell'oggetto `risultato`. Un metodo è una funzione che 'appartiene' ad un oggetto ed è denominato `obj.NomeMetodo`, dove `obj` è un qualche oggetto (può essere un'espressione), e `NomeMetodo` è il nome di un metodo che viene definito dal tipo di oggetto. Diversi tipi definiscono metodi diversi. Metodi di tipi diversi possono avere lo stesso nome senza causare ambiguità. (E 'possibile definire i propri tipi e metodi degli oggetti, utilizzando le classi, vedere *Classes*) Il metodo `append()` mostrato nell'esempio è definito per gli oggetti lista, aggiunge un nuovo elemento alla fine della lista. In questo esempio è equivalente a `risultato = risultato + [a]`, ma più efficiente.

4.7. Ancora Sulla Definizione Di Funzioni.

E 'anche possibile definire funzioni con un numero variabile di argomenti. Ci sono tre forme, che possono essere combinate:

4.7.1. Valori Argomenti Predefiniti.

La forma più utile è specificare un valore predefinito per uno o più argomenti. Questo crea una funzione che può essere chiamata con un numero di argomenti definiti per interagire con essa. Per esempio:

```
def quesito(prompt, tentativi=4, conferma='Si o no, prego!'):
    while True:
        ok = input(prompt)
        if ok in ('S', 'Si', 's', 'si'):
            return True
        if ok in ('n', 'no', 'N', 'NO'):
            return False
```

```
        tentativi = tentativi - 1
        if tentativi < 0:
            raise IOError('refusenik user')
        print(conferma)
```

Questa funzione può essere richiamata in diversi modi:

- dando solo l'argomento obbligatorio: quesito('Vuoi davvero uscire?')
- dando uno degli argomenti opzionali: quesito('? OK per sovrascrivere il file', 2)
- o anche dando tutti gli argomenti: quesito('? OK per sovrascrivere il file', 2, 'Ok, solo sì o no')

Questo esempio introduce anche la parola chiave `in`. Questa verifica se una sequenza contiene un certo valore.
I valori predefiniti vengono valutati al momento della definizione nella funzione nell'ambito definito, in modo che:

```
i = 5

def f(arg=i):
    print(arg)

i = 6
f()
```

Stamperà 5.

Avvertenza importante: Il valore predefinito, viene valutato solo una volta. Questo fa differenza quando il valore predefinito è un oggetto mutabile come una lista, un dizionario o istanze nella maggior parte delle classi. Ad esempio, la seguente funzione accumula gli argomenti ad essa passati in chiamate successive:

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
Verrà stampato questo:
[1]
[1, 2]
[1, 2, 3]
```

Se non si desidera che il valore predefinito sia condiviso tra chiamate successive, è possibile scrivere la funzione in questo modo:

```
    if L is None:
        L = []
    L.append(a)
```

```
    return L
```

4.7.2. Argomenti Chiave.

Le funzioni possono essere chiamate anche usando *keyword arguments* cioè gli argomenti chiave nella forma `ArgChiave=valore`. Per esempio, la seguente funzione :

```
def motore(volts, stato='fermo', azione='gira', tipo='sincrono'):
     print("-- questo motore non farebbe", azione, end=' ')
     print("Se immetti", volts, "ai morsetti.")
     print("-- non molto usato il motore ",tipo)
     print("-- è", stato, "!")
```

accetta un argomento obbligatorio (`volts`) e tre argomenti opzionali (`stato`, `azione`, e `tipo`). Questa funzione può essere chiamata in uno dei seguenti modi:

```
motore(1000)                                # 1 argomento posizionale
motore(volts=1000)                          # 1 chiave con argomento
motore(volts=1000000, azione='brucia')      # 2 chiavi con argomento
motore(azione='brucia', volts=1000000)      # 2 chiavi con argomento
motore('un milione', 'morto', 'fermo')      # 3 argomenti posizionali
motore('mille', stato='fuma')               # 1 posizionale, 1 chiave
```

invece tutte queste chiamate non sarebbero valide:

```
motore()                                    # argomento richiesto omesso
motore(volts=75.0, 'morto')                 # argomento senza chiave
motore(110, volts=220)                      # valore argomento duplicato
motore(colore='giallo')                     # argomento chiave inesistente
```

In una chiamata alle funzioni, gli argomenti chiave devono seguire argomenti posizionali. Tutti gli argomenti chiave passati, devono corrispondere ad uno degli argomenti accettati dalla funzione e il loro ordine non è importante ad esempio, `colore` non è un argomento valido per la funzione `motore`. Questo include anche argomenti non opzionali. Nessun argomento può ricevere più di un valore alla volta. Ecco un esempio che non funziona a causa di questa restrizione:

```
>>> def function(a):
...     pass
...
>>> function(0, a=0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: function() got multiple values for keyword argument 'a'
```

Quando un parametro formale finale nella forma `**nome` è presente, esso riceve un dizionario (vedere *Mapping Types — dict*) contenente tutti gli argomenti-chiave, ad eccezione di quelle corrispondenti al parametro formale. Questo può essere combinato con un parametro nella forma `*nome` (descritto nella prossima sezione) che riceve una **tupla** contenente gli argomenti posizionali oltre la lista dei parametri

formali. (`*nome` deve venire prima di `**nome`) Ad esempio, se si definisce una funzione come questa:

```
def formaggi(tipo, *argomenti, **chiavi):
    print("-- Avete del ", tipo, "?")
    print("-- Mi dispiace, è finito il ", tipo)
    for arg in argomenti:
         print(arg)
    print("- * 40)
    chiavi = sorted(chiavi.keys())
    for Ch in keys:
        print(Ch,":",chiavi[Ch])
```

Si potrebbe definire così:

```
formaggi("Limburger", "è veramente cremoso, signore.",
        "è veramente, veramente cremoso, signore.",
        negoziante="Filippo",
        cliente="Arturo"
        scena="Negozio di formaggi")
```

che stamperebbe:

```
-- Avete del  Limburger ?
-- Mi dispiace, è finito il  Limburger
è veramente cremoso, signore.
è veramente, veramente cremoso, signore.
```

```
cliente : Arturo
negoziante : Filippo
scena : Negozio di formaggi
```

Si noti che l'elenco dei nomi degli argomenti-chiave, viene creato ordinando il risultato con il metodo `sorted()`. Se questo metodo non viene utilizzato,il suo contenuto, sarà restituito in modo indefinito.

4.7.3. Liste di argomenti arbitrari.

Infine, l'opzione utilizzata meno frequentemente è quella di specificare che una funzione può essere chiamata con un numero arbitrario di argomenti. Questi argomenti dovranno essere incapsulati in una **tupla** (vedi *Tuples and Sequences*). Prima del numero variabile di argomenti, possono esservi o nessuno o più argomenti normali.

```
def scrittura_argomenti_multipli(file, separatore, *args):
    file.write(separatore.join(args))
```

Normalmente, questo tipo di argomenti `variabili` saranno gli ultimi nella lista dei parametri formali, perché vanno in profondità nidificata su tutti gli argomenti di input rimanenti che vengono passati alla funzione. Tutti i parametri formali che si verificano dopo il parametro `*args` sono solo argomenti passati come parole-chiave , il che significa che possono essere usati solo come parole chiave piuttosto che

argomenti posizionali.

```
>>> def concatena(*args, sep="/"):
...     return sep.join(args)
...
>>> concatena("terra", "marte", "venere")
'terra/marte/venere'
>>> concatena("terra", "marte", "venere", sep=".")
'terra.marte.venere'
```

4.7.4. Scompattamento degli elenchi di argomenti.

La situazione inversa si verifica quando gli argomenti sono già in una lista o in una tupla, ma hanno bisogno di essere estratti per una chiamata di funzione che richiede argomenti posizionali separati. Per esempio, la funzione `range()` si aspetta start e stop come argomenti separati. Se non sono disponibili separatamente, scrivete la chiamata di funzione con l'operatore * per scompattare gli argomenti di una lista o di una tupla:

```
>>> list(range(3, 6))   # normale chiamata con argomenti separati
[3, 4, 5]
>>> args = [3, 6]
>>> list(range(*args))  # chiamata con argomenti scompattati da lista
[3, 4, 5]
```

Allo stesso modo, i dizionari sono in grado di fornire argomenti chiave con l'operatore **:

```
def motore(volts, stato='fermo', azione='morto', tipo='sincrono'):
...     print("-- Questo motore non farebbe", azione, end=' ')
...     print("se immetti", volts, "volts.", end=' ')
...     print("sarebbe", stato, "!")
...
>>> d = {"volts": "quattromila", "stato": "morto", "azione": "VOOM"}
>>> motore(**d)
-- Questo motore non farebbe VOOM se immetti quattromila volts.
Sarebbe morto!
```

4.7.5. Espressioni Lambda.

Delle piccole funzioni anonime possono essere create con la parola chiave `lambda`. Questa funzionalità, restituisce la somma dei suoi due argomenti: `lambda a, b: a+b`. Le funzioni lambda possono essere usate ovunque siano richiesti dalle funzioni sugli oggetti. Esse sono sintatticamente limitate ad una singola espressione. Semanticamente, sono solo zucchero sintattico per una definizione di funzione normale. Come le definizioni di funzioni annidate, le funzioni lambda possono riferirsi a variabili il cui campo è contenuto:

```
>>> def incrementatore(n):
...     return lambda x: x + n
```

```
...
>>> f = incrementatore(42)
>>> f(0)
42
>>> f(1)
43
```

C precedente utilizza un'espressione `lambda` per restituire una funzione. Un altro utilizzo è quello di passare una piccola funzione come argomento:

```
>>> coppia = [(1, 'uno'), (2, 'due'), (3, 'tre'), (4, 'quattro')]
>>> coppia.sort(key=lambda coppia: coppia[1])
>>> coppia
[(2, 'due'), (3, 'tre'), (4, 'quattro'), (1, 'uno')]
```

Nota del traduttore:
Questo argomento, non è trattato in modo esaustivo e/o chiaro nella guida originale. Per cui non essendo in grado di tradurla e spiegarla in maniera ampia, mi rimetto alla mera traduzione. Comunque in una eventuale prossima release di questa guida, mi riservo di spiegarlo in modo più chiaro aggiungendovi concetti ed esempi.

4.7.6. Stringhe di documentazione.

In questo paragrafo, tratteremo un' argomento già accennato all'inizio della guida e cioè la documentazione. Questa che può sembrare una cosa di scarso valore pratico (infatti è spesso trascurata dal programmatore), è invece una risorsa preziosa in quanto permette a chiunque di capire cosa è stato fatto, ma soprattutto permette a noi stessi di capire cosa abbiamo scritto magari solo alcuni mesi fa. A chi non è capitato di avere difficoltà a leggere il nostro stesso codice? Inoltre questa pratica, se ben fatta consente di avere praticamente una guida del programma da consultare. Passiamo ora a le convenzioni usate.

- La prima riga deve sempre essere una orientata alla finalità dell'oggetto. Per brevità, si intende che non deve indicare esplicitamente il nome o il tipo di oggetto, dal momento che queste informazioni sono disponibili con altri mezzi (tranne se il nome sembra essere un verbo che descrive il funzionamento di una funzione). Questa linea dovrebbe iniziare con una lettera maiuscola e terminare con un punto.
- Se vi sono più righe nella stringa documentazione, la seconda dovrebbe essere vuota, per separare visivamente il sommario dal resto della descrizione.
- Le seguenti righe devono essere uno o più paragrafi che descrivono le convenzioni di chiamata dell'oggetto, i suoi effetti collaterali, e quant'altro.

Il **parser** di Python cioè quello strumento facente parte dell'interprete che si occupa di capire cosa abbiamo scritto e se lo abbiamo scritto bene, non toglie le identazioni dalle stringhe letterali, in modo che gli strumenti che si occupano della documentazione possano o meno identare a seconda che lo si desideri o no. Questo viene reso possibile utilizzando la seguente convenzione:

- La prima riga non vuota dopo la prima riga della stringa determina la quantità di rientro per l'intera stringa di documentazione. (Non possiamo usare la prima linea dal momento che è

generalmente adiacente all'apertura delle virgolette della stringa per cui il suo rientro non sarebbe evidente.) Gli spazi "equivalenti" a tale indentazione vengono poi tolti dall'inizio di tutte le linee della stringa.

- Linee non identate correttamente, non dovrebbero esserci, ma se ciò si verificasse, tutti i loro spazi principali dovrebbero essere tolti.
- L'equivalenza degli spazi dovrebbe essere testata dopo l'applicazione delle tabulazioni (a 8 spazi, normalmente).

Ecco un' esempio di documentazione multi linea:

```
>>> def mia_funzione():
...     """Non fa nulla, ma è documentata.
...
...     No, non fa nulla davvero!
...     """
...     pass
...
>>> print(mia_funzione.__doc__)
Non fa nulla, ma è documentata

    No, non fa nulla davvero!
```

4.7.7. Annotazioni di Funzioni.

Le annotazioni sulle funzioni (*Function annotations*), sono cose per pignoli. I meta-dati definiti dall'utente sono completamente opzionali e di carattere arbitrario. Né Python al suo interno, né librerie esistenti usano per le annotazioni funzionali come uno standard per qualsiasi uso. Questa sezione ne mostra solo la sintassi, per cui progetti sviluppati di altre persone sono liberi di utilizzare annotazioni alle funzioni per la propria documentazione come meglio credono.

Le annotazioni di una funzione, sono memorizzate nel attributo `__annotations__` come un dizionario e non hanno nessun effetto operativo sulla funzione. I parametri di annotazione, sono definiti da due punti dopo il nome del parametro, seguito da una valutazione di espressione per il valore dell'annotazione. Esse sono definite da un letterale '`->`' seguita da un'espressione tra la lista dei parametri e i due punti che denota la fine dell'istruzione `def()`. L'esempio seguente ha un argomento posizionale, un argomento chiave, e il valore di ritorno annotato con cose senza senso:

```
>>> def f(prosciutto: 42, uova: int = 'spam') -> "Niente da vedere
qui":
...     print("Annotazioni:", f.__annotations__)
...     print("Argomenti:", prosciutto, uova)
...
>>> f('Stupendo')

Annotazioni: {'prosciutto': 42, 'return': 'Niente da vedere qui',
'uova': <class 'int'>}

Argomenti: Stupendo spam
```

4.8. Intermezzo: Stile di Codifica.

Ora che siamo pronti per la scrittura di codice serio e quindi più lungo, è giunto il momento di parlare di stili di codifica. La maggior parte dei linguaggi permettono una codifica del codice abbastanza arbitraria per cui il risultato è quello di una moltitudine di stili più o meno eleganti, più o meno leggibili ma comunque personalizzati. Questo produce diversi modi di codificare e addirittura diverse scuole di pensiero. Rendere più facile la lettura del codice agli altri e anche a se stessi è sempre una buona idea! Quindi l'adozione di un buon stile di codifica, è indispensabile in quanto aiuta moltissimo.

Per Python, dal documento **PEP 8** è emerso come sia importante aderire allo stile di guida nella maggior parte dei progetti al fine di promuove uno stile di codifica molto leggibile e piacevole per gli occhi. Ogni sviluppatore Python ad un certo punto deve leggere il suo o l'altrui codice. Qui di seguito ci sono gli stili più importanti estratti per voi:

- Usare un' identazione di 4 spazi senza l'uso del tabulatore.
- 4 spazi sono un buon compromesso tra la piccola identazione (2 spazi) che consente maggiore profondità di annidamento) e la grande identazione (8 spazi) più facile da leggere. La tabulazione, introduce solo confusione, è meglio non adoperarla.
- Incapsulare le linee in modo che non superino 79 caratteri.
  Questo aiuta gli utenti che hanno display di piccole dimensioni e viceversa consente di avere più file di codice a chi possiede schermi più grandi.
- Utilizzare righe vuote per separare le funzioni e le classi e grossi blocchi di codice all'interno di funzioni.
- Quando possibile, inserire commenti su una linea propria.
- Utilizzare docstrings.
- Usare spazi fra gli operatori e dopo le virgole, ma non direttamente all'interno di costrutti fra parentesi: `a = f(1,2) + g(3,4)`.
- Date un nome coerente alle vostre classi e funzioni, la convenzione è quella di utilizzare la cosi detta '`CamelCase`' per le classi e `lettere_minuscole_con_sottolineatura` per funzioni e metodi. Utilizzare sempre sé stesso come nome per il primo argomento del metodo (vedere *A First Look at Classes*).
- Non utilizzare codifiche fantasiose se il codice è pensato per essere utilizzato in ambienti internazionali. In maniera predefinita Python, usa UTF-8, ma la codifica ASCII va bene in ogni caso ed è più capibile.
- Per lo stesso motivo, non utilizzare caratteri non-ASCII in identificatori. Se chi dovrà leggere il codice appartiene ad un'altra lingua non avrà difficoltà a leggerlo e a modificarlo.

Note:

[1] In realtà, la chiamata per riferimento all'oggetto sarebbe una descrizione migliore, dato che se viene passato un oggetto mutabile, il chiamante vedrà le eventuali modifiche fatte ad esso (gli elementi inseriti in una lista).

5. Strutture di Dati.

Questo capitolo descrive alcune cose che sono già state affrontate precedentemente ma in modo più dettagliato, e aggiunge anche alcune nuove cose.

5.1. Approfondimento Sulle Liste.

Il tipo di dati **lista** ha alcuni altri metodi oltre a quelli già visti. Ecco tutti i metodi degli oggetti lista:

`list.append(x)`

Aggiunge un' elemento alla fine della lista. Equivale a: `a[len(a):] = [x]`.

`list.extend(L)`

Amplia la lista aggiungendo tutti gli elementi della lista data. Equivale a `a[len(a):] = L`.

`list.insert(i, x)`

Inserisce un oggetto in una data posizione. Il primo argomento è l'indice dell'elemento davanti al quale inserire, così che `a.insert(0,x)` inserisce nella parte anteriore della lista, e `a.insert(len(a),x)`. Equivale a `a.append(x)`.

`list.remove(x)`

Rimuovere il primo elemento della lista il cui elemento è x. Nel caso in cui vi siano più elementi identici, viene rimosso il primo. Si ha un'errore se non tale elemento non esiste..

`list.pop([i])`

Rimuovere la voce nella lista nella posizione indicata, e lo restituisce. Se non viene specificato alcun indice, `a.pop()` rimuove e restituisce l'ultimo elemento della lista. (**Le parentesi quadre intorno la i nell'esempio su fatto, indicano che il parametro è opzionale, non è che è necessario digitare le parentesi quadre in quella posizione**). Vedrete questa notazione frequentemente nella Python Library Reference).

`list.clear()`

Rimuove tutti gli elementi dalla lista. Equivale a `del a[:]`.

`list.index(x)`

Restituisce l'indice nella lista del primo elemento il cui valore è x. Se l'elemento specificato in x non esiste allora viene generato un'errore.

`list.count(x)`

Restituisce il numero di volte che x compare nella lista.

`list.sort()`

Ordina gli elementi della lista.

`list.reverse()`

Rovescia l'ordine degli elementi della lista.

`list.copy()`

Restituisce una copia superficiale della lista. Equivale a `a[:]`.

Ecco un' esempio che utilizza la maggior parte dei metodi delle liste:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
```

Potreste aver notato che metodi tipo `insert`, `remove` o `sort` non restituiscono valori ma solo `None` [1] Questo è un principio di progettazione per tutte le strutture dati mutabili in Python.

5.1.1. Utilizzo Delle Liste Come Pile.

I metodi delle liste rendono molto facile utilizzare una lista come una pila, dove l'ultimo elemento aggiunto è il primo elemento recuperato ("*last-in, first-out*"). Per aggiungere un elemento in fondo alla pila, utilizzare `append()`. Per recuperare un elemento dal fondo della pila, utilizzare `pop()` senza un indice esplicito. Per esempio:

```
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
```

```
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

5.1.2. Uso Delle Liste e Delle Code.

È anche possibile utilizzare un elenco come coda, dove il primo elemento aggiunto è il primo ad essere prelevato ("*first-in, first-out*"), tuttavia, le liste non sono efficaci per questo scopo. Mentre l'accodamento e il prelevamento dal basso sono veloci, fare inserimenti o prelevamenti dall'inizio di una lista risulta lento perché tutti gli altri elementi devono essere spostati uno ad uno.

Per implementare una coda, è possibile usare `collections.deque` che è stato progettato per avere accodamenti e prelevamenti veloci da entrambe le estremità. Per esempio:

```
>>> from collections import deque
>>> coda = deque(["Enrico", "Gianni", "Michele"])
>>> coda.append("Aldo")                  # Aldo aggiunto
>>> coda.append("Stefano")               # Stefano aggiunto
>>> coda.popleft()                       # il primo ora esce
'Enrico'
>>> coda.popleft()                       # ora esce il secondo
'Gianni'
>>> coda                                 # chi rimane in ordine arrivo
deque(['Michele', 'Aldo', 'Stefano'])
```

5.1.3. Comprensioni Di Lista.

Le comprensioni di lista forniscono un modo conciso per creare liste. Le applicazioni più comuni sono per fare nuovi elenchi in cui ogni elemento è il risultato di alcune operazioni applicate a ciascun membro di un'altra sequenza o iterabile, o per creare una sotto sequenza di quegli elementi che soddisfano una determinata condizione.

Ad esempio, supponiamo di voler creare un elenco di quadrati come:

```
>>> quadrati = []
>>> for x in range(10):
...     quadrati.append(x**2)
...
>>>  quadrati
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

E' possibile ottenere lo stesso risultato con:

```
quadrati = [x**2 for x in range(10)]
```

Che è' anche l'equivalente di `quadrati = list(map(lambda x: x**2, range(10)))`, ma è più conciso e più leggibile.

Un comprensione di lista, compone i tasselli che contengono un'espressione seguita da una clausola `for`, quindi nessuna o più clausole `for` o `if`. Il risultato sarà un nuovo elenco risultante dalla valutazione dell'espressione nel contesto delle clausole `for` e `if` che seguono. Ad esempio, questo esempio, combina gli elementi di due liste se non sono uguali:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Ed esso è equivalente a:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Notate come l'ordine dei `for` e dell' `if` sia lo stesso in entrambi gli esempi.

Se l'espressione è una tupla (ad esempio `(x, y)` nell'esempio precedente), esso deve essere racchiuso fra parentesi.

```
>>> # crea una nuova lista con i valori raddoppiati.
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # Filtra la lista per escludere i valori negativi.
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # assegna alla funzione tutti gli elementi.
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # chiama un metodo per ogni elemento.
>>> fruttafresca = ['  banana', '  more ', 'passiflora  ']
>>> [weapon.strip() for weapon in fruttafresca]
['banana', 'more', 'passiflora']
>>> # crea una lista di 2 tuples tipo (numero, quadrato).
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # le tuple, devono essere fra parentesi, altrimenti si ha errore.
>>> [x, x**2 for x in range(6)]
  File "<stdin>", line 1, in ?
    [x, x**2 for x in range(6)]
               ^
SyntaxError: invalid syntax
```

```
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Una comprensione di lista, può contenere espressioni complesse e funzioni nidificate:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

5.1.4. Comprensioni Di Lista Nidificate.

L'espressione iniziale in un elenco di comprensione può essere qualsiasi espressione arbitraria, tra cui un altro elenco di comprensione.
Si consideri il seguente esempio di una matrice 3x4 implementato come una lista di 3 liste di lunghezza 4:

```
>>> matrice = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

La seguente lista di comprensione trasporrà righe e colonne:

```
>>> [[riga[i] for riga in matrice] for i in intervallo(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Come abbiamo visto nella sezione precedente, la lista di comprensione nidificata viene valutata nel contesto del `for` che lo segue, quindi questo esempio è equivalente a:

```
>>> trasposto = []
>>> for i in intervallo(4):
...     trasposto.append([riga[i] for riga in matrice])
...
>>> trasposto
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Che a sua volta equivale a:

```
>>> trasposto = []
>>> for i in intervallo(4):
...     # le 3 linee seguenti implementano la list comp nidificata
...     trasposto_riga = []
...     for riga in matrice:
```

```
...         trasposto_riga.append(riga[i])
...     trasposto.append(trasposto_riga)
...
>>> trasposto
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Nella pratica, si dovrebbero preferire funzioni precostituite per situazioni complesse. La funzione `zip()` per esempio fa un grande lavoro in questa situazione riducendo inoltre la possibilità di errore:

```
>>> list(zip(*matrice))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

vedere anche *Unpacking Argument Lists* per maggiori dettagli sull'asterisco della prima linea.

5.2. L'Istruzione `del`.

C'è un modo per rimuovere un elemento da un elenco dato il suo indice invece del suo valore: usando l'istruzione `del` . Questa differisce dal metodo `pop()`, che restituisce un valore. L'istruzione `del` può anche essere usata per rimuovere gruppi da un elenco o cancellare l'intero elenco (che precedentemente abbiamo ottenuto assegnando una lista vuota al gruppo). Per esempio:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` inoltre, può essere usata per cancellare intere variabili:

```
>>> del a
```

Riferirsi ad `a` se `a` non esiste genera un'errore (almeno fino a quando un altro valore viene assegnato ad esso). Troveremo altri usi di `del` successivamente.

5.3. Tuple e Sequenze.

Abbiamo visto come le liste e le stringhe hanno molte caratteristiche comuni, come le operazioni di indicizzazione e suddivisione che sono due esempi sui tipi di dati sequenziali (vedere *Sequence Types — list, tuple, range*). Dal momento che Python è un linguaggio in evoluzione, possono essere aggiunti altri tipi di dati sequenziali. Ma a completare la richiesta, esiste un' altro tipo di dati sequenziale: la tupla. Essa condivide le caratteristiche tipiche delle liste e delle stringhe, ma a differenza di loro la tupla non può essere modificata.

Per instanziare una tupla, basta assegnare una serie di valori separati da una virgola:

```
>>> t = 12345, 54321, 'salve!'
>>> t[0]
12345
>>> t
(12345, 54321, 'salve!')
>>> # le tuple possono essere nidificate:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'salve!'), (1, 2, 3, 4, 5))
>>> # le tuple sono immutabili:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Come si può notare, la stampa delle tuple viene sempre rappresentata fra parentesi tonde, questo per fare in modo che si possano interpretare meglio eventuali tuple annidate. Le tuple possono essere create anche senza le parentesi tonde anche se spesso sono necessarie in processi più complessi. Non è possibile fare assegnamenti ai singoli elementi di una tupla, tuttavia è possibile creare tuple che contengono oggetti mutabili, come le liste (**veramente potente**).

Sebbene le tuple possono sembrare simili alle liste, sono spesso utilizzate in situazioni diverse e per scopi diversi. Le tuple sono *immutable* (immutabili), e di solito contengono una sequenza eterogenea di elementi a cui si accede tramite l'estrazione (vedere più avanti in questa sezione) o d'indicizzazione (o anche l'attributo nel caso di `namedtuples`). Le liste sono *mutable* (mutabili), e i loro elementi di solito sono omogenei e sono accessibili scorrendo l'elenco.

Un problema particolare è la costruzione di tuple contenenti 0 o 1 elementi: la sintassi contiene alcune stranezze. Tuple vuote vengono costruite usando una coppia vuota di parentesi; una tupla con un elemento è costruita seguendo un valore con una virgola (non è sufficiente a racchiudere un singolo valore tra parentesi). Brutto, ma efficace. Per esempio:

```
>>> vuota = ()
>>> singola = 'salve',    # notare la virgola al seguito.
>>> len(vuota)
0
>>> len(singola)
1
>>> singola
('salve',)                # riportata anche in fase di stampa.
```

L'istruzione `t = 12345, 54321, 'ciao!'` è un esempio di impacchettamento di una tupla: i valori

`'12345, 54321, 'ciao!'` sono riuniti nella tupla. E' anche possibile l'operazione inversa:

```
>>> x, y, z = t
```

Questo si chiama, in modo abbastanza appropriato, sequenza di spacchettamento e funziona per qualsiasi sequenza sul lato destro. La sequenza di spacchettamento, richiede che vi siano altrttante variabili sul lato sinistro del segno uguale (=) per gli elementi della sequenza. Si noti che l'assegnazione multipla è in realtà solo una combinazione di impacchettamento su tupla e una sequenza di spacchettamento. La mancanza di uno o più elementi a sinistra per lo spacchettamento, provoca un'errore.

5.4. Insiemi o Sets.

Python include anche un tipo di dati per gli insiemi o sets. Un insieme, è un gruppo di elementi non ordinato senza duplicati. Utulizzi di base includono il test per impedire gli elementi duplicati. Gli insiemi di oggetti supportano anche operazioni matematiche come l'unione, l'intersezione, la differenza, e la differenza simmetrica.

Le parentesi graffe o la funzione `set()` possono essere usati per creare insiemi. Nota: per creare un insieme vuoto si deve usare `set()`, non `{}`; quest'ultima crea un dizionario vuoto, una struttura di dati che discuteremo nella prossima sezione.

Questa è una breve dimostrazione:

```
>>> cestino = {'mela', 'arancia', 'mela', 'pera', 'arancia',
'banana'}
>>> print(cestino)                      #notare l'assenza dei duplicati.
{'arancia', 'banana', 'pera', 'mela'}
>>> 'arancia' in cestino                #test veloce dell'elemento.
True
>>> 'fragole' in cestino
False

>>> # Dimostrazione dell'uso di set().
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                    # lettere uniche in a.
{'a', 'r', 'b', 'c', 'd'}
>>> a - b                                # lettere in a ma non in b.
{'r', 'd', 'b'}
>>> a | b                                # lettere sia in a che in b.
{'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
>>> a & b                                # lettere solo in a e b.
{'a', 'c'}
>>> a ^ b                             # lettere in a o b ma non entrambe
{'r', 'd', 'b', 'm', 'z', 'l'}
```

Analogamente alle comprensioni di lista (*list comprehensions*) sono supportati anche set di comprensioni:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
{'r', 'd'}
```

5.5. Dizionari.

Un altro tipo di dati molto utile integrato in Python è il dizionario (vedere *Mapping Types — dict*). I dizionari si trovano a volte in altri linguaggi come "memorie associative" o "array associativi". A differenza delle sequenze, che sono indicizzate da una serie di numeri, i dizionari sono indicizzati da chiavi, che possono essere di qualsiasi tipo immutabile compreso stringhe e numeri. Le tuple possono essere usate come chiavi se contengono solo stringhe, numeri o tuple; se una tupla contiene un qualsivoglia oggetto mutabile direttamente o indirettamente, non può essere giustamente utilizzato come una chiave. Non è possibile utilizzare le liste come chiavi, dal momento che esse possono essere modificate usando le assegnazioni a indice, assegnazioni suddivisioni, o metodi come `append()` e `extend()`.

E 'meglio pensare a un dizionario come un insieme non ordinato di coppie chiave:valore, con il requisito che le chiavi sono uniche (all'interno di un dizionario). Una coppia di parentesi graffe crea un dizionario vuoto: `{}`. L'inserimento di un elenco di coppie chiave:valore all'interno delle parentesi graffe, viene effettuato separando i valori usando delle virgole. Questo è anche il modo in cui i dizionari sono visualizzati sullo schermo.

Le operazioni principali su un dizionario sono la memorizzazione di un valore con una qualche chiave e l'estrazione del valore per mezzo della stessa. E 'anche possibile eliminare una chiave:valore con `del`. Se si memorizza un valore utilizzando una chiave già in uso, il vecchio valore associato a tale chiave viene sostituito. Cercare di estrarre un valore utilizzando una chiave inesistente, genera un'errore.

Utilizzare `list(d.keys())` su un dizionario, restituisce un elenco di tutte le chiavi usate nel dizionario, in ordine arbitrario (se invece volete averlo ordinato, basta usare `sorted(d.keys())`). [2] Per verificare se una singola chiave è presente nel dizionario, utilizzare la parola chiave `in`.

Ecco un piccolo esempio sull'uso dei dizionari:

```
>>> tel = {'gigi': 4098, 'nino': 4139}
>>> tel['guido'] = 4127
>>> tel
{'nino': 4139, 'guido': 4127, 'gigi': 4098}
>>> tel['gigi']
4098
>>> del tel['nino']
>>> tel['ugo'] = 4127
>>> tel
{'guido': 4127, 'ugo': 4127, 'gigi': 4098}
>>> list(tel.keys())
['ugo', 'guido', 'gigi']
>>> sorted(tel.keys())
['guido', 'ugo', 'gigi']
```

```
>>> 'guido' in tel
True
>>> 'gigi' not in tel
False
```

L'istruzione `dict()` costruisce dizionari direttamente da sequenze di coppie chiave-valore:

```
>>> dict([('nino', 4139), ('guido', 4127), ('giacomo', 4098)])
{'nino': 4139, 'giacomo': 4098, 'guido': 4127}
```

Inoltre, dict comprehensions possono essere usati per creare dizionari da chiavi arbitrarie e valore d' espressioni:

```
>>> {x: x**2 for x in (2, 4, 6)}
{2: 4, 4: 16, 6: 36}
```

Quando le chiavi sono semplici stringhe, è più facile specificare le coppie che utilizzano argomenti chiave:

```
>>> dict(ale=4139, guido=4127, gianni=4098)
{'ale': 4139, 'gianni': 4098, 'guido': 4127}
```

5.6. Tecniche di Ciclo.

Il metodo `items()`, viene utilizzato con un ciclo per scorrere un dizionario per recuperare gli elementi presenti.

```
>>> cavalieri = {'gallahad': 'il puro', 'robin': 'il coraggioso'}
>>> for k, v in cavalieri.items():
...     print(k, v)
...
gallahad il puro
robin il coraggioso
```

Iterando in una sequenza, l'indice di posizione e il valore corrispondente possono essere richiamati allo stesso tempo utilizzando la funzione `enumerate()`.

```
>>> for i, v in enumerate(['tic', 'tac', 'toe']):
...     print(i, v)
...
0 tic
1 tac
2 toe
```

Per un ciclo su due o più sequenze contemporaneamente, le voci possono essere accoppiate con la

funzione `zip()`.

```
>>> domanda = ['nome', 'obiettivo', 'colore preferito']
>>> risposta = ['lancillotto', 'il sacro gral', 'blu']
>>> for q, a in zip(domanda, risposta):
...     print('il vostro {0}?  è {1}.'.format(q, a))
...
il vostro nome?  è lancillotto.
il vostro obiettivo?  è il sacro gral.
il vostro colore preferito?  è blu.
```

Per un ciclo su una sequenza in senso inverso, prima specificare la sequenza in avanti e poi chiamare la funzione`reversed()`.

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
9
7
5
3
1
```

Per ordinare con un ciclo una sequenza di dati, utilizzare la funzione `sorted()`, che restituisce una nuova lista ordinata lasciando inalterata l'originale.

```
>>> cestino = ['mela', 'fragole', 'pera', 'arancia']
>>> for f in sorted(set(cestino)):
...     print(f)
...
arancia
fragole
mela
pera
```

Per modificare una sequenza si effettua l'iterazione all'interno del ciclo (ad esempio per duplicare alcune voci), si consiglia di eseguire una copia preventiva. Iterare su una sequenza non implica una copia. La notazione slice cioè suddivisione la rende particolarmente conveniente:

```
>>> voci = ['gatto', 'finestra', 'defenestrato']
>>> for w in voci[:]:  # Cicla su una copia slice dell'intera lista.
...     if len(w) > 6:
...         voci.insert(0, w)
...
>>> voci
['defenestrato', 'finestra', 'gatto', 'finestra', 'defenestrato']
```

5.7. Approfondimento sulle condizioni.

Le condizioni utilizzate con le dichiarazioni `while` ed `if` , possono contenere tutti gli operatori, non solo quelli di confronto. Gli operatori di confronto `in` e `not`, testano se un valore che viene rilevato o no in una sequenza. Gli operatori `is` e `is not`, confrontano se due oggetti sono o no in realtà lo stesso oggetto; questo vale solo per oggetti mutabili, come le liste. Tutti gli operatori di confronto hanno la stessa priorità, che è inferiore a quella di tutti gli operatori numerici.

I confronti possono essere anche concatenati. Ad esempio, **`a < b == c`** `a` è minore di `b` ed inoltre `b` sia uguale `c`.

I confronti possono essere combinati utilizzando gli operatori booleani `and` e `or`, e il risultato di un confronto (o di qualsiasi altra espressione booleana) possono essere negativi con `not`. Questi hanno priorità inferiore rispetto a operatori di confronto, tra loro, `not` ha la priorità più alta e `or` la più bassa, in modo che `A and not B or C` è equivalente a `(A and (not B)) or C`. Come sempre, l'utilizzo di parentesi, consente di ottenere la priorità desiderata.

Gli operatori booleani `and` e `or` sono cosiddetti operatori di cortocircuito: i loro argomenti vengono valutati da sinistra a destra, e la valutazione termina non appena il risultato è determinato. Ad esempio, se `A` e `C` sono vere, ma `B` è falsa, `A and B and C` non sarà valutata l'espressione `C`. Quando viene usato un valore generale e non uno booleano, il valore di ritorno di un operatore di corto circuito è l'ultimo argomento valutato.

È possibile assegnare il risultato di un confronto di altre espressioni booleane a una variabile. Per esempio:

```
>>> stringa1, stringa2, stringa3 = '', 'Roma', 'nuova Dely'
>>> no_null = string1 or string2 or string3
>>> no_null
'roma'
```

Si noti che in Python, a differenza del C, l'assegnazione non può avvenire all'interno di espressioni. I programmatori C potrebbero lamentarsi di questo, ma questo evita quel tipo di problemi incontrati nei programmi in C tipo: digitando = in un'espressione quando invece s'intendeva == .

5.8. Confronto Di Sequenze e Altri Tipi.

Le sequenze di oggetti, possono essere confrontate con altri oggetti con lo stesso tipo di sequenza. Il confronto utilizza un ordinamento lessicografico: prima i primi due elementi vengono confrontati, e se questi differiscono ciò determina il risultato del confronto, se sono uguali, vengono confrontati i due elementi successivi, e così via, fino a quando la sequenza si esaurisce. Se due elementi da confrontare sono essi stessi sequenze dello stesso tipo, il confronto lessicografico viene effettuato ricorsivamente. Se tutti gli elementi di due sequenze risultano uguali, le sequenze sono considerate uguali. Se una sequenza è una sotto sequenza iniziale dell'altra, la sequenza più breve è la più piccola (minore) di uno. L'ordinamento lessicografico per le stringhe utilizza la codifica **Unicode** per ordinare i singoli caratteri. Ecco alcuni esempi di confronti tra sequenze dello stesso tipo:

```
(1, 2, 3)              < (1, 2, 4)
[1, 2, 3]              < [1, 2, 4]
'ABC' < 'C' < 'Pascal' < 'Python'
(1, 2, 3, 4)           < (1, 2, 4)
(1, 2)                 < (1, 2, -1)
(1, 2, 3)             == (1.0, 2.0, 3.0)
(1, 2, ('aa', 'ab'))   < (1, 2, ('abc', 'a'), 4)
```

Si noti che il confronto oggetti di diversi tipi con **<** or **>** è ammesso a condizione che gli oggetti abbiano adeguati metodi di confronto. Ad esempio, i tipi numerici misti vengono confrontati in base al loro valore numerico, così **0** è uguale a **0.0**, ecc Altrimenti, piuttosto che fornire un ordinamento arbitrario, l'interprete solleverà un'eccezione `TypeError`.

Note:

[1] Altri linguaggi, possono restituire l'oggetto mutato, che permette il metodo del concatenamento, come inserire `d->insert("a")->remove("b")->sort()`.

[2] Chiamare `d.keys()` restituirà una vista dell'oggetto dizionario. Esso supporta operazioni come prova l'adesione e iterazione, ma il suo contenuto non è indipendente dal dizionario originale - è solo una vista.

6. Moduli.

Se si esce dall'interprete Python e poi vi si rientra, le definizioni introdotte (funzioni e variabili) vengono perse. Pertanto, se si vuole scrivere un programma più lungo, è meglio usare un editor di testo per gestire l'input per l'interprete passandogli il file da eseguire invece dell'input diretto. Questo è noto come la creazione di uno script. Inoltre se il programma si allunga, è consigliabile dividerlo in più file per facilitarne la manutenzione. Se hai scritto una funzione comoda da utilizzare in più programmi, si consiglia di salvarla per poi utilizzarla invece che copiarla ogni volta.

Per fare in modo che ciò possa essere fatto, Python ha un modo per gestire il tutto. Da la possibilità di mettere tutto in un file da richiamarlo poi sia in modo interattivo che all'interno di altri file. Tale file si chiamano **moduli**. Le definizioni di un modulo possono essere importate in altri moduli o nel modulo principale (l'insieme di variabili di cui si ha accesso a in uno script eseguito al livello superiore e in modalità calcolatrice).

Un modulo praticamente non è altro che un file contenente definizioni e istruzioni Python. Il nome del file è il nome del modulo con aggiunto il suffisso `.py`. All'interno di un modulo, il nome del modulo (come una stringa) è disponibile come valore nella variabile globale `__name__`. Per esempio, usare il vostro editor di testo preferito per creare un file chiamato `fibo.py` nella directory corrente con i seguenti contenuti e salvatelo:

```
# Modulo numeri di Fibonacci.

def fib(n):    # serie di numeri Fibonacci fino ad n stampati.
    a, b = 0, 1
    while b < n:
```

```
        print(b, end=' ')
        a, b = b, a+b
    print()

def fib2(n): # serie di numeri Fibonacci fino ad n restituiti.
    risultato = []
    a, b = 0, 1
    while b < n:
        risultato.append(b)
        a, b = b, a+b
    return risultato
```

Ora immette dall'interprete Python il seguente comando:

```
>>> import fibo
```

Questo non immette i nomi delle funzioni definite in `fibo` direttamente nella tabella dei simboli corrente, immette solo il riferimento al modulo `fibo`. Facendo riferimento a questo modulo, è possibile usare la funzione in esso contenuta.

```
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

Notare che la funzione `fib`, viene richiamata dal modulo `fibo` con la notazione del punto (`fibo.fib`). In effetti il modulo viene tratto in Python come una classe a tutti gli effetti, anzi per essere più precisi come un'oggetto. Questo argomento verrà ripreso più avanti in questa guida.

Se avete intenzione di usare spesso una funzione è possibile assegnargli un nome locale:

```
>>> fib = fibo.fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

6.1. Ancora Sui Moduli.

Un modulo può contenere istruzioni eseguibili così come la definizioni di funzione . Queste dichiarazioni hanno lo scopo di inizializzare il modulo. Esso viene caricato solo all'occorrenza e cioè quando un riferimento ad esso viene esplicitamente richiamato altrimenti c'è solo un riferimento ad esso per il suo eventuale uso. [1] Sono inoltre eseguiti se il file viene eseguito come uno script cioè lanciato direttamente.

Ogni modulo ha la sua tabella dei simboli privata , che è usata come tabella dei simboli globale da tutte le funzioni definite nel modulo . Così, l'autore di un modulo può utilizzare le variabili globali nel modulo senza preoccuparsi di conflitti accidentali con le variabili globali di un' altro utente . D'altra parte , se si sa cosa si sta facendo è possibile gestire le variabili globali di un modulo con la stessa notazione usata per

riferirsi alle sue funzioni, `nome_modulo.nome_voce`.

I moduli possono importare altri moduli. E' una consuetudine, ma non indispensabile mettere tutte le istruzioni `import` all'inizio di un modulo (o script per questo contesto). I nomi dei moduli importati vengono inseriti nella tabella dei simboli globale del modulo di importazione. C'è una variante dell'istruzione import che importa nomi da un modulo direttamente nella tabella dei simboli del modulo di importazione. Per esempio :

```
>>> from fibo import fib, fib2
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Questo non introduce il nome del modulo dal quale vengono importati nella tabella dei simboli locali (così nell'esempio, `fibo` non è definito), ma solo la funzione richiesta. C'è inoltre anche una variante per importare tutti i nomi definiti in un modulo:

```
>>> from fibo import *
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Questo importa tutti i nomi tranne quelli che iniziano con un carattere di sottolineatura (underscore (**_**)). Nella maggior parte dei casi i programmatori Python non utilizzano questo strumento in quanto introduce un insieme sconosciuto di nomi nell'interprete e di probabile non utilizzo nascondendo alcune cose che sono già state definite.

Si noti che in generale la pratica di importare con ***** da un modulo o un pacchetto è malvista, dato che spesso comporta poca leggibilità del codice. Tuttavia, è bene usarla per risparmiare digitazioni eccessive in sessioni interattive.

Note:

Per ragioni di efficienza, ogni modulo viene importato solo una volta per sessione interprete. Pertanto, se si cambiano i moduli, è necessario riavviare l'interprete o se è solo un modulo che si desidera testare in modo interattivo, utilizzare `imp.reload()`, ad esempio:

```
import imp; imp.reload(nome_modulo).
```

6.1.1. Eseguire Moduli Come Script.

Basta avviare Python da linea di comando, associandogli un file con estensione **.py** in questo modo:

```
python fibo.py <argomenti opzionali>
```

Il codice presente nel modulo sarà eseguito, come se si fosse importato, ma con il `__name__` impostato su `__main__`. Questo significa l'aggiunta di questo codice alla fine del modulo:

```
if __name__ == "__main__":
    import sys
    fib(int(sys.argv[1]))
```

è possibile rendere il file utilizzabile come un script così anche come un modulo importabile, questo perché il codice che analizza la riga di comando viene eseguito solo se nel file modulo viene eseguito "**main**":

verifichiamolo meglio con una prova. In questo caso il modulo viene eseguito.

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

Se il modulo è importato, in questo caso il codice non viene eseguito:

```
>>> import fibo
>>>
```

Questo è spesso utilizzato sia per fornire una comoda interfaccia utente per un modulo, o per scopi di test (che eseguono il modulo come uno script esegue una serie di test).

6.1.2. Il Modulo Ricerca Path.

Quando un modulo per esempio di nome `prova.py` viene importato, l'interprete prima ricerca un modulo esistente che abbia quel nome, se non lo trova, allora lo cerca in una lista di directory contenuta nella variabile `sys.path`, la quale viene inizializzata da queste posizioni:

- La directory contenente lo script ingresso (o la directory corrente).
- PYTHONPATH (una variabile ambiente contenete i percorsi in cui cercare, con la stessa sintassi PATH variabile utilizzata dalla shell).
- Nell'installazione dipendente predefinita.

Dopo l'inizializzazione, i programmi Python possono modificare `sys.path`. La directory contenente lo script sarà messa all'inizio del percorso di ricerca, all'inizio del percorso di libreria standard. Questo significa che verranno caricati gli script in quella directory, invece che moduli dello stesso nome nella directory di libreria. Questo è un errore almeno che la tale sostituzione non sia desiderata. Vedere la sezione Moduli standard *Standard Modules* per ulteriori informazioni.

6.1.3. Files Python “Compilati”.

Per accelerare l'esecuzione dei moduli, Python memorizza nella cache la versione compilata di ciascun modulo nella directory `__pycache__` sotto il nome `module.version.pyc`, dove `version` codifica il formato del file compilato, che contiene in genere il numero di versione di Python.
Ad esempio, in CPython rilascio 3.3 la versione compilata di prova.py sarebbe memorizzata nella cache cioè il luogo dove Python conserva molte informazioni come `__pycache__/prova.cpython-33.pyc`. Questa convenzione di denominazione consente a moduli compilati da versioni diverse e diverse versioni di Python di coesistere senza conflitti.

Python controlla la data di modifica del file sorgente con la versione compilata per vedere se è immutato oppure deve essere ricompilato. Questo processo è completamente automatico. Inoltre, i moduli compilati sono indipendenti dalla piattaforma, in modo che la stessa libreria possa essere condivisa tra sistemi con differenti architetture.

Python non controlla la cache in due circostanze.

- Quando ricompila il modulo che viene caricato direttamente dalla riga di comando.
- Quando c'è il modulo di origine.

Per il supporto a una distribuzione senza sorgente (solo compilato), il modulo compilato deve essere nella directory dei sorgenti, e non ci deve essere un modulo di origine.

Alcuni consigli per i più esperti:

- È possibile utilizzare l'opzione -O o -OO per attivare il comando Python che riduce le dimensioni di un modulo compilato ottimizzando le risorse. L'opzione `-O` rimuove tutte le affermazioni proprie delle dichiarazioni, L'opzione `-OO` rimuove sia le affermazioni delle dichiarazioni che la documentazione togliendo le stringhe __doc__. Poiché alcuni programmi possono fare assegnamento sulla loro disposizione, si consiglia di utilizzare questa opzione solo se si sa cosa si sta facendo. I moduli "ottimizzati", hanno un suffisso . Pyo invece che . Pyc e di solito sono più piccoli. Le versioni future potrebbero cambiare il modo con cui viene eseguita l'ottimizzazione.

- Un programma non viene eseguito più velocemente quando viene letto da un file pyo o pyc rispetto a quando viene letto da un file py. L'unica cosa che è più veloce su pyc o pyo è la velocità con cui sono caricati i files. Questa possibilità va usata solo per programmi di dimensione veramente grande. Questo può tornare utile quando si lavora in rete, mentre in stand alone con la velocità delle attuali macchine il tutto perde di senso.

- Il modulo `compileall` (compila tutto) può creare file con estensione .Pyc (o .Pyo file quando viene usato -O o -OO) per tutti i moduli in una directory. Per maggiori dettagli su questo processo, compreso un diagramma di flusso delle decisioni, dare un'occhiata a PEP 3147.

6.2. Moduli Standard.

Python viene fornito con una libreria di moduli standard, descritti in un documento separato, la “Python Library Reference” (di seguito "Libreria di riferimento"). Alcuni moduli sono parte integrata dell'interprete che possiamo definire come indispensabili, questi forniscono l'accesso a operazioni che non fanno parte del nucleo del linguaggio ma cionondimeno sono interne, per garantire efficienza o per fornire accesso alle primitive del sistema operativo come ad esempio le chiamate di sistema. L'insieme di tali moduli è un'opzione di configurazione che dipende anche dalla piattaforma sottostante. Ad esempio, il modulo `winreg` viene fornito solo su sistemi Windows. Un modulo particolare che merita una certa attenzione è `sys`, che è specifico in ogni interprete Python che dipende appunto dalla piattaforma sottostante. Le variabili `sys.ps1` e `sys.ps2` definiscono le stringhe utilizzate come prompt primario e secondario “**>>>**” e “**...**”:

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
```

```
>>> sys.ps1 = 'Mio Prompt: '
Mio Prompt print('Yuck!')
Yuck!
Mio Prompt:
```

Queste due variabili sono funzionali all'interprete solo in modalità interattiva (in altri contesti sarebbe un non senso).

La variabile `sys.path` è una lista di stringhe che determina il percorso di ricerca dei moduli. Questa viene inizializzata ad un percorso predefinito facendo riferimento alla variabile d'ambiente `PYTHONPATH`, o da una precostituita se `PYTHONPATH` non è impostata. È possibile apportare modifiche usando le operazioni standard permesse con le liste:

```
>>> import sys
>>> sys.path.append('/usr/pippo/lib/python')
```

6.3. La Funzione `dir()`.

La funzione precostituita `dir()` , è usata per vedere quali nomi un modulo fornisce restituiti sotto forma di una lista ordinata di stringhe:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__egginsert', '__excepthook__',
 '__loader__', '__name__', '__package__', '__plen', '__stderr__',
 '__stdin__', '__stdout__', '_clear_type_cache', '_current_frames',
 '_debugmallocstats', '_getframe', '_home', '_mercurial',
'_xoptions',
 'abiflags', 'api_version', 'argv', 'base_exec_prefix',
'base_prefix',
 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags',
'float_info',
 'float_repr_style', 'getcheckinterval', 'getdefaultencoding',
 'getdlopenflags', 'getfilesystemencoding', 'getobjects',
'getprofile',
 'getrecursionlimit', 'getrefcount', 'getsizeof',
'getswitchinterval',
 'gettotalrefcount', 'gettrace', 'hash_info', 'hexversion',
 'implementation', 'int_info', 'intern', 'maxsize', 'maxunicode',
 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
 'platform', 'prefix', 'ps1', 'setcheckinterval', 'setdlopenflags',
 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace',
 'stderr', 'stdin', 'stdout', 'thread_info', 'version',
'version_info',
 'warnoptions']
```

Senza argomenti, `dir()` visualizza una lista di nomi attualmente definita:

```
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Notate che nella lista, sono presenti tutti i tipi di nomi: Variabili, funzioni, moduli ecc.

`dir()` non produce una lista delle variabili e delle funzioni precostituite. Per ottenere questo, bisogna utilizzare il modulo standard `builtins`:

```
>>> dir(builtins)
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException',
 'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
 'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
 'ConnectionRefusedError', 'ConnectionResetError',
'DeprecationWarning',
 'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
 'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
 'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
 'ImportWarning', 'IndentationError', 'IndexError',
'InterruptedError',
 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
 'MemoryError', 'NameError', 'None', 'NotADirectoryError',
'NotImplemented',
 'NotImplementedError', 'OSError', 'OverflowError',
 'PendingDeprecationWarning', 'PermissionError',
'ProcessLookupError',
 'ReferenceError', 'ResourceWarning', 'RuntimeError',
'RuntimeWarning',
 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning',
'UserWarning',
 'ValueError', 'Warning', 'ZeroDivisionError', '_',
'__build_class__',
 '__debug__', '__doc__', '__import__', '__name__', '__package__',
'abs',
 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes',
'callable',
 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec',
'exit',
 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals',
'hasattr',
 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance',
'issubclass',
 'iter', 'len', 'license', 'list', 'locals', 'map', 'max',
'memoryview',
```

```
 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print',
'property',
 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr',
'slice',
 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type',
'vars',
 'zip']
```

6.4. I Packages (o pacchetti).

I packages (pacchetti) sono un modo per strutturare gli **namespace** (spazio dei nome riguardanti le variabili) nei moduli di Python utilizzando "i nomi dei moduli con punteggiatura". Ad esempio, il nome del modulo A.B designa un sotto-modulo chiamato B in un pacchetto chiamato A. L'uso di nomi di modulo punteggiati esime gli autori di pacchetti multi-modulo, tipo NumPy o PythonImagingLibrary o qualsiasi altro modulo per esempio di doversi preoccupare che le proprie variabili possano entrare in conflitto con altre di altri moduli.

si supponga di voler progettare una collezione di moduli (cioè un package) per la gestione uniforme dei file audio e dati audio. Ci sono molti diversi formati di file audio (di solito riconoscibili dalla loro estensione, per esempio: `.wav`, `.aiff`, `.au`), quindi un package potrebbe essere necessario per creare e mantenere una collezione crescente di moduli per la conversione tra i vari formati di file. Ci sono anche molte operazioni che si potrebbe voler effettuare sui dati (quali il mixaggio, l'aggiunta di eco, applicare una funzione di equalizzazione, creare un effetto stereo artificiale), quindi in aggiunta, potreste scrivere un quantitativo notevole di moduli per eseguire queste operazioni. Ecco una possibile struttura per il pacchetto (espresso in termini di file system gerarchico):

```
sound/                          livello alto del package
      __init__.py               Inizializzazione del package sound
      formats/                  Sotto package per i file di conversione
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/                  Sotto package per gli effetti suono
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/                    Sotto package per i filtri
              __init__.py
              equalizer.py
              vocoder.py
```

```
        karaoke.py
      ...
```

Quando si importa un package, Python cerca attraverso le directory in `sys.path` le sotto directory del package. I file `__init__.py` permettono a Python di trattare le directory come contenitori di packages, questo viene fatto per evitare che directory con un nome stringa in comune, nascondano involontariamente moduli validi che si verificano in seguito nel percorso di ricerca del modulo. Nel caso più semplice, `__init__.py` può essere solo un file vuoto, ma può anche eseguire codice di inizializzazione per il pacchetto o impostare la variabile `__all__` come descritto più avanti.

Gli utenti del package possono importare singoli moduli del package, per esempio:
```
import Sound.Effects.echo
```

Questo carica il modulo Sound.Effects.echo che deve essere referenziato con il suo nome completo.
```
sound.effects.echo.echofilter (input, output, delay = 0.7, atten = 4)
```

Un modo alternativo per importare il sotto-modulo è:
```
from Sound.Effects import echo
```

Questo comando, carica il sotto-modulo echo, e lo rende disponibile senza il prefisso del pacchetto, in modo che possa essere utilizzato come segue:
```
echo.echofilter (input, output, delay = 0.7, atten = 4)
```

Ancora un'altra variante è quella di importare la funzione o la variabile desiderata direttamente:
```
from Sound.Effects.echo echofilter import
```

Di nuovo, questo carica il sotto-modulo echo, ma rende la sua funzione echofilter () direttamente disponibile:
```
echofilter (input, output, delay = 0.7, atten = 4)
```

Come si vede, Python mette a disposizione molte strade per ottenere la stessa cosa! Questo fa di Python un linguaggio potente, semplice e personalizzabile. Cosa chiedere di più.

Si noti che quando si usa `from package import voce`, l'elemento può essere o un modulo (sub package del pacchetto), o qualche altro nome definito nel package, come una funzione, una classe o una variabile. I primi test sulle istruzione `import` vengono fatti per verificare se l'elemento è definito all'interno del pacchetto, se così non è, allora si presuppone che sia un modulo e tenta di caricarlo. Se non riesce a trovarlo, un'eccezione `ImportError` viene sollevata.

Al contrario, quando si usa una sintassi del tipo `import voce.sottovoce.sottosottovoce`, ogni elemento eccetto l'ultimo deve essere un pacchetto, l'ultimo elemento può essere un modulo o un package ma non può essere una classe o una funzione o una variabile definita nel punto precedente .

6.4.1. Importazione * From Da Un Package.

Ora, cosa succede quando l'utente scrive `from sound.effects import *`? Idealmente, si potrebbe sperare che questo in qualche modo va al file system, per constatare che siano presenti nel pacchetto dei sotto-moduli, e li importa tutti. Questo potrebbe richiedere molto tempo inoltre l'importazione di sotto-moduli potrebbe avere effetti collaterali indesiderati che possono accadere solo

quando il sotto-modulo è esplicitamente importato.

L'unica soluzione è che l'autore del pacchetto fornisca un indice esplicito del pacchetto. L'istruzione import utilizza la seguente convenzione: se il codice `__init__.py` di un pacchetto definisce una lista di nome`__all__`, esso viene considerato come l'elenco dei nomi di moduli che devono essere importati quando `from package import *` viene richiesto. Spetta all'autore del pacchetto mantenere questo elenco aggiornato qualora una nuova versione del pacchetto venga rilasciata. Gli autori dei packages possono anche decidere di non sostenere questa pratica se non vedono l'utilità di importare con ***** dal loro package. Ad esempio, il file `sound/effects/__init__.py` potrebbe contenere il seguente codice:

```
__all__ = ["echo", "surround", "reverse"]
```

Ciò significa che `from sound.effects import *` importerebbe i tre sotto-moduli sopracitati del package `sound`.

Se`__all__` non è definito, l'istruzione `from sound.effects import *` non importa tutti i sotto moduli del package `sound.effects` nel namespace (spazio dei nomi) corrente, ma si assicura solo che il package `sound.effects` sia stato importato (possibilmente l'esecuzione con qualsiasi codice di inizializzazione in `__init__.py`) e quindi importa qualunque nome sia definito nel pacchetto. Ciò include qualsiasi nome definito (e i sotto moduli esplicitamente caricati) in `__init__.py`. Esso comprende anche eventuali sotto moduli del packages che siano stati esplicitamente caricati da precedenti istruzioni. Considerate questo codice:

```
import sound.effects.echo
import sound.effects.surround
from sound.effects import *
```

In questo esempio, i moduli `echo` e `surround` sono importati nello spazio dei nomi corrente poiché vengono definiti nel package `sound.effects` quando viene eseguita l'istruzione `from...import *`. (Questo funziona anche quando `__all__` è definita.)

Sebbene alcuni moduli siano progettati per esportare solo nomi che seguono certi schemi quando si utilizza `import *`, essa è ancora considerata una cattiva pratica nel codice di produzione.
Ricordate, non c'è niente di sbagliato utilizzare `from Package import specifico_sottomodulo`! In realtà, questa è la notazione consigliata, a meno che il modulo importatore debba usare dei sotto moduli con lo stesso nome da packages differenti.

6.4.2. Referenze Intra-package.

Quando i pacchetti sono strutturati in sotto-package (come con il pacchetto `sound` nell'esempio), è possibile utilizzare l' import assoluto per riferirsi a sotto-moduli di packages fratelli. Ad esempio, se il modulo `sound.filters.vocoder` deve utilizzare il modulo `echo` nel pacchetto `sound.effects`, si può usare `from sound.effects import echo`.

Si possono scrivere anche import relativi, con il `from module import nome` di dichiarazioni di importazione. Questi import utilizzano dei punti per indicare i packages attuali e i packages di levello

superiore coinvolti nell'importazione. Dal modulo `surround` per esempio, è possibile utilizzare:

```
from . import echo
from .. import formats
from ..filters import equalizer
```

Si noti che le relative import, si basano sul nome del modulo corrente. Poiché il nome del modulo principale è sempre "`__main__`", moduli, impiegati come il modulo principale di una applicazione Python devono sempre utilizzare import assoluti.

6.4.3. Packages in Directory Multiple.

I packages, supportano un altro attributo speciale, `__path__`. Questo viene inizializzato per ottenere una lista contenente il nome della directory che contiene tutti i packages `__init__.py` prima che il codice in quel file venga eseguito. Questa variabile può essere modificata, così facendo influenza le future ricerche i moduli e i sotto-packages contenuti nel pacchetto.

Mentre questa caratteristica non è spesso necessaria, può essere utilizzata per estendere l'insieme dei moduli trovati in un package.

Note:

[1] In realtà le definizioni di funzione sono anche "dichiarazioni" che vengono 'eseguite; l'esecuzione di una definizione di funzione a livello di modulo entra con il nome nella tabella dei simboli globali del modulo.

7. Input e Output.

Esistono diversi modi per mostrare l'output di un programma; i dati possono essere stampati in una forma leggibile, o scritti in un file per un uso futuro. Questo capitolo tratterà alcune possibilità.

7.1. Formattazione Avanzata dell'Output.

Finora abbiamo incontrato due modi di scrivere valori: le dichiarazioni di espressioni e la funzione `print()`. Un terzo modo è usare il metodo `write()` dell'oggetto file, il file di output standard ha come riferimento lo `sys.stdout` (Vedere la Library Reference per ulteriori informazioni in proposito).

Spesso si desidera avere un maggiore controllo sulla formattazione dell'output invece che stampare semplicemente dei valori separati da spazio. Ci sono due modi per formattare l'output:

- il primo modo è quello di fare tutto con la gestione delle stringhe; tramite operazioni di divisione e di concatenazione con ciò è possibile creare qualsiasi presentazione che si possa immaginare. Il tipo stringa ha alcuni metodi che eseguono operazioni utili per la formattazione in considerazione di una determinata larghezza di colonna, che saranno discusse a breve.

- Il secondo modo è quello di utilizzare il modulo `string` che contiene il metodo `str.format()` e una classe modello (`Template`) che offre un modo diverso per sostituire i valori nelle stringhe.

Una questione rimane, naturalmente: come si fa a convertire i valori in stringhe? Fortunatamente, Python

ha un modo per convertire qualsiasi valore in una stringa, passando alle funzioni `repr()` o `str()` Il controllo dell'output.

La funzione `str()` è concepita per restituire una rappresentazione dei valori che sono abbastanza leggibili, mentre `repr()` è destinata a generare rappresentazioni che possono essere lette dall'interprete (che provocano l'eccezione Syntax Error se non c'è una sintassi equivalente). Per gli oggetti che non hanno una rappresentazione particolare per il contesto, `str()` restituirà lo stesso valore di `repr()`. Molti valori, come numeri strutture o liste e dizionari, mostrano la stessa rappresentazione utilizzando la funzione Strings, in particolare, hanno due distinte rappresentazioni.

Ecco alcuni esempi:

```
>>> s = 'Ciao mondo.'
>>> str(s)
'Ciao mondo.'
>>> repr(s)
"'Ciao mondo.'"
>>> str(1/7)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'Il valore di x è ' + repr(x) + ', e di y è ' + repr(y) +
'...'
>>> print(s)
Il valore di x è 32.5, e di y è 40000...
>>> # repr() aggiunge alla stringa le virgolette e la barra rovescia
... hello = 'Ciao mondo\n'
>>> hellos = repr(hello)
>>> print(hellos)
'Ciao mondo\n'
>>> # L'argomento di repr() può essere un oggetto Python
... repr((x, y, ('spam', 'uova')))
"(32.5, 40000, ('spam', 'uova'))"
```

Questi sono due modi per scrivere un tabella di quadrati e cubi:

1° modo

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Notare l'uso di 'end' sulla precedente linea
...     print(repr(x*x*x).rjust(4))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
```

```
 9  81  729
10 100 1000
```

2° modo

```
>>> for x in range(1, 11):
...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
...
 1   1    1
 2   4    8
 3   9   27
 4  16   64
 5  25  125
 6  36  216
 7  49  343
 8  64  512
 9  81  729
10 100 1000
```

Si noti che nel primo esempio, che uno spazio tra ogni colonna che è stata aggiunto dal metodo `print()` il quale, aggiunge sempre gli spazi tra i suoi argomenti.)

Questo esempio illustra il metodo `str.rjust()` per gli oggetti stringa, che giustifica a destra una stringa in un campo di una data ampiezza riempiendola di spazi a sinistra. Esistono metodi complementari `str.ljust()` e `str.center()`. Questi metodi non scrivono nulla, ma restituiscono una nuova stringa formattata. Se la stringa di input è troppo lunga, non viene troncata ma restituita intatta; questa scelta permette di non perdere dati a scapito di una formattazione che potrebbe non avere senso. Ma se lo scopo è quello di troncare comunque, è sempre possibile utilizzate la funzione di divisione del tipo `x.ljust(n)[:n]`.C'è un altro metodo, `str.zfill()`, che formatta una stringa numerica sulla sinistra con zeri. La si capisce in merito a segni più e meno:

```
>>> '12'.zfill(5)Input e Output
'00012'
>>> '-3.14'.zfill(7)
'-003.14'
>>> '3.14159265359'.zfill(5)
'3.14159265359'
```

L'aspetto nell'uso base del metodo `str.format()` si presenta così:

```
>>> print('Noi siamo i {} che dicono "{}!"'.format('cavalieri',
'Ni'))
Noi siamo i cavalieri che dicono "Ni!"
```

Le parentesi e i caratteri al loro interno (chiamati campi di formato) vengono sostituiti con gli oggetti passati nel metodo `str.format()`. Un numero tra parentesi graffe può essere usato per riferirsi alla posizione dell'oggetto passato al metodo `str.format()`.

```
>>> print('{0} e {1}'.format('pancetta', 'uova'))
pancetta e uova
>>> print('{1} e {0}'.format('pancetta', 'uova'))
uova e pancetta
```

Se argomenti di parole chiave vengono utilizzati nel metodo `str.format()`, i loro valori sono indicati utilizzando il nome dell'argomento.

```
>>> print('Questo {bevanda} è {aggettivo}.'.format(
...       bevanda='vino', aggettivo='veramente buono'))
Questo vino è veramente buono.
```

Argomenti posizionali e parole-chiave possono essere combinati in modo arbitrario.

```
>>> print('La vita di {0}, {1}, e {altro}.'.format('Gianni',
'Alfredo', altro='Giorgio'))

La vita di Gianni, Alfredo e Giorgio.
```

Le seguenti forme contratte, possono essere utilizzate per convertire i valori prima della formattazione; `'!a'` (applica `ascii()`), `'!s'` (applica `str()`) e `'!r'` (applica `repr()`) possono essere usati per convertire il valore prima della formattazione:

```
>>> import math
>>> print('Il valore di PI approssimato è {}.'.format(math.pi))
Il valore di PI approssimato è 3.14159265359.
>>> print('Il valore di PI approssimato è {!r}.'.format(math.pi))
Il valore di PI approssimato è 3.141592653589793.
```

Un `':'` opzionale seguito da una formattazione specifica, permette un maggiore controllo sul modo in cui il valore è formattato. L'esempio seguente arrotonda Pi a tre cifre dopo la virgola.

```
>>> import math
>>> print('Il valore di PI è approssimativamente
{0:.3f}.'.format(math.pi))
Il valore di PI è approssimativamente 3.142.
```

Passando un intero dopo il `':'` farà sì che il campo sia un numero minimo di caratteri estesi. Questo è utile per fare tabelle ben formattate.

```
>> tabella = {'Alfredo': 4127, 'Gianni': 4098, 'Andrea': 7678}
>>> for nome, telefono in tabella.items():
...     print('{0:10} ==> {1:10d}'.format(nome, telefono))
...
Gianni     ==>       4098
Andrea     ==>       7678
Alfredo    ==>       4127
```

Se si ha una stringa di formato davvero lunga che non si vuole dividere, sarebbe bello se si potesse fare riferimento alle variabili da formattare per nome invece che per posizione. Questo può essere fatto semplicemente passando il dizionario e utilizzando le parentesi quadre `'[]'` per accedere alle chiavi.

```
>>> tabella = {'Alfredo': 4127, 'Gianni': 4098, 'Andrea': 8637678}
>>> print('Gianni: {0[Gianni]:d}; Alfredo: {0[Alfredo]:d}; '
...       'Andrea: {0[Andrea]:d}'.format(tabella)
Gianni: 4098; Alfredo: 4127; Andrea: 8637678
```

Ciò potrebbe essere fatto anche passando la tabella come argomenti chiave con la notazione '**'.

```
>>> tabella = {'Alfredo': 4127, 'Gianni': 4098, 'Andrea': 8637678}
>>> print('Gianni: {Gianni:d}; Alfredo: {Alfredo:d}; Andrea:
{Andrea:d}'.format(**table))
Gianni: 4098; Alfredo: 4127; Andrea: 8637678
```

Ciò è particolarmente utile in combinazione con la funzione precostituita `vars()`, che restituisce un dizionario contenente tutte le variabili locali.

Come si è potuto notare, Python possiede una capacità di formattazione dell'output veramente notevole che permette buoni risultati estetici con poca fatica. Tutto questo è utile per formattazioni su schermo a linea di comando oppure per la scrittura su file, ma perdere un po' di valore quando Python viene utilizzato con interfacce grafiche le quali si occupano in prima persona della formattazione dei vari campi.

Per una panoramica completa sulla formattazione delle stringhe usando `str.format()` , fare riferimento a *Format String Syntax*.

7.1.1. Formattazione Stringhe Classica.

L'operatore `%` può essere utilizzato anch'esso per la formattazione di una stringa. Esso interpreta l'argomento di sinistra come la stringa di formato `sprintf()`, e restituisce la stringa formattata da questa operazione. Ma vediamolo meglio con un' esempio:

```
>>> import math
>>> print('Il valore approssimato di PI è %5.3f.' % math.pi)
Il valore approssimato di PI è 3.142.
```

Ulteriori informazioni possono essere trovate nella sezione *printf-style String Formatting* .

7.2. Lettura e Scrittura File.

`open()` restituisce un oggetto di tipo file (*file object*), ed è comunemente usato usando due argomenti: `open(nome_del_file, modalita)`:

```
>>> f = open('file_interessato', 'w')
```

Il primo argomento è una stringa contenente il nome del file da utilizzare ed il suo eventuale percorso. Il secondo argomento è un'altra stringa contenente alcuni caratteri che descrivono il modo in cui verrà

utilizzato il file. Per cui `modalita` può essere:

- `'r'-` Il file sarà aperto in modalità testo solo in lettura. Questa è la modalità predefinita se nessuna modalità viene specificata.
- `'w'-` Il file sarà aperto nella sola modalità testo solo in scrittura. Se esiste un file con lo stesso nome, esso sarà sovrascritto.
- `'a'-` Il file sarà aperto in modalità testo aggiungendo alla fine tutto ciò che verrà scritto.
- `'r+'-` Il file sarà aperto in modalità lettura/scrittura.
- `'modalita' + 'b'-` Se aggiunta alle altre modalità descritte, l'opzione `'b'`, aprirà il file in modalità binaria, in questo caso i dati vengono letti e scritti sotto forma di oggetti byte. Questa modalità dovrebbe essere utilizzata per tutti i file che non contengono testo.

Siccome il marcatore di fine file cambia in base alla piattaforma (**\n** su Unix/Linux **\r\n** su Windows) Python apre i file in lettura e usa come marcatore di fine file quello di Unix/Linux cioè **\n** . Quando il file invece viene scritto, Python converte il carattere di fine file nella piattaforma specifica in modo automatico. Quindi se un file viene letto e scritto sotto Windows quando verrà chiuso il terminatore di file ritornerà **\r\n** . Questo modo di gestire il terminatore di file garantisce la portabilità sulle piattaforme ma pone un potenziale rischio e cioè: il rischio di corrompere file di tipo non testo cioè `JPEG, EXE` ed altri di tipo binario per esempio. State molto attenti a utilizzare la modalità binaria durante la lettura e la scrittura di questi file.

7.2.1. Metodi dell' Oggetto File.

Gli esempi che proporremo in questa sezione presuppongono che un oggetto file chiamato **f** sia già stato creato.

Per leggere il contenuto di un file, chiamare `f.read((opz)dimensione)`, che legge una certa quantità di dati e li restituisce come una stringa o un oggetto byte. La dimensione è un argomento numerico facoltativo. Se `dimensione` viene omesso o è negativa, l'intero contenuto del file verrà letto e restituito, se `dimensione` viene specificata allora viene restituita quella specifica quantità . E' un problema del programmatore gestire il fatto che un file possa essere più grande della memoria disponibile sul computer. Se è stata raggiunta la fine del file, `f.read()` restituisce una stringa vuota (''').

```
>>> f.read()
'Questo è l'intero file.\n'
>>> f.read()
''
```

L'istruzione `f.readline()`, legge una singola riga del file comprensiva del carattere (**\n**) che viene lasciato alla fine della stringa, e viene omesso solo nell'ultima riga del file se il file non termina con un ritorno a capo. Questo disambigua il valore restituito, infatti se `f.readline()` restituisce una stringa vuota, significa che la fine del file è stata raggiunta, mentre se è una riga bianca è rappresentata da '**\n**', cioè una stringa contenente un solo '**\n**'.

```
>>> f.readline()
'Questa è la prima linea del file.\n'
```

```
>>> f.readline()
'Seconda linea del file\n'
>>> f.readline()
''
```

Per leggere delle linee da un file, si può eseguire un ciclo sull'oggetto file. Questa tecnica è veloce utilizza in mode efficiente la memoria e richiede poco codice esempio:

```
>>> for line in f:
...     print(line, end='')
...
Questa è la prima linea del file.
Seconda linea del file
```

Se volete leggere tutte le linee del file in una lista potete usare anche `list(f)` o `f.readlines()`.

La funzione `f.write(stringa)` scrive tutto il contenuto di `stringa` sul file e restituisce il numero di caratteri scritto.

```
>>> f.write('Questo è un test\n')
16
```

Notare che il valore restituito, non è comprensivo del marcatore di fine linea. Per scrivere qualcosa di diverso da una stringa, deve essere prima convertito in una stringa:

```
>>> valore = ('una risposta', 42)
>>> s = str(valore)
>>> f.write(s)
20
```

Notare che il valore restituito, è comprensivo di tutto anche delle parentesi tonde. `f.tell()` restituisce un intero che fornisce il numero di byte dell'oggetto file misurati dall'inizio del file.

Per variare la posizione dell'oggetto file si usi `f.seek(offset, da_cosa)`. La posizione viene calcolata aggiungendo ad offset un punto di riferimento, selezionato tramite l'argomento `da_cosa`. Un valore di `da_cosa` pari a `0` effettua la misura dall'inizio del file, `1` utilizza come punto di riferimento la posizione attuale, `2` usa la fine del file. `da_cosa` può essere omesso ed il suo valore predefinito è pari a `0`, viene quindi usato come punto di riferimento l'inizio del file.

```
>>> f = open('mio_file', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5)     # Va al 6° byte del file.
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Va al 3° byte prima della fine.
13
>>> f.read(1)
b'd'
```

I file di testo (quelli aperti, senza la `b` nella stringa di modalità) ammettono solo ricerche relative dall'inizio del file (ad eccezione di cercare dalla fine di file con `seek(0, 2)`), e gli unici valori validi sono quelli restituiti da `f.tell()`. Qualsiasi altro valore di scostamento produce un comportamento indefinito. Quando si finisce di lavorare con un file, bisogna chiamare `f.close()` per chiudere il file e scrivere i dati eventualmente presenti nel buffer di memoria su disco, e liberare le risorse. Ogni successivo tentativo di lavorare con esso genererà un'eccezione.

```
>>> f.close()
>>> f.read()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ValueError: I/O operation on closed file
```

Quando si lavora con i file, un buon sistema è l'utilizzo del metodo `with`. Questo ha il vantaggio che il file verrà chiuso correttamente dopo aver finito il suo lavoro anche nel caso in cui nel frattempo si generi un'eccezione. La scrittura del suo codice inoltre è molto più compatta rispetto all'equivalente del blocco `try-finally`.

```
>>> with open('file_lavoro', 'r') as f:
...     dati_letti = f.read()
>>> f.closed
True
```

Gli oggetti file hanno alcuni metodi addizionali, come `isatty()` e `truncate()`, che sono utilizzati meno frequentemente, consultare la Reference Library per una guida completa agli oggetti file.

7.2.2. Salvare strutture di dati con `json`.

Da un file, possono essere lette o scritte delle stringhe. Ottenere dei numeri invece richiede qualche sforzo in più. Dal momento che il metodo `read()` restituisce solo stringhe, per ottenere dei numeri esse dovranno essere passate ad una funzione tipo `int()` , tanto per fare un' esempio prendendo una stringa tipo '123' la funzione `int()` restituisce il relativo valore numerico 123. Se si desidera salvare tipi di dati più complessi, come le liste nidificate e i dizionari, analizzare e serializzare a mano i dati diventa assai complicato.

Al fine di evitare che molti programmatori siano costantemente impegnati nella scrittura di codice per risolvere la questione con inevitabile ricorso al debug, Python consente di utilizzare il popolare formato di interscambio dati chiamato `JSON (JavaScript Object Notation)`. Questo modulo standard può prendere le gerarchie di dati Python, e convertirle in rappresentazioni sotto forma di stringa, questo processo è chiamato serializzazione. Ricostruire i dati dalla rappresentazione dal formato stringa è chiamato deserializzazione. Tra serializzazione e deserializzazione, la stringa che rappresenta l'oggetto può essere memorizzata in un file di dati, o inviati tramite una connessione di rete ad una macchina remota.

Note:

Il formato JSON è oggi comunemente usato nelle moderne applicazioni che permettono l'interscambio di dati. Alcuni programmatori, hanno già familiarizzato con questo formato, facendone un'ottima scelta per l'interoperabilità.

Se avete un' oggetto X per esempio, potete vedere in una stringa il contenuto con una semplice linea di codice:

```
>>> json.dumps([1, 'semplice', 'lista'])
'[1, "semplice", "lista"]'
```

Una variante della funzione `dumps()` chiamata `dump()`, semplicemente serializza l'oggetto codificandolo in un file di testo(*text file*). Quindi se **f** è un file oggetto di testo, aprendolo in scrittura è possibile fare questo:

```
json.dump(x, f)
```

Per decodificare nuovamente l'oggetto, se `f` è un oggetto file di testo che è stato aperto per la lettura:

```
x = json.load(f)
```

Questa semplice tecnica di serializzazione è in grado di gestire liste e dizionari, ma la serializzazione di istanze di classi arbitrarie in JSON richiede uno sforzo in più. Riferimenti al modulo `json` contengono una spiegazione a questo.

Il modulo `pickle` , contrariamente a JSON, è un protocollo che consente la serializzazione di oggetti Python arbitrariamente complessi. Come tale, è specifico solo per Python e non può essere utilizzato per comunicare con applicazioni scritte in altri linguaggi. E 'anche insicuro per definizione! Deserializzare dati **pickle** provenienti da una fonte non attendibile può eseguire pericolosamente codice arbitrario, se i dati sono stati realizzati da un cracker esperto.

8. Errori ed Eccezioni.

Fino ad ora i messaggi di errore sono stati solo accennati, ma se avete provato gli esempi probabilmente ne avrete fatti alcuni. Ci sono (almeno) due tipi di errori: errori di sintassi e di logica.

8.1. Errori di sintassi.

Gli errori di sintassi, noti anche come errori di analisi, sono forse il tipo più comune commessi quando si sta imparando un linguaggio di programmazione. Python non fa eccezione a questa regola.

```
>>> while True print('Ciao mondo')
                    ^
SyntaxError: invalid syntax
```

Il **parser** cioè l'analizzatore del codice introdotto, mostra la linea incriminata e con una freccia cerca di puntare all'errore rilevato. L'errore è causato (o quantomeno rilevato in) dal pezzo che precede la freccia nell'esempio, in questo caso si riferisce non alla funzione `print()`, ma ai due punti (`':'`) mancanti prima di essa. L'esecuzione del codice corretto di cui sopra, genera la stampa infinita di 'Ciao mondo' interrompibile solo con un **CTRL-C**.

8.2. Eccezioni.

Anche se una dichiarazione o un'espressione sono sintatticamente corrette, può avvenire un errore quando

si tenta di eseguirla. Gli errori rilevati durante l'esecuzione sono chiamati **eccezioni** e sono spesso errori di logica, ma potrebbero riferirsi anche all'esaurimento della memoria o lo spazio su disco ecc. A volte sono fatali terminando senza troppi complimenti il programma o addirittura bloccare il calcolatore. Fortunatamente impareremo a gestirli al meglio. Tuttavia la maggior parte delle eccezioni sono gestite dal programma e possono produrre messaggi di errore come illustrato di seguito:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: Can't convert 'int' object to str implicitly
```

L'ultima riga del messaggio di errore indica che cosa è accaduto. Le eccezioni sono di diversi tipi, e il tipo viene stampato come parte del messaggio: i tipi nell'esempio sono `ZeroDivisionError`, `NameError` e `TypeError`. Questo è vero per tutte le eccezioni precostituite, ma non lo è per le eccezioni definite dall'utente (anche se è una convenzione utile). I nomi delle eccezioni standard sono identificatori predefiniti. Non sono parole riservate. Il resto della linea fornisce dettagli in base al tipo di eccezione e che cosa l'ha causata.

La parte antecedente del messaggio di errore mostra il contesto in cui è avvenuta l'eccezione, nella forma di una traccia dello stack. In generale, esso contiene una traccia che elenca le linee del sorgente, tuttavia, non visualizzerà le linee lette dallo standard input.

Note:
Built-in Exceptions elenca le eccezioni precostituite e il loro significato.

8.3. Trattamento delle Eccezioni.

E 'possibile scrivere programmi che gestiscono le eccezioni selezionate. Il seguente esempio, continua a chiedere all'utente di immettere un valore fino a che esso sia un' intero valido, ma permette all'utente di interrompere il programma (usando **CTRL-C** o qualunque altra combinazione di tasti che il sistema operativo ospite supporta), notare che in questo caso, un'interruzione generata dall'utente viene segnalata sollevando l'eccezione `KeyboardInterrupt`.

```
>>> while True:
...     try:
```

```
...         x = int(input("Immetti un numero intero: "))
...         break
...     except ValueError:
...         print("Uhm! Questo non è un numero valido. Ritenta...")
...
```

L'istruzione `try` lavora come segue.

- 1°: `try` è facente parte del blocco `try` e `except`.
- 2°: Se non si verificano errori la clausola `except` non viene eseguita e il programma prosegue oltre terminando il test di `try` .
- 3°: Se un'eccezione viene sollevata durante l'esecuzione di try, le rimanenti istruzione nel blocco vengono saltate. Se l'eccezione coincide con `except`, allora except viene eseguita e poi l'esecuzione continua dopo l'istruzione `try`.

- 4°: Se si verifica un'eccezione che non corrisponde a quella citata nella clausola except, si passa a istruzioni try più esterne, cioè per esempio alla funzione di un blocco chiamante e se non viene trovato alcun gestore, la gestione passa a Python e l'esecuzione si ferma visualizzando un messaggio come mostrato sopra terminando il programma.

Un'istruzione try può avere più di una clausola except, per specificare più gestori per diverse eccezioni, ma solo uno di essi verrà eseguito. I gestori si occupano solo delle eccezioni che si verificano nella clausola try corrispondente, non in altri gestori della stessa istruzione try. Una clausola di eccezione può nominare più di un'eccezione come fosse un tupla tra parentesi. Per esempio:

```
... except (RuntimeError, TypeError, NameError):
...     pass
```

Nell'ultima clausola except si può omettere il nome o i nomi utilizzandola come come caratteri jolly nel contesto. Questa pratica, è da utilizzare con estrema cautela in quanto è facile mascherare un errore di programmazione vero e proprio in questo modo! Può anche essere utilizzato per stampare un messaggio di errore e quindi risollevare l'eccezione (permettendo ad una procedura chiamante di gestire l'eccezione):

Una buona pratica di programmazione anche se un po' tediosa, è quella di creare un file di Log su cui scrivere il tipo di errore avvenuto e in che parte del programma. Questo permette la rapida correzione dello stesso su programmi distribuiti.

```
import sys

try:
    f = open('mio_file.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as err:
    print("errore I/O: {0}".format(err))
except ValueError:
    print("Impossibile convertire il dato in un intero.")
```

```
except:
    print("Errore inaspettato:", sys.exc_info()[0])
    raise
```

Con il blocco `try` ... `except` , è possibile utilizzare anche la clausola opzionale else, che, quando presente, deve seguire la clausola except. E' utile quando il codice che deve essere eseguito non è soggetto ad una nuova eccezione sollevata da try. Per esempio:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print('impossibile aprire.', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

L'uso della clausola else è preferibile all'aggiunta di codice supplementare alla clausola try poiché evita l'intercettazione accidentale di un'eccezione che non è stata sollevata dal codice protetto dal blocco try ... except.

Un'eccezione, può avere un valore associato, conosciuto anche come *argomento dell'eccezione*. La presenza e il tipo dell'argomento dipendono dal tipo di eccezione.

La clausola except, può specificare una variabile dopo il suo nome. Questa variabile è legata all'istanza dell'eccezione con argomenti memorizzati in `instance.args`. Per comodità, l'eccezione definisce `__str__()` in modo che gli argomenti possono essere stampati direttamente senza dover fare riferimento ad `.args`. Se si desidera, è anche possibile lanciare un'eccezione con l'istruzione raise aggiungendo gli attributi desiderati.

```
>>> try:
...     raise Exception('fagioli', 'uova')
...     except Exception as tipo:
...     print(type(tipo))    # L'istanza dell'eccezione.
...     print(tipo.args)     # Gli argomenti contenuti in .args.
...     print(tipo)          # __str__ permette la stampa di args.
...                          # ma può essere sovrascritto in exception.
...     x, y = tipo.args     # scompatta args.
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('fagioli', 'uova')
('fagioli', 'uova')
x = pancetta
y = uova
```

Se un'eccezione ha argomenti, questi vengono stampati come ultima parte ('il dettaglio') del messaggio per le eccezioni non gestite.

I gestori delle eccezioni non si occupano solo delle eccezioni gestite dal blocco try ... except, ma anche di tutti quegli errori che si possono verificare in qualsiasi parte del programma in quanto è Python che utilizzando il blocco try ... except al suo interno, li gestisce al meglio anche in quelle funzioni che vengono chiamate indirettamente. Per esempio:

```
>>> def questa_fallisce():
...     x = 1/0
...
>>> try:
...     questa_fallisce()
... except ZeroDivisionError as errore:
...     print('Gestione errore di esecuzione:', errore)
...
Gestione errore di esecuzione: int division or modulo by zero
```

8.4. Lancio delle Eccezioni.

L'istruzione `raise` , permette al programmatore di forzare all'occorrenza un'eccezione specifica. Per esempio:

```
>>> raise NameError('Questo errore')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: Questo errore
```

L'unico argomento che `raise` ha, indica l'eccezione da sollevare. Questo deve essere un'istanza di exception o una classe exception (cioè una classe che deriva da `Exception`).

Se avete bisogno di determinare se un eccezione è stata sollevata, ma non avete intenzione di gestirla, una forma semplice della dichiarazione `raise` permette di risollevare l'eccezione:

```
>>> try:
...     raise NameError('Questo errore')
... except NameError:
...     print('Un'eccezione si è sollevata!')
...     raise
...
Un'eccezione si è sollevata!
Traceback (most recent call last):
  File "<stdin>", line 2, in ?
NameError:  Questo errore
```

8.5.Eccezioni Definite Dall'Utente.

I programmatori possono creare le proprie eccezioni definendo una nuova classe di eccezione (vedere

Classes per saperne di più sulle classi Python). Le eccezioni dovrebbero normalmente essere derivate dalla classe `Exception`, direttamente o indirettamente. Per esempio:

```
>>> class MioErrore(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MioErrore(2*2)
... except MioErrore as e:
...     print('La mia eccezione, valore:', e.value)
...
La mia eccezione, valore: 4
>>> raise MioErrore('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
__main__.MioErrore: 'oops!'
```

In questo esempio, i valori predefiniti di `__init__()` e `Exception` sono stati sovrascritti, per cui il nuovo comportamento crea semplicemente l'attributo value. Questo sostituisce il comportamento predefinito di args.

Classi di eccezioni possono essere definite come si può fare con una qualsiasi altra classe, ma solitamente esse vengono mantenute semplici. Spesso esse offrono solo una serie di attributi che permettono di conoscere la natura dell'errore. Quando si crea un modulo che può sollevare diversi errori distinti, una pratica comune è quello di creare una classe base per le eccezioni definite da quel modulo, e una sottoclasse per creare specifiche classi di eccezioni per condizioni di errore diverse.

```
class Error(Exception):
    """Classe base per eccezioni in questo modulo."""
    pass

class InputError(Error):
    """Eccezione lanciata su errore di input.

    Attributi:
        espressione -- input espressione di quale errore è avvenuto.
        Messaggio   -- Spiegazione dell'errore.
    """

    def __init__(self, espressione, messaggio):
        self.espressione = espressione
        self.messaggio = messaggio

class ErroreTransizione(Error):
    """Lanciata quando un'operazione tenta una transizione che non è
permessa.
```

```
    Attributi:
        precedente – Stato all'inizio della transizione.
        successiva – tentativo nuovo stato.
        messaggio -- Spiegazione perché la specifica non è consentita.
    """

    def __init__(self, precedente, successiva, messaggio):
        self. precedente =  precedente
        self. successiva =  successiva
        self. messaggio =  messaggio
```

La maggior parte delle eccezioni vengono definite con nomi che terminano in "**Error**" simile alla denominazione delle eccezioni standard, questo facilita e accellera la comprensione.

Molti moduli standard e non, definiscono le proprie eccezioni per riportare errori che possono verificarsi nelle funzioni che definiscono. Ulteriori informazioni sulle classi saranno presentate nelle capitolo *Classi* .

8.6. Definizione dell'Azione di Pulizia (Clean-up).

L'istruzione `try` ha un'altra clausola facoltativa che mira a definire azioni di pulizia che deve esserci in tutte le circostanze. Per esempio:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print('Arrivederci mondo!')
...
Arrivederci mondo!
KeyboardInterrupt
```

Una clausola `finally` viene sempre eseguita prima di lasciare l'istruzione `try`, sia che si sia verificata o no un'eccezione. Se si verifica un'eccezione nella clausola `try` e nessuna gestione è stata prevista con `except`, (o si è verificato in `except` o `else`), allora viene eseguita l'istruzione `finally`. La clausola `finally` viene eseguita anche all'uscita del blocco quando un'altra clausola di `try` viene lasciata tramite `break`, `continue` o `return`. Un esempio più complesso:

```
>>> def divide(x, y):
...     try:
...         risultato = x / y
...     except ZeroDivisionError:
...         print("divisione per zero!")
...     else:
...         print("il risultato è", risultato)
...     finally:
...         print("esegue la clausola finally")
...
>>> divide(2, 1)
il risultato è 2.0
esegue la clausola finally
```

```
>>> divide(2, 0)
divisione per zero!
esegue la clausola finally
>>> divide("2", "1")
esegue la clausola finally
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Come potete vedere, la clausola `finally` viene eseguita in ogni caso. Il `TypeError` sollevato dividendo due stringhe non viene gestito dalla clausola `except` e quindi viene rilanciato dopo che la clausola `finally` è stata eseguita.

Nelle applicazioni del mondo reale, la clausola `finally` è utile per liberare risorse esterne (ad esempio file, connessioni di rete ecc.), indipendentemente dal fatto che l'uso della risorsa abbia avuto successo.

8.7. Azioni Predefinite di Pulizia (Clean-up).

Alcuni oggetti definiscono azioni di pulizia standard da intraprendere quando l'oggetto non è più necessario a prescindere dal fatto che l'oggetto sia stato utilizzato o meno. Guardate il seguente esempio, che cerca di aprire un file e stampare il contenuto sullo schermo.

```
for line in open("mio_file.txt"):
    print(line, end="")
```

Tutto bene, ma il problema con questo codice è che lascia il file aperto per un periodo di tempo indeterminato anche quando il codice ha terminato l'esecuzione. Questo non è un problema in semplici script, ma può essere un problema per le applicazioni più grandi o multiutente. Utilizzando invece al suo posto l'istruzione `with` la quale permette a oggetti come i file di essere utilizzati in un modo tale da garantire che siano sempre prontamente e correttamente rilasciati.

```
with open("mio_file.txt") as f:
    for line in f:
        print(line, end="")
```

Nell'esempio sopra, dopo che viene eseguita l'istruzione, il file **`f`** viene letto ed immediatamente chiuso anche se si verifica un problema durante l'elaborazione delle linee. Oggetti come i file, prevedono il clean-up nella loro documentazione.

9. Classi.

Premesso che l'argomento richiederebbe un trattato a se stante, vista la complessità, lo tratteremo qui in modo sintetico e peculiare a Python. Rispetto ad altri linguaggi di programmazione, il meccanismo delle classi di Python permette di aggiunge nuove classi con una sintassi e semantica minimi. Si tratta di una miscela di meccanismi per la manipolazione delle classi che si trovano in C++ e Modula-3 che forniscono Python di tutte le caratteristiche standard della **OOP** (*Object Oriented Programming*). Il meccanismo

dell'ereditarietà permette classi di base multiple, e una classe derivata può sovrascrivere tutti i metodi della sua classe o della classe da cui deriva, e un metodo può chiamare il metodo di una classe base con lo stesso nome. Gli oggetti possono contenere quantità arbitrarie di tipi e dati. Così come i moduli, le classi partecipano alla natura dinamica di Python: vengono create in fase di esecuzione, e possono essere ulteriormente modificate dopo la creazione.

Nella terminologia C++, normalmente i membri di classe (inclusi i membri dati) sono pubblici (tranne le variabili private (*Private Variables*) (vedi sotto)), e tutti membri delle funzioni sono virtuali. Le funzioni virtuali sono il meccanismo con cui si realizza in C++ il polimorfismo, che è una delle più importanti caratteristiche dei linguaggi orientati agli oggetti, che permette ad oggetti "simili" di rispondere in modo diverso agli stessi comandi. Come in Modula-3, non ci sono scorciatoie per riferirsi ai membri dell'oggetto dai suoi metodi. La funzione di metodo viene dichiarata con un' esplicito primo argomento che rappresenta l'oggetto, che viene fornito in modo implicito dalla chiamata. Come in Smalltalk, le classi stesse sono oggetti. Ciò fornisce una semantica per l'importazione e la ridenominazione. A differenza di C++ e Modula-3, tipi precostituiti possono essere utilizzati come classi base per essere estese dall'utente. Inoltre, sempre come in C + +, operatori precostituiti (operatori aritmetici, indicizzazioni, ecc), possono essere utilizzati con una sintassi speciale che può essere ridefinita per istanziare nuove classe.

La mancanza di una terminologia universalmente accettata per parlare di classi, ci porta a paragonare Python ad altri linguaggi per meglio comprendere ciò che viene detto. Questo non significa che Python non abbia pari potenzialità anzi! Questo si rende necessario perché la programmazione ad oggetti pur essendo oramai in uso da anni, rimane ostica ai più specialmente ai neofiti. Quindi paragoni ed esempi si rendono indispensabili.

9.1. Considerazioni Circa i Nomi e Gli Oggetti.

Gli oggetti hanno una propria vita, e più nomi (in più ambiti) possono essere associati allo stesso oggetto. Questo è noto come aliasing cioè la situazione che si verifica quando più simboli in un programma referenziano la stessa zona di memoria. Questo non è generalmente apprezzato di primo acchito da Python infatti può essere tranquillamente ignorato quando si tratta di tipi immutabili di base (numeri, stringhe, tuple). Tuttavia, l'aliasing ha un effetto forse sorprendente sulla semantica del codice Python che coinvolgono oggetti mutabili come liste, dizionari, e molti altri tipi. Questo viene usato a beneficio del programma dato che gli alias si comportano come puntatori per alcuni aspetti. Ad esempio, passando un oggetto è conveniente che solo un puntatore venga passato al momento dalla realizzazione, e se una funzione modifica un oggetto passato come argomento, il chiamante vedrà il cambiamento, questo elimina la necessità di due meccanismi diversi (per riferimento e per valore) per passare gli argomenti come avviene in Pascal .

9.2. Spazio dei Nomi e Visibilità in Python.

Prima di introdurre il meccanismo delle classi in Python, va detto qualcosa circa le regole di visibilità che Python usa. Le definizioni di classe giocano con alcuni trucchetti con gli spazi dei nomi, ed è necessario sapere come gli ambiti e gli spazi dei nomi lavorano per comprendere appieno cosa sta succedendo.

Cominciamo con alcune definizioni:

Un namespace (uno spazio dei nome) è una mappatura di nomi agli oggetti. La maggior parte degli spazi dei nomi vengono attualmente implementati come dizionari Python, ma che non sono normalmente visibili in alcun modo (tranne che per le prestazioni), e potrebbero cambiare in futuro. Esempi di spazi dei

nomi sono: l'insieme dei nomi precostituiti (contenenti funzioni come abs () e i nomi delle eccezioni precostituite), i nomi globali in un modulo, e i nomi locali in una funzione chiamata. In un certo senso l'insieme degli attributi di un oggetto contribuisce a formare uno spazio dei nomi. La cosa importante da sapere sugli spazi dei nomi è che non c'è assolutamente nessuna relazione tra i nomi in diversi spazi dei nomi, per esempio, due moduli diversi potrebbero entrambi definire una funzione con lo stesso nome senza confusione, questo è possibile anteponendo al nome della funzione il nome del modulo con la notazione del punto tipo mioModulo.miaFunzione.

A proposito sempre a causa di una mancanza di specifiche precise sulle classi, io uso la parola attributo per qualsiasi nome che segua un punto per esempio nel z.real, real è un attributo dell'oggetto z. A rigor di termini, i riferimenti a nomi nei moduli sono riferimenti ad attributi come già detto nel paragrafo precedente.
Gli attributi, possono essere a sola lettura o scrivibili. In quest'ultimo caso, l'assegnazione di attributi è possibile.

Gli attributi dei moduli sono scrivibili: si può scrivere `nomemodulo.risposta = 42`. Gli attributi scrivibili possono anche essere cancellati con l'istruzione `del`.
Ad esempio, `del nomemodulo.risposta` rimuoverà l' attributo `risposta` dall'oggetto nominato `nomemodulo.`

Gli spazi dei nomi vengono creati in momenti diversi e hanno diverse vite. Lo spazio dei nomi che contiene i nomi precostituiti viene creato quando l'interprete Python si avvia, e non viene **mai** eliminato. Lo spazio dei nomi globale riferiti a un modulo vengono creati quando la definizione del modulo viene letta; normalmente, gli spazi dei nomi del modulo durano fino a quando l'interprete si chiude. Le istruzioni eseguite dall'invocazione di livello superiore dell'interprete, lette da un file di script o interattivamente, sono considerate parte di un modulo chiamato `__main__`, in modo da avere il proprio spazio dei nomi globale. (I nomi precostituiti in realtà vivono anche in un modulo che si chiama `builtins`.)

Lo spazio dei nomi locale per una funzione viene creato quando la funzione viene chiamata, e cancellati quando la funzione restituisce il suo valore o solleva un'eccezione non gestita all'interno della funzione. (In realtà, la dimenticanza sarebbe uno modo sporco per descrivere meglio ciò che effettivamente accade.) Naturalmente, invocazioni ricorsive hanno ciascuna il proprio spazio dei nomi locale (memoria permettendo).

Un' ambito, è una visibilità che una regione testuale di un programma Python possiede dove uno spazio è accessibile direttamente. "Direttamente accessibile" qui significa che un riferimento qualificato ad un nome cerca di trovare il nome nello spazio dei nomi opportuno.

Anche se gli ambiti sono determinati staticamente cioè che non possono essere modificati, essi sono utilizzati in modo dinamico. In qualsiasi momento durante l'esecuzione ci sono almeno tre visibilità annidate cui i spazi sono direttamente accessibili:

- L'area più interna, è quella che viene ricercata per prima e contiene i nomi locali.
- La visibilità di tutte le funzioni viene ispezionata cercando nello spazio dei nomi più vicino contenente il simbolo cercato, fino ad espandere la ricerca verso simboli non locali, ma anche verso simboli non globali.
- l'ambito next-to-last contiene i nomi dei moduli globali attuali.
- l'ambito più esterno (l'ultimo) è lo spazio dei nomi che contiene i nomi precostituiti.

Ecco una rappresentazione grafica sullo spazio dei nomi:

Se un nome viene dichiarato globale, allora tutti i riferimenti e le assegnazioni vanno direttamente al campo centrale che contiene i nomi globali del modulo. Per ricollegare le variabili presenti al di fuori del proprio ambito, può essere utilizzata la dichiarazione `nonlocal`, se non dichiarate non locale, quelle variabili sono in sola lettura (un tentativo di scrivere su tale variabile sarà sufficiente per creare una nuova variabile locale nel campo di applicazione più interno, lasciando il nome identico della variabile esterna invariato).

Di solito, l'ambito locale fa riferimento a nomi locali della (testualmente) funzione corrente. Al di fuori delle funzioni, l'ambito locale fa riferimento allo stesso spazio dei nomi dell'ambito globale del modulo. Le definizioni di classe pone un altro spazio dei nomi nell'ambito locale.

E 'importante rendersi conto che gli ambiti sono determinati testualmente: la visibilità globale di una funzione definita in un modulo è lo spazio dei nomi di quel modulo, non importa da dove o da che alias la funzione viene chiamata. D'altra parte, la ricerca effettiva dei nomi viene fatta dinamicamente in fase di esecuzione tuttavia, la definizione del linguaggio si sta evolvendo verso la risoluzione dei nomi statici, in fase di "compilazione", quindi non si basano sulla risoluzione dei nomi dinamici! (In effetti, variabili locali sono già determinate staticamente.)

Un cavillo particolare di Python è che se nessuna dichiarazione globale (`global`) è in essere, le assegnazioni ai nomi vanno sempre nel perimetro più interno. Le assegnazioni non copiano dati, ma creano solo i nomi agli oggetti. Lo stesso vale per le cancellazioni: l'istruzione `del x` rimuove l'associazione di `x` dallo spazio dei nomi a cui fa riferimento l'ambito locale. In realtà tutte le operazioni

che introducono nuovi nomi usano l'ambito locale e in particolare, le dichiarazioni di importazione (`import`) e le definizioni di funzioni legano il modulo o nome di funzione all'ambito locale.

L'istruzione `global` può essere utilizzata per indicare che particolari variabili vivono in ambito globale e lì dovrebbero essere riferite, l'istruzione `nonlocal` indica che particolari variabili vivono in un ambito di inclusione e lì dovrebbero essere riferite.

9.2.1. Esempi di Ambiti e Spazio Dei Nomi.

Questo esempio, dimostra come il riferimento a diversi ambiti e spazi dei nomi e in che modo `global` e `nonlocal`, influenzano il collegamento alla variabile.

```
def test_visibilita():
    def do_local():
        spam = "local spam"
    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
    def do_global():
        global spam
        spam = "global spam"
    spam = "test spam"
    do_local()
    print("Dopo l'assegnamento locale:", spam)
    do_nonlocal()
    print("Dopo l'assegnamento non locale:", spam)
    do_global()
    print("Dopo l'assegnamento globale:", spam)

test_visibilita()
print("In visibilità globale:", spam)
Dopo l'assegnamento locale: test spam
Dopo l'assegnamento non locale: nonlocal spam
Dopo l'assegnamento globale: nonlocal spam
In visibilità globale: global spam
```

Si noti come l'assegnazione local (che è di default) non ha cambiato il legame di spam su test_visibilita. L'assegnazione `nonlocal` cambia il legame di spam su test_visibilita, e l'assegnazione `global` ha cambiato il legame a livello di modulo.

Si può anche vedere che non c'era nessun precedente vincolante per spam prima dell'assegnamento `global`.

9.3. Un Primo Sguardo alle Classi.

Le classi introducono un po' di sintassi nuova, tre nuovi tipi di oggetti e nuova semantica.

9.3.1. Sintassi nella Definizione di Classe.

La forma più semplice di definizione di una classe è questo:

```
class NomeClasse:
    <istruzione-1>
    .
    .
    .
    <istruzione-n>
```

Le definizioni di classe, similarmente alle definizioni di funzione (istruzione `def`) devono essere eseguite prima di avere alcun effetto. (Si potrebbe concettualmente, inserire una definizione di classe in un ramo di un'istruzione `if`, o all'interno di una funzione.)

In pratica, le dichiarazioni all'interno di una definizione di classe saranno di solito le definizioni di funzioni, ma altre dichiarazioni sono consentite e talvolta utili, torneremo più avanti su questo tema. Le definizioni di funzione all'interno di una classe normalmente hanno una peculiare forma di elenco di argomenti, dettata dalle convenzioni di chiamata per i metodi anche questo sarà spiegato più avanti.

Quando si accede in una nuova definizione di classe, un nuovo spazio dei nomi viene creato e utilizzato come l'ambito locale, pertanto tutte le assegnazioni a variabili locali finiscono in questo nuovo spazio dei nomi. In particolare, le definizioni di funzione legano qui il nome della nuova funzione.
Quando una definizione di classe esiste, viene creato un oggetto di classe. Questo è fondamentalmente un a locazione di memoria che si può paragonare ad un guscio variabile al cui interno, vengono creati gli spazi dei nomi definiti con la classe. Nelle sezioni successive, impareremo di più su oggetti e classi. L'ambito locale originale (quello in vigore poco prima che la definizione di classe fosse inserita) viene ridefinito, e l'oggetto della classe è tenuto qui al suo interno, e il nome della classe indicata nell'intestazione della definizione di classe (`NomeClasse` nell'esempio).

9.3.2. Oggetti Classe.

Gli oggetti classe supportano due tipi di operazioni: i riferimenti ad attributi e l' istanziazione.

- Attributi riferimenti: Utilizzano la sintassi standard utilizzata per tutti i riferimenti attributi in Python; `oggetto.nome`. Per nomi di attributi validi s'intende tutti i nomi che sono nello spazio dei nomi della classe quando l'oggetto classe è stato creato. Quindi, se la definizione di classe si presentava così:

  ```
  class MiaClasse:
  """Un semplice esempio di classe"""
  i = 12345
  def f(self):
  return 'Ciao mondo'
  ```

  Allora `MiaClasse.i` e `MiaClasse.f` sono riferimenti validi ai suoi attributi, che restituiscono rispettivamente un intero ed un oggetto funzione. Gli attributi di classe possono inoltre essere assegnati in modo da poter cambiare il valore di `MiaClasse.i` per assegnamento. `__doc__` è anch'esso un attributo valido, restituendo la stringa di documentazione appartenente alla classe, cioè "Un semplice esempio di classe".

- L'istanzazione di una classe, viene fatta con la notazione delle funzioni. Basta far finta che l'oggetto di classe è una funzione senza parametri per avere una nuova istanza della classe. Ad esempio (supponendo la precedente classe ad esempio):

```
x = MiaClasse()
```

Crea una nuova istanza della classe assegnata all'oggetto **x** in una variabile locale.

L'operazione di istanziazione crea un oggetto vuoto. Molte classi creano oggetti personalizzati per uno specifico stato iniziale. Una classe infine può definire un metodo speciale chiamato`__init__()` come questo:

```
def __init__(self):
    self.data = []
```

Quando una classe viene definita con il metodo `__init__()`, l'istanziazione della classe, invoca automaticamente `__init__()` per la nuova istanza di classe creata. Per cui in questo esempio, una nuova istanziazione di classe può essere ottenuta così:

```
x = MiaClasse()
```

Naturalmente, il metodo `__init__()` può avere argomenti, per una maggiore flessibilità. In tal caso, gli argomenti forniti alla classe vengono passati a`__init__()`. Per esempio:

```
>>> class Complessa:
...     def __init__(self, partereale, parteimmaginaria):
...         self.r = partreale
...         self.i = partimmaginaria
...
>>> x = Complessa(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3. Instanze di Oggetti.

Ora cosa possiamo fare con un'istanza di oggetti? Le uniche operazioni conosciute dagli oggetti istanza sono i riferimenti ai suoi attributi. Ci sono due tipi di attributi validi, gli attributi dei **dati** e gli attributi dei **metodi**. Per capire meglio la differenza fra le due entità e poterli distinguere al volo, basta ricordarsi che:

- I **Dati** corrispondono a delle proprietà e possono essere associati al verbo **ESSERE**.
- I **metodi** corrispondono a delle elaborazioni e possono essere associati al verbo **FARE**.

Quindi:

Gli attributi di dati, corrispondono alle "variabili d'istanza" in Smalltalk, e ai "dati membro" in C + +. Gli attributi di dati non necessitano di essere dichiarati come le variabili locali, essi cominciano ad esistere quando viene fatta la prima assegnazione. Ad esempio, se **x** è l'istanza di `MiaClasse` creata in precedenza, il seguente pezzo di codice stamperà il valore `16`, senza lasciare traccia (notare che **x** è un valore e quindi va associata al verbo **ESSERE**):

```
x.contatore = 1
while x.contatore < 10:
    x.contatore = x.contatore * 2
print(x.contatore)
del x.contatore
```

L'altro tipo d'attributo d'istanza si riferisce ai metodi. Un metodo è una funzione che "appartiene a" un oggetto. (In Python, il termine metodo non è unico a istanze di classi: altri tipi di oggetti possono avere dei metodi, ad esempio, oggetti lista hanno metodi chiamati `append`, `insert`, `remove`, `sort` (notare che tutti fanno qualcosa e quindi vanno associati al verbo **FARE**) e così via, detto ciò, nella discussione che segue, useremo il termine metodo esclusivamente per indicare i metodi degli oggetti d'istanza di classe, se non diversamente specificato.)

Nomi validi per metodi di un'istanza d'oggetto, dipendono dalla sua classe. Per definizione, tutti gli attributi di una classe che sono oggetti funzionali definiscono metodi corrispondenti alle sue istanze. Quindi, nel nostro esempio, **`x.f`** è un riferimento valido ad un metodo, dal momento che `MiaClasse.f` è una funzione, ma **`x.i`** non è, in quanto `MiaClasse.i` non lo è. Ma **`x.f`** non è la stessa cosa di `MiaClasse.f` , è un oggetto **metodo**, non un oggetto **funzione**.

9.3.4. Metodi degli Oggetti.

Di solito, un metodo viene chiamato subito dopo che è stato creato:

```
x.f()
```

Nell'esempio `MiaClasse`, questo restituirà la stringa '**`Ciao mondo`**'. Tuttavia, non è necessario chiamare un metodo in modo immediato: `x.f` è un oggetto metodo, e può essere riposto e chiamato in un secondo momento. Per esempio:

```
xf = x.f
while True:
    print(xf())
```

Che continuerà a stampare '**`Ciao mondo`**' all'infinito.

Che cosa succede esattamente quando un metodo viene chiamato? Avrete notato che `x.f()` di cui sopra è stato chiamato senza argomenti, anche se la definizione della funzione di `f()` specifica un argomento. Che fine ha fatto l'argomento? Sicuramente Python solleva un'eccezione quando una funzione che richiede un argomento viene chiamata senza, anche se l'argomento non viene effettivamente utilizzato …

In realtà, ci si può già immaginare la risposta: la particolarità dei metodi è che l'oggetto viene passato come primo argomento della funzione. Nel nostro esempio, la chiamata `x.f()` è esattamente equivalente a `MiaClasse.f(x)`. In generale, chiamando un metodo con una lista di *n* argomenti è equivalente a chiamare la corrispondente funzione con una lista di argomenti creata inserendo i metodi dell'oggetto prima che il primo argomento.

Se non si capisce ancora come i metodi lavorino, uno sguardo alle loro implementazioni può forse servire a chiarire le cose. Quando viene fatto riferimento ad un attributo di istanza che non è un attributo di dati, viene ricercata la sua classe. Quando un metodo dell'oggetto viene chiamato con una lista di argomenti, una nuova lista di argomenti viene creata nell'istanza dell'oggetto e la funzione viene chiamata con questa

nuova lista di argomenti.

9.4. Note Casuali.

I dati con lo stesso nome dei metodi, prevalgono su di essi per evitare conflitti accidentali, che possono causare malfunzionamenti difficili da trovare specialmente in programmi di grandi dimensioni, è quindi consigliabile utilizzare una sorta di convenzione che riduce al minimo il rischio di conflitti. Alcune convenzioni possibili, includono la capitalizzazione del nomi nei metodi, anteponendo ai nomi dati agli attributi con una piccola stringa univoca (per esempio un carattere di sottolineatura), oppure utilizzando verbi per i metodi e sostantivi per i dati.

Attributi di dati possono essere referenziati dai metodi come anche dagli utenti di un oggetto. In altre parole, le classi non sono utilizzabili per implementare tipi di dati astratti e puri. In realtà, nulla in Python permette l'offuscamento dei dati, tutto si basa unicamente sulle convenzioni. D'altra parte, l'implementazione di Python, scritta in C, può nascondere completamente i dettagli dell'implementazione e se necessario controllare l'accesso a un oggetto, che può essere utilizzato da estensioni a Python scritte in C.

Gli utenti, dovrebbero usare gli attributi dato con cura, essi possono rovinare invariabilmente dati conservati dai metodi sovrascrivendoli. Si noti che gli utenti possono aggiungere attributi dato propri ad un'istanza d'oggetto senza intaccare la validità dei metodi, purché vengano evitati conflitti con i nomi . Ancora una volta, una convenzione sulla denominazione dei nomi, può risparmiare un sacco di mal di testa.

Non esistono scorciatoie per referenziare attributi di dati (o altri metodi!) dall'interno dei metodi. Trovo che questo in realtà aumenta la leggibilità dei metodi: non c'è possibilità di confondere le variabili locali e le variabili di istanza guardando un metodo.

Spesso, il primo argomento di un metodo viene chiamato `self`. Questa non è altro che una convenzione: il nome `self` non ha assolutamente alcun significato speciale per Python. Si noti, tuttavia, che, non seguendo questa convenzione il codice potrebbe essere meno leggibile da altri programmatori Python, ed è anche concepibile che un programma scritto per visualizzare le classi per esempio, possa essere scritto in base a tali convenzioni.
Quello delle convenzioni, potrebbe sembrare a prima vista un problema di poco conto. Pensate a due persone che interloquiscono fra loro parlando la stessa lingua, se ad un certo punto uno dei due, sostituisce parole note con altre provenienti da un linguaggio sconosciuto all'altro, esso avrebbe quanto meno difficoltà nel comprendere il senso del discorso. Di contro Python non infarcisce il suo lessico con quantitativi di parole chiave come avviene in altri linguaggi, con i suoi pro e i suoi contro. Concludendo quindi, è meglio attenersi allo standard proposto da Python.

Qualsiasi funzione di un' oggetto essendo un' attributo di classe, definisce un metodo per le istanze della classe stessa. Non è necessario che la definizione della funzione sia inclusa nella definizione della classe. E' possibile anche assegnare una funzione a una variabile locale nella classe. Per esempio:

```
# Funzione definita al di fuori della classe.
def funzione_esterna(self, x, y):
    return min(x, x+y)
```

```
class C:
    f = funzione_esterna
    def g(self):
        return 'hello world'
    h = g
```

Ora **f**, **g** e **h** sono tutti attributi della classe **C** che si riferiscono ad oggetti funzione, e di conseguenza sono tutti metodi delle istanze di **C** essendo **h** esattamente equivalente a **g**. Questa pratica, viene di solito usata per rendere più difficile la comprensione di un sorgente ad altri programmatori.

I metodi possono chiamare altri metodi usando gli attributi metodo con l'argomento `self`:

```
class Bag:
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

I metodi possono referenziare nomi globali allo stesso modo come le funzioni ordinarie. La visibilità globale associata a un metodo è il modulo che contiene la sua definizione (Una classe non viene mai usata con visibilità globale). Mentre non ci sono dei buoni motivi per usare dati globali in un metodo, ci sono molti usi legittimi della visibilità globale: per dirne una, funzioni e moduli importati in ambito globale possono essere utilizzati dai metodi, nonché funzioni e classi definite in esso. Di solito, la classe che contiene il metodo ed essa stessa è definita in questo ambito globale. Nella prossima sezione vedremo alcune buone ragioni per cui un metodo potrebbe voler referenziare la propria classe.

Ogni valore è un oggetto, e quindi ha una classe (chiamato anche tipo) e viene memorizzato come `object.__class__`.

9.5. L'Ereditarietà.

Naturalmente, un linguaggio che supporta le classi senza ereditarietà non sarebbe degno di attenzione. La sintassi per la definizione di una classe derivata (l'ereditarietà appunto) si ottiene più o meno così:

```
class NomeClasseDerivata(NomeClasseBase):
    <istruzione-1>
    .
    .
    .
    <istruzione-n>
```

Il nome `NomeClasseBase` deve essere definito in un ambito che contiene la definizione della classe derivata. Al posto di un nome di classe base, sono ammesse anche altre espressioni arbitrarie. Ciò può essere utile, per esempio, quando la classe base è definita in un altro modulo:

classe NomeClasseDerivata (nomemodulo.NomeClasseBase)

Esecuzione di una definizione di classe derivata procede nello stesso modo di una classe base. Quando l'oggetto di classe viene costruito, la classe base viene ricordata. Questo meccanismo viene utilizzato per risolvere i riferimenti ad attributi: se un attributo richiesto non viene trovato nella classe, la ricerca procede fino a guardare nella classe base. Questa regola viene applicata ricorsivamente se la classe base a sua volta è derivata da un'altra classe.

Non c'è niente di speciale creazione di classi derivate: `NomeClasseDerivata()` crea una nuova istanza della classe. I riferimenti ai metodi vengono risolti come segue: l'attributo classe corrispondente viene cercato, salendo lungo la catena di classi base, se necessario, e il riferimento al metodo è valido se questo produce un oggetto funzione.

Le classi derivate possono sovrascrivere i metodi delle loro classi base. Poiché i metodi non hanno privilegi speciali quando chiamano altri metodi dello stesso oggetto, un metodo di una classe base che chiama un altro metodo definito nella stessa classe base può finire per chiamare un metodo di una classe derivata che lo sovrascrive. (Per i programmatori C + +: tutti i metodi in Python sono effettivamente `virtual` (virtuali)).

Generalmente la sovrascrittura di un metodo in una classe derivata, ha lo scopo di estenderne le funzionalità piuttosto che sostituirlo in toto. Infatti a rigore di logica se a un metodo manca una funzionalità, lo si sovrascrive aggiungendocela, se invece è diverso, se ne crea uno nuovo anche se non sempre è così. C'è un modo semplice per chiamare direttamente il metodo della classe base: basta chiamare `NomeClasseBase.nomemetodo(self, argomenti)`. Questo è talvolta risulta il suo utilizzo (va detto che questo funziona solo se la classe base è accessibile come `NomeClasseBase` in ambito globale.)

Python dispone di due funzioni precostituite che funzionano con l'ereditarietà:

- `isinstance()` utilizzata per verificare il tipo di istanza: `isinstance(oggetto, int)` sarà vero solo se `oggetto.__class__` è di tipo `int` o di qualche classe derivata da `int`.

- `issubclass()` utilizzata per verificare l'ereditarietà della classe: `issubclass(bool, int)` è `True` in quanto `bool` è una sottoclasse di `int`. Ma per esempio, `issubclass(float, int)` risulta `False` in quanto `float` non è una sottoclasse di `int`.

9.5.1. Ereditarietà Multipla.

Python supporta pure una forma di ereditarietà multipla. Una definizione di classe con più classi base che assomiglia a questo:

```
class NomeClasseDerivata(ClasseBase1, ClasseBase2, ClasseBase3):
    <istruzione-1>
    .
    .
    .
    <istruzione-n>
```

Per la maggior parte degli utilizzi, nei casi più semplici, si può pensare che la ricerca per gli attributi

ereditati da una classe genitore venga cercata scalando in profondità, da sinistra a destra, non cercando due volte nella stessa classe in cui vi è una sovrapposizione nella gerarchia. Così, se un attributo non viene trovato in `NomeClasseDerivata`, la ricerca avviene in `ClasseBase1`, poi (ricorsivamente) nelle classi base di `ClasseBase2`, e se non è stato trovato, si è cerca in `ClasseBase3`, e così via.

In verità, la cosa è leggermente più complessa. La risoluzione dell'ordine di un metodo cambia dinamicamente per supportare chiamate cooperative a `super()`. Questo approccio è noto in alcune altri linguaggi con ereditarietà multipla come **`call-next-method`** ed è più potente della chiamata `super` presente in altri linguaggi a ereditarietà singola.

L'ordinamento dinamico è necessario in quanto tutti i casi di ereditarietà multipla mostrano uno o più diamond relationships (cioè una specie di relazionamento superiore) in cui almeno una delle classi genitore può accedere attraverso percorsi multipli della classe più in basso. Ad esempio, tutte le classi ereditano da `object`, così che qualsiasi caso di ereditarietà multipla fornisca più di un percorso per arrivare all'oggetto. Per mantenere le classi di base da cui si accede più di una volta, l'algoritmo dinamico linearizza l'ordine di ricerca in modo da preservare l'ordinamento da sinistra a destra specificato in ogni classe, che chiama ciascun genitore solo una volta, e che è monotona (cioè una classe può essere sottoclasse senza influenzare l'ordine di precedenza dei suoi genitori). Nel loro insieme, queste caratteristiche rendono possibile progettare classi affidabili ed estensibile con ereditarietà multipla. Per una spiegazione più dettagliata e approfondita, vedere:
http://www.python.org/download/releases/2.3/mro/.

9.6. Variabili Private.

Le variabili d'istanza definite "Private" a cui non è possibile accedere se non da all'interno di un oggetto semplicemente non esistono in Python. Tuttavia, vi è una convenzione che viene seguita dalla maggior parte del codice Python: un nome preceduto da un carattere di sottolineatura (ad esempio _spam) dovrebbe essere trattato come una parte non pubblica delle API (se si tratta di una funzione, un metodo o un membro dati) . Dovrebbe essere considerato un dettaglio di implementazione e sono soggetti a modifiche senza preavviso.

Poiché esistono casi d'uso validi per i membri per classi-private (per evitare conflitti di nome con nomi definiti dalle sottoclassi), c'è un supporto limitato per un tale meccanismo, chiamato ***name mangling*** *(*alterazione dei nomi). Qualsiasi identificatore nella forma `__spam` (con almeno due sottolineature all'inizio e al massimo una sottolineatura finale) è testualmente sostituita con `_nomeclasse__spam`, dove `nomeclasse` è il nome della classe corrente con la sottolineatura tolta. Questa ricomposizione viene eseguita a prescindere dalla posizione sintattica dell'identificatore, fintanto che si verifica nella notazione di una classe.

La ricomposizione dei nomi è utile per fare in modo che sottoclassi sostituiscono i metodi senza rompere le chiamate interclasse di metodo. Per esempio:

```
def __init__(self, iterable):
        self.items_list = []
        self.__update(iterable)

    def update(self, iterable):
        for item in iterable:
            self.items_list.append(item)
```

```
    __update = update   # copia privata del metodo originale update()

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # fornisce una nuova firma per update()
        # ma non interrompe __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Si noti che le regole di ricomposizione sono pensate principalmente per evitare incidenti, è ancora possibile accedere o modificare una variabile considerata privata questo può anche rivrlarsi utile in circostanze particolari, come nel debugging.

Notare che il codice passato a `exec()` o `eval()` non considera il nome della classe chiamante come classe corrente, questo effetto è simile all'istruzione `global`, il cui effetto è ristretto in modo simile al codice byte che viene compilato assieme. La stessa limitazione si applica a `getattr()`, `setattr()` e `delattr()`, così come quando si fa riferimento direttamente a `__dict__`.

9.7. Varie.

A volte è utile avere un tipo di dati simile al "record" di Pascal o C tipo "struct", associare alcuni elementi di dati denominati. Una definizione di classe vuota lo farà bene:

```
class Impiegati:
    pass

Marco = Impiegati() # Crea un record vuoto di tipo impiegati.

# Riempie i campi del record.
Marco.Soprannome = 'Marco l'incredibile'
Marco.Qualifica = 'Tecnico computer'
Marco.Stipendio = 1000
```

Un pezzo di codice Python che prevede un particolare tipo di dato astratto può essere invece passato a una classe che emula i metodi di questo tipo di dati. Ad esempio, se si dispone di una funzione che formatta alcuni dati da un oggetto file, è possibile definire una classe con metodi `read()` e `readline()` che ottenga invece i dati da un buffer di stringa, e passarlo come argomento.

L'istanza di metodi di oggetto, possiede anche degli attributi tipo: `m.__self__` che è l'oggetto istanza con il metodo `m()`, e `m.__func__` è la funzione oggetto corrispondente al metodo.

9.8. Le Eccezioni Possono Essere Anche Classi.

Eccezioni definite dall'utente vengono identificate da classi pure. Utilizzando questo meccanismo è possibile creare gerarchie estendibili di eccezioni.

Ci sono due nuove forme (semantiche) valide per l'istruzione `raise`:

```
raise Class

raise Instance
```

Nella prima forma, `Class` deve essere un'istanza di `type` o di una classe derivata da essa. La prima forma è una scorciatoia per:

```
raise Class()
```

Una clausola `except` in una classe è compatibile con un'eccezione se si tratta della stessa classe o di una classe base della stessa (ma non viceversa, una clausola except che riporti una classe derivata non è compatibile con una classe base). Ad esempio, il codice seguente stamperà B, C, D in questo ordine:

```
class B(Exception):
    pass
class C(B):
    pass
class D(C):
    pass

for cls in [B, C, D]:
    try:
        raise cls()
    except D:
        print("D")
    except C:
        print("C")
    except B:
        print("B")
```

Si noti che se la clausola except fosse state invertita (con `except B` per prima), sarebbe stato stampato B, B, B - la prima clausola except verrebbe stata attivata.

Quando un messaggio di errore viene stampato per un'eccezione non gestita, prima viene stampato il nome della classe seguito da due punti, poi uno spazio, e infine l'istanza convertita in una stringa utilizzando la funzione precostituita `str()`.

9.9. Iteratori.

Ormai probabilmente avete notato che la maggior parte degli oggetti possono essere iterati utilizzando l'istruzione `for`:

```
for elemento in [1, 2, 3]:
    print(elemento)
for elemento in (1, 2, 3):
    print(elemento)
for chiave in {'uno':1, 'due':2}:
```

```
    print(chiave)
for stringa in "123":
    print(stringa)
for linee in open("miofile.txt"):
    print(linee)
```

Questo codice, è chiaro, conciso e conveniente. L'uso degli iteratori pervade e unifica Python. Dietro le quinte, l'istruzione `for` chiama `iter()` sull'oggetto contenitore. La funzione restituisce un oggetto iteratore che definisce il metodo `__next__()` che accede agli elementi nel contenitore uno alla volta. Quando non ci sono più elementi, `__next__()` solleva un'eccezione `StopIteration` che sancisce la fine del ciclo `for`. È possibile chiamare il metodo `__next__()` utilizzando la funzione precostituita `next()`, questo esempio mostra come funziona il tutto:

```
>>> s = 'abc'
>>> prova = iter(s)
>>> prova
<prova iterator object at 0x00A1DB50>
>>> next(prova)
'a'
>>> next(prova)
'b'
>>> next(prova)
'c'
>>> next(prova)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
    next(prova)
StopIteration
```

Dopo aver visto i meccanismi alla base del protocollo iteratore, è facile aggiungere un comportamento iteratore alle vostre classi. Definire un metodo `__iter__()`, che restituisce un oggetto con un metodo `__next__()`. Se la classe definisce `__next__()`, allora `__iter__()` può semplicemente restituire `self`:

```
    """Cicla su una sequenza rovesciata."""
    def __init__(self, data):
        self.data = data
        self.index = len(data)
    def __iter__(self):
        return self
    def __next__(self):
        if self.index == 0:
            raise StopIteration
        self.index = self.index - 1
        return self.data[self.index]
>>> rovescia = Rovescia('spam')
>>> iter(rovescia)
<__main__.Rovescia object at 0x00A1DB50>
>>> for carattere in rovescia:
...     print(carattere)
```

```
...
m
a
p
s
```

9.10. Generatori.

I generatori (*Generator*s) sono uno strumento semplice e potente per la creazione di iteratori. Sono scritti come funzioni regolari, ma usano l'istruzione `yield` ogni volta che devono restituire i dati. Ogni volta che `next()` viene chiamato su di esso, il generatore riprende da dove si era fermato (si ricorda tutti i valori dei dati e l'ultima espressione eseguita). Un esempio mostra che i generatori possono essere banalmente facili da creare:

```
def rovescia(dati):
    for indice in range(len(dati)-1, -1, -1):
        yield dati[indice]
>>> for carattere in rovescia('golf'):
...     print(carattere)
...
f
l
o
g
```

Tutto ciò che può essere fatto con i generatori può essere fatto anche con classi base iteratori come descritto nella sezione precedente. Ciò che rende i generatori così compatti è che i metodi `__iter__()` e `__next__()` , vengono creati automaticamente.

Un'altra caratteristica fondamentale è che le variabili locali ed il loro stato di esecuzione vengono salvati automaticamente tra le chiamate. Questo ha reso la funzione più facile da scrivere e molto più chiara che con un' approccio usando variabili di istanza come `self.index` e `self.data`.

Oltre alla creazione automatica del metodo e del salvataggio dello stato del programma, quando i generatori terminano, sollevano automaticamente `StopIteration`. In combinazione, queste caratteristiche rendono facile creare gli iteratori rispetto ad una funzione classica.

9.11. Generatore di Espressioni.

Alcuni generatori di espressioni semplici possono essere codificati in modo succinto utilizzando una sintassi simile a quella lista di comprensione (comprehensions) ma con parentesi quadre (**[ ]**) invece di parentesi tonde (**( )**). Queste espressioni sono progettate per situazioni in cui il generatore utilizza subito una funzione inclusa. Esso è più compatto, ma meno versatile delle funzioni complete e tendono a consumare più memoria di un'equivalente lista di comprensione. Esempio:

```
>>> sum(i*i for i in range(10))                 # somma di quadrati
285
```

```
>>> xvec = [10, 20, 30]
>>> yvec = [7, 5, 3]
>>> sum(x*y for x,y in zip(xvec, yvec))         # prodotto scalare
260

>>> from math import pi, sin
>>> sine_table = {x: sin(x*pi/180) for x in range(0, 91)}

>>> parole_uniche = set(word  for line in page  for word in
line.split())

>>> graduatoria = max((student.gpa, student.name) for student in
graduates)

>>> data = 'golf'
>>> list(data[i] for i in range(len(data)-1, -1, -1))
['f', 'l', 'o', 'g']
```

[1] Tranne che per una cosa. Oggetti modulo hanno un attributo di sola lettura chiamato `__dict__` che restituisce il dizionario usato per implementare lo spazio dei nomi (namespace) del modulo nascosto, il nome `__dict__` è un attributo ma non un nome globale. Ovviamente, questo vìola (nel senso di profanazione e non del colore) l'astrazione dell'implementazione dello spazio dei nomi, e dovrebbe essere limitato a cose come i debugger post-mortem.

10. Panoramica Sulle Librerie Standard.

10.1. Interfaccia al Sistema Operativo.

Il modulo `os` fornisce molte funzioni per interagire con il sistema operativo:

```
>>> import os
>>> os.getcwd()      # Restituisce la cartella di lavoro attuale.
'C:\\Python33'       # Macchine Windows
'/home/utente'       # macchine Linux
>>> os.chdir('/server/accesslogs')   # Cambia cartella.
>>> os.system('mkdir today')   # Lancia il comando mkdir nella shell.
0
```

Accertatevi di utilizzare `import os` invece che `from os import *`. questo garantirà l'utilizzo di `os.open()` invece della funzione precostituita `open()` che opera in modo molto diverso.

Le funzioni precostituite `dir()` e `help()` sono utili per lavorare con moduli di grandi dimensioni come `os`:

```
>>> import os
>>> dir(os)
```

```
<Restituisce la lista di tutti i moduli funzione>
>>> help(os)
<Restituisce un manuale completo creato dal modulo docstrings>
```

Per la gestione di file e directory, il modulo `shutil` fornisce un'interfaccia di più alto livello più facile da usare:

```
>>> import shutil
>>> shutil.copyfile('miei_dati.db', 'salvataggio_miei_dati.db')
>>> shutil.move('/build/executables', 'installdir')
```

10.2. File Jolly.

Il modulo `glob` fornisce una funzione per fare liste di file da directory e ricerche con i caratteri jolly:

```
>>> import glob
>>> glob.glob('*.py')
['primes.py', 'random.py', 'quote.py']
```

10.3. Argomenti della Linea di Comando.

Gli script di utilità spesso hanno bisogno di elaborare degli argomenti della riga di comando. Questi argomenti vengono memorizzati nella attributi ***argv*** del modulo `sys` come un elenco. Per esempio il comando `python demo.py uno due tre` produce i seguenti risultati:

```
>>> import sys
>>> print(sys.argv)
['demo.py', 'uno', 'due', 'tre']
```

Il modulo `getopt` processa sys.argv usando le convenzioni della funzione Unix `getopt()`. Per un'elaborazione della riga di comando più potente e flessibile, utilizzare le funzioni fornite dal modulo `argparse`.

10.4. Reidirizzamento Output degli Errori e Terminazione Programma.

Il modulo `sys` ha anche attributi per ***stdin***, ***stdout***, and ***stderr***, cioè l'input standard, l'uscita standard, e l'output per gli errori. Quest'ultimo è utile per l'emissione di avvisi e messaggi di errore per renderli visibili anche quando ***stdout*** viene rediretto:

```
>>> sys.stderr.write('Avviso! File log non trovato aperto uno
nuovo.\n')
Avviso! File log non trovato aperto uno nuovo.
```

Il modo più diretto per terminare uno script è usare `sys.exit()`.

10.5. Corrispondenza di Stringhe.

Il modulo `re` fornisce gli strumenti per le espressioni regolari per processare le stringhe. Per la corrispondenza complessa e la manipolazione, le espressioni regolari offrono soluzioni ottimizzate e compatte:

```
>>> import re
>>> re.findall(r'\bf[a-z]*', 'una foto fatta con amici fuori porta')
['foto', 'fatta', 'fuori']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'gatto nel sacco')
'gatto nel sacco'
>>> 'Fido è un bel gatto'.replace('gatto', 'cane')
'Fido è un bel cane'
```

Va notato che quando sono necessarie funzionalità semplici, i metodi di stringa sono preferiti perché sono più facili da leggere ed eseguire per il debug:

10.6. Matematica.

Il modulo di matematica `math` dà accesso alle funzioni di libreria sottostanti C per la matematica in virgola mobile:

```
>>> import math
>>> math.cos(math.pi / 4)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Il modulo `random` , fornisce strumenti per fare delle selezioni casuali:

```
>>> import random
>>> random.choice(['mela', 'pera', 'banana'])
'mela'
>>> random.sample(range(100), 8)  # campionamento senza sostituzione
[30, 83, 16, 4, 8, 81, 41, 50]
>>> random.random()               # virgola mobile casuale
0.17970987693706186
>>> random.randrange(6)           # intero casuale su 6 numeri
4
```

Il progetto SciPy ha molti altri moduli per calcoli numerici vedere <http://scipy.org>.

10.7. Accesso a Internet.

Ci sono ovviamente anche una serie di moduli per l'accesso a Internet e per l'elaborazione dei protocolli Internet. Due dei più semplici sono `urllib.request` per il recupero dei dati da un' URL e `smtplib` per l'invio di posta elettronica:

```
>>> from urllib.request import urlopen
>>> for riga in urlopen('http://tycho.usno.navy.mil/cgi-
bin/timer.pl'):
...     righe = righe.decode('utf-8')        # Da dati binari a testo
...     if 'EST' in righe or 'EDT' in righe: # cerca il Time Zone
...         print(righe)

<BR>Nov. 25, 09:43:32 PM EST
```

```
>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Rispondi al più presto.
... """)
>>> server.quit()
```

(Notare che nel secondo esempio, è necessario che il server di posta attivo sia in localhost.)

10.8. Date e Tempo.

Il modulo `datetime` fornisce classi per manipolare date e orari in modo sia semplici che complesso. Esso supporta la matematica sulle date e sugli orari, ma l'ottica dell'implementazione è concentrata sull'estrazione, la formattazione e la manipolazione degli elementi. Il modulo supporta anche oggetti per il fuso orario.

```
>>> # Le date sono facilmente gestibili.
>>> from datetime import date
>>> oggi = date.today()
>>> oggi
datetime.date(2014, 1, 25)
>>> oggi.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 25 jan 2014 is a Saturday on the 25 day of January.'

>>> # La funzione supporta l'aritmetica delle date.
>>> compleanno = date(1964, 7, 31)
>>> anni = now - compleanno
>>> anni.days
14368
```

10.9. Compressione Dati.

La compressione dati per l'archiviazione viene supportata dai relativi moduli tra cui: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` e `tarfile`.

```
>>> import zlib
>>> s = b'Nel mezzo del camin di nostra vita ......'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'Nel mezzo del camin di nostra vita ......'
>>> zlib.crc32(s)
3044838932
```

10.10. Misurazione delle Prestazioni.

Alcuni utilizzatori di Python sono incuriositi dall'efficienza del codice sviluppato a prescindere dal problema. Quale sarà il codice più veloce? Python fornisce uno strumento di misura che risponde immediatamente a queste domande.

Ad esempio, si può essere tentati di utilizzare l'impacchettamento e lo spacchettamento di una tupla caratteristica, invece del tradizionale approccio dello scambio di argomenti. Il modulo `timeit`, riesce a dimostrare anche i più piccoli vantaggi di prestazioni:

```
>>> from timeit import Timer
>>> Timer('t=a; a=b; b=t', 'a=1; b=2').timeit()
0.57535828626024577
>>> Timer('a,b = b,a', 'a=1; b=2').timeit()
0.54962537085770791
```

In contrasto con `timeit` per il buon livello di granularità, i moduli `profile` e `pstats`, forniscono strumenti per identificare sezioni critiche su grandi blocchi di codice.

10.11. Controllo Qualità.

Un approccio per sviluppare software di alta qualità è scrivere dei test per ogni funzione sviluppata ed eseguire prove frequenti durante il processo di sviluppo.

Il modulo `doctest` fornisce uno strumento per la scansione di un modulo e la convalida di test inseriti nella documentazione (docstring) del programma. La costruzione è semplice come il taglio copia e incolla e permette di inserire nella documentazione (docstring) oltre al codice, anche i risultati riportati da un' esempio per fare in modo che il tutto risulti più chiaro. Questo migliora la documentazione da fornire all'utente che con un semplice esempio, permette al modulo doctest di convalidare la documentazione:

```
def media(valori):
    """Calcola la media aritmetica di una lista di numeri.
```

```
    >>> print(media([20, 30, 70]))
    40.0
    """
    return sum(valori) / len(valori)

import doctest
doctest.testmod()   # Convalida automaticamente i test integrati.
```

Il modulo unittest non è più piacevole del modulo doctest, ma permette una serie più completa e capibile di test da essere mantenuto in un file separato:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

    def test_average(self):
        self.assertEqual(average([20, 30, 70]), 40.0)
        self.assertEqual(round(average([1, 5, 7]), 1), 4.3)
        with self.assertRaises(ZeroDivisionError):
            average([])
        with self.assertRaises(TypeError):
            average(20, 30, 70)

unittest.main() # Chiamato dalla linea di comando, che lancia tutti i
test
```

10.12. Tutto compreso.

Python ha una filosofia "tutto compreso". Questo si nota meglio attraverso le sofisticate e robuste caratteristiche dei dei suoi pacchetti più grandi. Per esempio:

- I moduli xmlrpc.client e xmlrpc.server implementano chiamate a procedure remote con un' utilizzo quasi banale. Nonostante i nomi dei moduli lo farebbero pensare, non è necessaria alcuna conoscenza diretta della gestione di XML.
- Il package per la gestione delle email è una libreria che comprende i MIME e altri messaggi di documenti RFC basati su 2822. A differenza di smtplib e poplib che in realtà inviano e ricevono messaggi, il package email ha un set di strumenti completo per la costruzione o la decodifica di complesse strutture di messaggi (allegati inclusi) per attuare la codifica internet e i protocolli di intestazione.
- I pacchetti xml.dom e xml.sax forniscono un robusto supporto per l'analisi e l'interscambio di questo popolare formato di dati. Allo stesso modo, il modulo csv supporta la lettura e scrittura diretta di un formato DataBase. L'insieme, questi moduli e pacchetti semplificano notevolmente lo scambio di dati tra applicazioni Python e altri strumenti.
- L'internazionalizzazione è supportata da una serie di moduli, inclusi tra cui gettext, locale, e il pacchetto di codecs.

11. Panoramica Sulle Librerie Standard Parte II.

Questa seconda panoramica copre i moduli più avanzati che danno supporto alle esigenze di programmazione professionali. Questi moduli raramente s'incontrano in piccoli script.

11.1. Output Formattato

Il modulo `reprlib` fornisce una versione di `repr()` su misura per i display piccoli o di grandi dimensioni con molteplici nidificazioni di codice:

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticoespiralitoso'))
"set(['a', 'c', 'd', 'e', 'f', 'g', ...])"
```

Il modulo di `pprint` (la p come prefisso a print, sta a pretty cioè graziosa) offre un controllo più sofisticato su stampa sia precostituita che in oggetti definiti dall'utente in un modo che sia leggibile dall'interprete. Quando il risultato è più lungo di una riga, la "stampa graziosa", aggiunge interruzioni di riga e rientro per formattare più chiaramente struttura dei dati rendendola più intelleggibile per quanto possibile:

```
>>> import pprint
>>> t = [[['nero', 'ciano'], 'bianco', ['verde', 'rosso']],
[['magenta',
...     'giallo'], 'blu']]]
...
>>> pprint.pprint(t, width=30)
[[[['nero', 'ciano'],
   'bianco',
   ['verde', 'rosso']],
  [['magenta', 'giallo'],
   'blu']]]
```

Il modulo `textwrap` formatta paragrafi di testo per adattarla ad una determinata larghezza dello schermo:

```
>>> import textwrap
>>> documento = """Il metodo wrap() è proprio come fill() eccetto che
restituisce
... una lista di stringhe con caratteri di nuova linea invece di una
grossa stringa unica."""
...
>>> print(textwrap.fill(documento, width=40))
Il metodo wrap() è proprio come fill()
eccetto che restituisce  una lista di
stringhe con caratteri di nuova linea
invece di una grossa stringa unica.
```

Il modulo `locale` accede ad un database specifico per le convezioni culturali per i formati di dati in uso

nei vari paesi in cui vengono utilizzati. L'attributo di raggruppamento della funzione format di locale fornisce un modo diretto per la formattazione dei numeri con separatori di gruppo:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'Inglese_Stati-Uniti.1252')
'Inglese_Stati-Uniti.1252'
>>> conv = locale.localeconv()    #Acquisisce una mappa convenzionale
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                      conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2. Modellazione.

Il modulo string include una classe versatile Template (modello) con una sintassi semplificata adatta alla modifica da parte di utenti finali. Questo permette di personalizzare le proprie applicazioni senza dover modificare l'applicazione.

Il formato utilizza dei simboli come segnaposto formate da $ ed altri identificatori validi di Python (caratteri alfanumerici e di sottolineatura) i quali all'interno di parentesi graffe, permettono la sostituzione di blocchi di testo in modo dinamico. Scrivendo $ $ crea un unico $ altrimenti non rappresentabile:

```
>>> from string import Template
>>> t = Template('${città} quartiere1 €10 per $causale.')
>>> t.substitute(città='Roma', causale='accesso musei')
'Roma quartiere1 €10 per accesso musei.'
```

Il metodo substitute() solleva un KeyError quando un segnaposto non viene fornito in un dizionario o in un' argomento chiave. Per le applicazioni in stile mail-merge, i dati degli utenti forniti possono essere incompleti e il metodo safe_substitute() può essere più appropriato lasciando così il segnaposto invariato se mancano i dati non sollevando errori:

```
>>> t = Template('Restituisce $oggetto a $proprietario.')
>>> d = dict(item='bicicletta')
>>> t.substitute(d)
Traceback (most recent call last):
  ...
KeyError: 'proprietario'
>>> t.safe_substitute(d)
'Restituisce bicicletta a $proprietario .'
```

Modelli di sottoclassi, possono specificare un delimitatore personalizzato. Ad esempio, un lotto ridenominato utilità per un visualizzatore di foto può scegliere di utilizzare segni di percentuale per i segnaposto come il formato di data corrente, numero di sequenza di immagini, o file:

```
>>> import time, os.path
>>> FileFoto = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Modello):
...     separatore = '%'
>>> tmp = input('Stile formato (%d-date %n-seqnum %f-format): ')
Stile formato (%d-date %n-seqnum %f-format):  Marco_%n%f

>>> t = BatchRename(tmp)
>>> date = time.strftime('%d%b%y')
>>> for i, NomeFile in enumerate(FileFoto):
...     base, ext = os.path.splitext(NomeFile)
...     NuovoNome = t.substitute(d=date, n=i, f=ext)
...     print('{0} --> {1}'.format(NomeFile, NuovoNome))

img_1074.jpg --> Marco_0.jpg
img_1076.jpg --> Marco_1.jpg
img_1077.jpg --> Marco_2.jpg
```

Un'altra applicazione di modellazione è presente nella logica dei dettagli nei programmi in molteplici formati di output. Ciò rende possibile sostituire modelli personalizzati per i file XML, rapporti di testo, e i rapporti web HTML.

11.3. Lavorare con i Data Record Layouts Binari.

Il modulo `struct` è munito delle funzioni `pack()` e `unpack()` per lavorare con formati di record binari di lunghezza variabili. L'esempio seguente mostra come scorrere le informazioni sull'intestazione in un file ZIP senza utilizzare il modulo `zipfile`. I codici `"H"` e `"I"` rappresentano rispettivamente numeri di due e quattro byte senza segno. `'"<"` indica che sono di dimensioni standard e l'ordine dei byte è di tipo little-endian:

```
import struct

with open('miofile.zip', 'rb') as f:
    mieidati = f.read()

inizio = 0
for i in range(3):                      # mostra i primi 3 file.
headers
    inizio += 14
    fields = struct.unpack('<IIIHH', mieidati[inizio:inizio+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

    inizio += 16
    filename = mieidati[inizio:inizio+filenamesize]
    inizio += filenamesize
    extra = mieidati[inizio:inizio+extra_size]
    print(filename, hex(crc32), comp_size, uncomp_size)

    inizio += extra_size + comp_size   # passa alla intestazione dopo
```

11.4. Multi-threading.

Il Threading è una tecnica per il disaccoppiamento dei processi che non sono sequenzialmente dipendenti e che vengono eseguiti in modo indipendente. I threads (i singoli processi) possono essere utilizzati per migliorare la reattività delle applicazioni che accettano input dell'utente mentre altre elaborazioni possono essere eseguite in background (sottofondo). Un caso d'uso, può essere quello relativo è in funzione di I / O in parallelo con i calcoli in un altro processo, oppure la scrittura di un file mentre si è in attesa di un input da tastiera.

Il seguente codice, mostra come il modulo di alto livello `threading` può essere elaborato in background mentre il programma principale continua a fare altre cose:

```
import threading, zipfile           # Importa i due moduli necessari

class Asincrona(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile
    def run(self):
        f = zipfile.ZipFile(self.outfile, 'w', zipfile.ZIP_DEFLATED)
        f.write(self.infile)
        f.close()
        print('Finished background zip of:', self.infile)

background = Asincrona('mieidata.txt', 'mioarchivio.zip')
background.start()
print('Il programma principale è sempre in primo piano.')

background.join()  # Aspetta che il processo in secondo piano termini
print('Il programma principale, aspetta che il processo termini.')
```

Il principale problema che le applicazioni multi-threaded pongono, sta nella condivisione dei dati con altre risorse. A tal fine, il modulo threading fornisce una serie di primitive di sincronizzazione, tra cui blocchi, eventi, variabili di condizione, e semafori.

Con questi potenti strumenti, anche i minimi errori di progettazione, possono causare problemi difficili da individuare. Quindi, l'approccio preferito per il coordinamento, è quello di concentrare tutto l'accesso a una risorsa in un singolo thread e di utilizzare quindi il modulo `queue` per alimentare il flusso con richieste da altri thread. Le applicazioni che utilizzano oggetti `queue` per la comunicazione inter-thread per il coordinamento sono più facili da progettare, più leggibili e più affidabili.

11.5. Logging.

Il modulo `logging` offre un completo sistema di registrazione caratteristico e flessibile. Nel caso più semplice, i messaggi di log vengono inviati ad un file o al `sys.stderr`:

```
import logging
logging.debug('Informazione di debugging')
logging.info('Messaggio')
logging.warning('Attenzione:file config %s non trovato',
'server.conf')
logging.error('Errore!')
logging.critical('Errore critico -- uscita')
```

Che produce il seguente output:

```
WARNING:root: Attenzione:file config file server.conf  non trovato
ERROR:root:Errore!
CRITICAL:root: Errore critico -- uscita
```

Per impostazione predefinita, i messaggi informativi e di debug sono soppressi e l'uscita viene inviata su standard error. Altre opzioni di output includono l'instradamento dei messaggi tramite e-mail, datagrams, sockets, o ad un server HTTP. Nuovi filtri possono scegliere un' instradamento diverso in base alla priorità del messaggio: `DEBUG`, `INFO`, `WARNING`, `ERROR`, e `CRITICAL`.

Il sistema di registrazione può essere configurato direttamente da Python o può essere caricato da un file di configurazione editabile dall'utente per la registrazione personalizzata senza modificare l'applicazione.

11.6. Referimenti deboli

Python fa una gestione automatica della memoria (il conteggio dei riferimenti per la maggior parte degli oggetti e la *garbage collection* (pulizia della memoria) per eliminare cicli). La memoria viene liberata appena l'ultimo riferimento ad essa è stato eliminato.

Questo meccanismo funziona bene per la maggior parte delle applicazioni ma occasionalmente vi è la necessità di tenere traccia degli oggetti solo finché sono utilizzati da qualcos'altro. Purtroppo, proprio il loro mantenimento crea di fatto un riferimento che li rende permanenti. Il modulo `weakref` fornisce gli strumenti per il monitoraggio di oggetti senza creare un punto di riferimento. Quando l'oggetto non è più necessario, viene rimosso automaticamente da una tabella del modulo `weakref` e una chiamata viene attivata per gli oggetti. Le applicazioni tipiche comprendono oggetti di caching che sono costose da creare:

```
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                    # crea una referenza
>>> d = weakref.WeakValueDictionary()
>>> d['primario'] = a            # non crea una referenza
>>> d['primario']                # Recupera l'oggetto se attivo
10
>>> del a                        # rimuove l'unica referenza
>>> gc.collect()                 # esegue il garbage collection subito
0
>>> d['primario']                # è stato rimosso automaticamente
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primario']               # è stato rimosso automaticamente
  File "C:/python33/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primario'
```

11.7. Strumenti per Lavorare con le Liste.

Molte strutture dati, si possono gestire con funzioni precostituite proprie, tuttavia, a volte vi è la necessità di implementazioni alternative con differenti prestazioni e compromessi.

Il modulo `array` (matrice) fornisce un' oggetto `array()` che è come una lista che memorizza solo i dati omogenei e li memorizza in modo più compatto. L'esempio seguente mostra un array di numeri memorizzati come due byte di numeri binari senza segno (typecode `"H"`) anziché i soliti 16 byte per voce per le liste regolari di oggetti tipo int di Python:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

Il modulo `collections` fornisce un' oggetto `deque()` che è come una lista con accodamento ma più veloce dal lato sinistro, ma con ricerche più lente nel mezzo. Questi oggetti sono adatti per effettuare code e grandi ricerche su strutture ad albero:

```
>>> from collections import deque
>>> d = deque(["compito1", "compito2", "compito3"])
>>> d.append("compito4")
>>> print("Collegamento", d.popleft())
Collegamento compito1
noncercato = deque([nodo_partenza])
def ampiezza_prima_ricerca(noncercato):
    nodo = noncercato.popleft()
    for m in gen_moves(nodo):
        if is_goal(m):
            return m
        noncercato.append(m)
```

In aggiunta alla lista delle implementazioni, la libreria offre in alternativa anche altri strumenti come il modulo `bisect` con funzioni per manipolare elenchi ordinati:

```
>>> import bisect
>>> punti = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500,
```

```
'python')]
>>> bisect.insort(punti, (300, 'ruby'))
>>> punti
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500,
'python')]
```

Il modulo heapq fornisce funzioni per l'attuazione di pile sulla base di liste regolari. La voce più bassa del valore è sempre mantenuta in posizione zero. Questo è utile per applicazioni che accedono ripetutamente l'elemento di valore più in basso, ma che non vogliono scorrere l'elenco completo ordinato:

```
>>> from heapq import heapify, heappop, heappush
>>> dati = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(dati)                      # riarrangia la lista nella pila
>>> heappush(dati, -5)                 # aggiunge un nuovo valore
>>> [heappop(dati) for i in range(3)]# estrae i tre più piccoli
[-5, 0, 1]
```

11.8. Aritmetica Decimale in Virgola Mobile.

Il modulo decimal offre un tipo di dati Decimal per l'aritmetica decimale in virgola mobile. Rispetto all'implementazione della funzione precostituita float questa classe specializzata viene in aiuto per:

- Applicazioni finanziarie e altri usi che richiedono una rappresentazione decimale esatta.
- Un controllo sulla precisione.
- Un controllo sull' arrotondamento per soddisfare i requisiti legali o regolamentari.
- Il monitoraggio di importanti cifre decimali.
- Applicazioni in cui l'utente si aspetta che i risultati corrispondano ai calcoli fatti a mano.

Ad esempio, calcolare una tassa del 5% su una tariffa telefonica 70 centesimi dà risultati diversi in virgola mobile decimale rispetto al binario in virgola mobile. La differenza diventa significativa se i risultati sono arrotondati al centesimo più vicino:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
0.73
```

A causa dei limiti hardware di ogni calcolatore, la precisione restituita dai numeri in virgola mobile seppur precisa, è sempre approssimativa quindi non adatta per certi usi. Decimal invece come già accennato sopra, permette di superare questo limite a prezzo di una minore velocità di calcolo. Decimal permette di definire la precisione in termini di decimali dopo la virgola specificando quanti decimali ci occorrono con risultati accurati la dove necessitano.
E' possibile inoltre con il modulo decimal definire la precisione dei numeri per tutto il programma o parte di esso rendendoli come predefiniti.

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

12. E Adesso?

La lettura di questa guida probabilmente ha suscitato l'interesse as utilizzare Python per risolvere i problemi del mondo reale. Cosa si deve fare per saperne di più?

Questa guida, è parte di un gruppo di documentazione Python, altra documentazione è possibile trovarla a questi indirizzi:

- *The Python Standard Library*:

  Si consiglia di sfogliare questo manuale, che dà un completo (anche se terse) riferimento materiale sui tipi, funzioni e moduli della libreria standard. La distribuzione standard di Python comprende un sacco di codice aggiuntivo. Ci sono moduli per leggere caselle di posta Unix, recuperare i documenti via HTTP, generare numeri casuali, analizzare le opzioni della riga di comando, scrivere programmi CGI, comprimere i dati, e molti altri compiti. Sfogliando il Reference Library avrete un'idea di ciò che è disponibile.
- *Installing Python Modules* spiega come installare moduli esterni scritti da altri utenti Python.
- *The Python Language Reference*: Spiega in modo dettagliato la sintassi e la semantica di Python. E una lettura un po' pesante, ma è utile come una guida completa al linguaggio.

Ancora altre risorse:

- http://www.python.org: Il principale sito web di Python. Esso contiene il codice, documentazione e puntatori a pagine su Python in tutto il web. Questo sito Web si riflette in vari siti del mondo, come Europa, Giappone e Australia, con una visibilità che può essere più veloce rispetto al sito principale, a seconda della posizione geografica.
- http://docs.python.org: L'accesso veloce alla documentazione di Python.
- http://pypi.python.org: L'indice dei Packages Python, in precedenza anche soprannominato Cheese Shop, è un indice di moduli Python creati dagli utenti disponibili per il download. Una volta che si inizia a scrivere del codice utile, puoi registrarti qui e mettere a disposizione degli altri dei moduli tuoi in modo che possano essere scaricati.
- http://aspn.activestate.com/ASPN/Python/Cookbook/: Il Python Cookbook è un posto con una

raccolta considerevole di esempi di codice, di moduli più grandi, e utili script. Contributi particolarmente importanti sono raccolti anche in un libro intitolato “Python Cookbook” (O'Reilly & Associates, ISBN 0-596-00797-3.)

- http://scipy.org: Il progetto scientifico di Python che include i moduli per calcoli di matrice veloci e manipolazioni, più una serie di pacchetti per cose come l'algebra lineare, trasformate di Fourier, solutori non lineari, le distribuzioni di numeri casuali, analisi statistica e simili.

Per domande correlate a Python e segnalazioni di problemi, è possibile postare il comp.lang.python newsgroup, o inviarli alla mailing list a python-list@python.org. Il newsgroup e mailing list sono gatewayed, quindi i messaggi inviati ad uno verranno automaticamente inoltrati all'altro. Ci sono circa 120 messaggi al giorno (con punte fino a diverse centinaia), chiedendo (e rispondendo) alle domande, suggerendo nuove funzionalità, e annunci di nuovi moduli. Prima di postare, assicurati di controllare la lista delle domande più frequenti (chiamate FAQ). Archivi delle mailing list sono disponibili a http://mail.python.org/pipermail/. Le FAQ rispondono a molte delle domande che si susseguono, dove potrebbe esserci la soluzione al vostro problema.

13. Input ed Editing Interattivo Sostituzioni e Cronologia.

Alcune versioni di Python supportano l'editing della riga di input corrente e la sostituzione con la cronologia, simili agli editor presenti nella shell Korn e la shell GNU Bash. Questo è implementato utilizzando la libreria GNU GNU Readline, che supporta lo stile Emacs e l'editing in stile vi. Questa libreria ha la propria documentazione che non voglio duplicare qui, tuttavia, le basi sono facilmente spiegabili. L'editing interattivo e la cronologia qui descritta sono disponibili a scelta nelle versioni Unix e Cygwin dell'interprete.

Questo capitolo non documenta i servizi di editing del pacchetto PythonWin di Mark Hammond o l'ambiente IDLE basato su Tk distribuito con Python. Il richiamo alla storia della riga di comando che opera all'interno delle finestre DOS su NT e alcuni altri ricordi DOS e Windows sono un'altra bestia.

13.1. Editare le Linee.

Se supportato, l'editing della riga di input è attivo ogni volta che l'interprete stampa un prompt primario o secondario. L'attuale linea può essere modificata usando i caratteri di controllo convenzionali di Emacs. I più importanti sono (`CTRL` sta per Control + tasto specifico):

- `CTRL-A` sposta il cursore all'inizio della riga.
- `CTRL-E` sposta il cursore alla fine.
- `CTRL-B` lo muove di una posizione a sinistra.
- `CTRL-F` lo muove di una posizione a destra.
- `BackSpace` cancella il carattere a sinistra del cursore.
- `CTRL-D` cancella il carattere alla sua destra.
- `CTRL-K` elimina il resto della riga alla destra del cursore.
- `CTRL-Y` ripristina l'ultima stringa eliminata.
- `CTRL-_` (sottolineatura) annulla l'ultima modifica effettuata, può essere ripetuto per effetto cumulativo.

13.2. Sostituzione cronologica.

La sostituzione cronologica funziona come segue. Tutte le linee di ingresso non vuote immesse vengono salvati in una memoria storica sequenziale, e quando viene dato un nuovo prompt si posiziona su una nuova riga in fondo a questa memoria. Quindi:

- `CTRL-P` sposta di una riga verso l'alto (indietro) nella memoria storica.
- `CTRL-N` si muove su una riga sotto (se esiste).
- `CTRL-S` avvia una ricerca in avanti.

Ogni linea nella memoria storica, può essere modificata. Premendo il tasto `INVIO` si passa la riga corrente all'interprete.

13.3. Tasti Collegati.

Le associazioni dei tasti e alcuni altri parametri della libreria readline possono essere personalizzati inserendo i comandi in un file di inizializzazione chiamato `~/.inputrc`. Le associazioni dei tasti hanno la seguente forma:

```
key-name: nome-funzione
```

oppure:

```
"stringa": nome-funzione
```

più opzioni possono essere impostate con:

```
set nome-opzione valore
```

Per esempio:

```
# A chi preferisce lo stile dell'editor vi:
set editing-mode vi

# Editare usando una linea singola:
set horizontal-scroll-mode On

# Ridefinire il collegamento di alcuni tasti:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Si noti che l'associazione per il `Tab` in Python è predefinito ad inserire un carattere di tabulazione invece di una funzione di completamento di Readline. Volendo, è possibile ignorare questo comportamento immettendo:

```
Tab: complete
```

nel file vostro file `~/.inputrc`. (Naturalmente, questo rende più difficile le righe di continuazione indentate se siete abituati ad usare `Tab` per tale scopo.)

Il completamento automatico dei nomi delle variabili e dei moduli è disponibile come optional. Per abilitarlo in modalità interattiva dell'interprete, aggiungere quanto segue al file di avvio: [1]

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

La seconda riga, associa il tasto `Tab` per la funzione di completamento, questo premendo due volte il tasto Tab.

Un file di impostazioni d'avvio evoluto, potrebbe essere simile a questo esempio. Si noti che questo cancella i nomi che crea una volta che non sono più necessari, questo viene fatto in quanto il file di avvio viene eseguito nello stesso spazio dei nomi dei comandi interattivi, e la rimozione dei nomi evita effetti collaterali per l'ambiente interattivo. Può essere utile mantenere alcuni moduli importati, come `os`, che risultano essere necessari nella maggior parte delle sessioni dell'interprete.

```
# Aggiunge l'auto-completamento e un file per la cronologia comandi.
# Viene attivato autocomplete .
# Setta il tasto 'Esc' come predefinito (si può cambiare
# vedere al documentazione readline).
# Salva nel file ~/.pystartup, e imposta una variabile ambiente
# a cui puntare. Che è:  "export PYTHONSTARTUP=~/.pystartup" con bash.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/.pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

13.4. Alternative all'Interprete Interattivo Predefinito.

Le funzionalità di questo nuovo interprete, è un enorme passo in avanti rispetto alle precedenti versioni tuttavia, alcuni cose lasciano ancora a desiderare: Sarebbe bello se la corretta indentazione fosse possibile su linee successive, magari aggiungendo un trattino di sottolineatura presente in altri linguaggi! (il parser saprebbe distinguerlo). Poi il meccanismo di completamento potrebbe usare la tabella dei simboli dell'interprete. Ancora un comando per controllare (o anche suggerire) le corrispondenze tra parentesi, le citazioni, ecc, tutto ciò, sarebbe molto utile. In effetti l'interprete standard così com'è non si preoccupa

molto dell'utente, presuppone un' utilizzatore che abbia grande dimestichezza con editor a linea di comando del tipo **vi** ed un grande allenamento digitale.

Un'alternativa valida all'interprete interattivo è IPython, che dispone del completamento automatico, l'esplorazione degli oggetti e la gestione avanzata della cronologia. Inoltre esso, può anche essere accuratamente adattato e incorporato in altre applicazioni. Un ambiente interattivo avanzato simile è bpython. Un' altro editor molto utilizzato presente su tutte le piattaforme è IDLE.

Note:

[1] Python eseguirà i contenuti di un file identificato dalla variabile di ambiente PYTHONSTARTUP quando si avvia l'interprete interattivo. Per personalizzare Python anche per la modalità non-interattiva, fare riferimento a *The Customization Modules*.

14. Aritmetica in Virgola Mobile: Problemi e Limiti.

I numeri in virgola mobile sono rappresentati nell'hardware del computer in base due (binari). Ad esempio, la frazione decimale:

0.125

Ha come valore **1/10 + 2/100 + 5**/1000.

Allo stesso modo la frazione binaria:

0.001

Il cui valore è **0/2 + 0/4 + 1/8**, rappresenta lo stesso valore, l'unica vera differenza è che il primo è scritto in notazione frazionaria base 10, e il secondo in base 2 (binaria).

Purtroppo, la maggior parte delle frazioni decimali non possono essere rappresentati esattamente come le frazioni binarie. Una conseguenza è che, in generale, i decimali numeri a virgola mobile inseriti sono solo delle approssimazioni ai numeri in virgola mobile binari effettivamente memorizzati nella macchina.

Il problema è più facile da capire prima in base 10. Si consideri la frazione **1/3**. Si può approssimare che come una frazione base 10:
0.3

o ancora più preciso:

0.33

ancora più preciso:

0.333

E così via. Non importa quante cifre siete disposti a scrivere, il risultato non sarà mai esattamente **1/3**, ma sarà un sempre un'approssimazione di **1/3** al meglio.

Allo stesso modo, non importa quanti bit su base 2 siete disposti a utilizzare, il valore decimale **0.1** non può essere rappresentato esattamente come frazione in base 2. In base 2, **1/10** è una frazione che si ripete all'infinito.

0.0001100110011001100110011001100110011001100110011...

Risulta evidente che sia necessario uno stop all'approssimazione dei decimali in funzione di ciò che si vuole ottenere. Su molte macchine oggi, i numeri in virgola mobile, sono approssimati utilizzando una frazione binaria con il numeratore che utilizza i primi 53 bit a partire dal bit più significativo e con il denominatore come una potenza di due. Nel caso di **1/10**, la frazione binaria è **`3602879701896397 / 2 ** 55`** che è simile ma non è esattamente uguale al valore reale di **1/10**.

Molti utenti non sono consapevoli di questa approssimazione a causa del modo in cui vengono visualizzati i valori. Python stampa solo una approssimazione decimale al valore decimale che è l' approssimativo più vicino all'hardware della macchina su cui Python viene eseguito. Sulla maggior parte delle macchine, se Python dovesse stampare il valore decimale vale per il ravvicinamento binario memorizzato per **0.1**, stamperebbe:

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Che ha cifre più che sufficienti per la maggior parte degli utilizzatori, quindi Python mantiene il numero di cifre ad alta approssimazione visualizzando però un valore arrotondato che è più umano.

```
>>> 1 / 10
0.1
```

È interessante notare che ci sono molti numeri decimali diversi che condividono la stessa vicina frazione binaria approssimativa.
Ad esempio i numeri:

0.1
0.10000000000000001
0.1000000000000000055511151231257827021181583404541015625

sono tutti approssimati da **3602879701896397 / 2 ** 55**. Dal momento che tutti questi valori decimali condividono la stessa approssimazione, ognuno di loro potrebbe essere visualizzato utilizzando `eval(repr(x)) == x`.

Storicamente, la funzione Python precostituita `repr()` dovrebbe scegliere quella con 17 cifre significative **0.10000000000000001**. Ma a partire da Python 3.1, sulla maggior parte dei sistemi è ora in grado di scegliere la più breve rappresentazione e visualizzare semplicemente **0.1**.

Si noti che questo è nella natura stessa del binario in virgola mobile, non è un bug in Python, e non è un bug generato dal codice. Vedrete la stessa cosa in tutti i linguaggi che supportano aritmetica in virgola mobile sul vostro hardware (anche se in alcuni linguaggi potrebbero non visualizzare la differenza per impostazione predefinita, o in tutte le modalità di uscita).

Per un' output più piacevole, si potrebbe desiderare di utilizzare i numeri sotto forma di stringhe formattate per visualizzare solo i decimali voluti.

```
>>> format(math.pi, '.12g')  # Da 12 decimali significativi.
'3.14159265359'

>>> format(math.pi, '.2f')   # Da 2 decimali significativi.
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

E 'importante rendersi conto che questa è solo un' illusione. Si sta semplicemente arrotondando la visualizzazione del vero valore per la propria macchina. Una illusione può generarne un' altra. Per esempio, dal **0.1** non corrisponde esattamente a **1/10**, sommando tre valori di **0.1** non si può produrre esattamente **0.3**, cioè:

```
>>> .1 + .1 + .1 == .3
False
```

Infatti la vera somma è:

0.30000000000000004

Inoltre, dato che il **0.1** non può ottenere il valore esatto di **1/10** e **0.3** non può ottenere il valore esatto più vicino a **3/10**, la funzione `round()`, ci può venire in aiuto con un pre-arrotondamento:

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

Anche se i numeri non possono essere resi più vicini ai loro valori esatti previsti, la funzione `round()` può essere utile per il post-arrotondamento in modo che i risultati con i valori esatti diventino confrontabili tra di loro:

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

L'aritmetica binaria in virgola mobile riserva molte sorprese come questa. Il problema con "**0.1**" è spiegato qui sotto in modo più dettagliato, nella sezione "Errori di rappresentazione". Vedere The Perils of Floating Point (I pericoli della virgola mobile) per un resoconto più completo ed altre sorprese.

Come che dice per finire, "non ci sono risposte facili." Comunque, non bisogna drammatizzare ed essere eccessivamente cauti con i numeri in virgola mobile! Gli errori nelle operazioni che Python fa, vengono ereditate dall'hardware sottostante, e sulla maggior parte delle macchine sono dell'ordine di non più di **1** su **2 ** 53** per operazione. Questo è più che sufficiente per la maggior parte delle attività, anche se va tenuto conto che ad ogni operazione in virgola mobile può subire un nuovo errore di arrotondamento, infatti gli **errori si sommano sempre non si sottraggono mai**.

Mentre esistono casi patologici, nella maggior parte dell'uso occasionale in virgola mobile avrete il risultato che vi aspettavate, semplicemente con il numero di cifre decimali che ci si aspetta. La funzione

`str()` di solito basta, mentre per un controllo più preciso fare riferimento al metodo `str.format()` meglio specificato in *Format String Syntax*.

Per i casi d'uso che richiedono una rappresentazione decimale più esatta, provare a utilizzare il modulo `decimal` che implementa l'aritmetica decimale adatto per applicazioni di contabilità e applicazioni di alta precisione.

Un'altra forma di aritmetica esatta è supportata dal modulo `fractions` che implementa aritmetica basata sui numeri razionali (così che i numeri **1/3** possono essere rappresentati esattamente).

Se sei un utente che fa uso pesante della virgola mobile, dai un'occhiata al pacchetto **Numerical Python Packagee** ed in molti altri pacchetti per le operazioni matematiche e statistiche fornite dal progetto **SciPy**. Vedere <http://scipy.org>.

Python fornisce strumenti che possono aiutare in quelle rare occasioni in cui davvero si vuole conoscere il valore esatto di un numero in virgola mobile. Il metodo `float.as_integer_ratio()` esprime il valore in virgola mobile come frazione:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Poiché il rapporto è esatto, può essere utilizzato per ricreare senza perdite il valore originale:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

Il metodo `float.hex()` rappresenta un valore in virgola mobile in esadecimale (base 16), anche in questo caso il valore esatto è quello memorizzato dal computer:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Questa rappresentazione esadecimale precisa può essere utilizzata per ricostruire esattamente il valore in virgola mobile.

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Dal momento che la rappresentazione è esatta, questa tecnica è utile per scambiare in modo affidabile valori tra le diverse versioni di Python (indipendenti dalla piattaforma) e lo scambio di dati con altri linguaggi che supportano lo stesso formato (come Java e C99).

Un altro strumento utile è la funzione `math.fsum()`, che consente di ridurre la perdita di precisione durante la sommatoria. Essa tiene traccia delle cifre perdute negli arrotondamenti per poi recuperarle nel finale. Questo può fare la differenza nella precisione complessiva modo che gli errori non si accumulano in modo significativo tali da inficiare il risultato finale.

```
>>> sum([0.1] * 10) == 1.0
False
```

```
>>> math.fsum([0.1] * 10) == 1.0
True
```

14.1. Errori di Rappresentazione.

Questa sezione illustra l'esempio "**0.1**" nel dettaglio, e mostra come è possibile eseguire un'analisi esatta in casi come questo da soli. Si assume che si abbia una certa familiarità con la rappresentazione in virgola mobile.

Errori di rappresentazione si riferisce al fatto che alcune (la maggior parte, in realtà), frazioni decimali non possono essere rappresentate esattamente come nel formato binario (base 2). Questa è la ragione principale per cui Python (o Perl, C, C + +, Java, Fortran, e molti altri), spesso non visualizzano il numero decimale esatto che ci si aspetta.

Ma perché è così? Perché **1/10** non è esattamente rappresentabile come frazione binaria? Quasi tutte le macchine di oggi utilizzano lo standard **IEEE-754** per l'aritmetica in virgola mobile, e quasi tutte le piattaforme di Python utilizzano **IEEE-754** in "doppia precisione". Esso utilizza 53 bit di precisione, per cui a comando il computer si sforza di convertire **0.1** alla frazione più vicina che può della forma **J / 2 ** N** dove **J** è un intero contenente esattamente 53 bit. Scrivendo

```
1 / 10 ~= J / (2**N)
```

è

```
J ~= 2**N / 10
```

ricordando che **J** ha esattamente 53 bit (è > = **2 ** 52** ma < **2 ** 53**), il miglior valore per **N** è **56**:

```
>>> 2**52 <=  2**56 // 10  < 2**53
True
```

Cioè, il **56** è l'unico valore per **N** che lascia **J** con esattamente 53 bit. Il valore migliore possibile per **J** è allora con quoziente arrotondato:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Poiché il resto è più della metà di 10, la migliore approssimazione si ottiene arrotondando:

```
>>> q+1
7205759403792794
```

Pertanto la migliore approssimazione di **1/10** in 754 doppia precisione è:

```
7205759403792794 / 2 ** 56
```

Dividendo sia il numeratore che il denominatore per due riduce la frazione a:

```
3602879701896397 / 2 ** 55
```

Si noti che da quando abbiamo arrotondato, questo è in realtà un po 'più grande di **1/10**; ma se non avessimo arrotondato, il quoziente sarebbe stato un po' più piccolo di **1/10**. Ma in nessun caso può essere esattamente **1/10!**

Così il computer non "vede" **1/10**, ma ciò che vede è la frazione esatta di cui sopra, con la miglior doppia approssimazione che lo standard 754 può ottenere:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Se moltiplichiamo la frazione per **10 ** 55**, possiamo vedere il valore a 55 cifre decimali:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
1000000000000000055511151231257827021181583404541015625
```

il che significa che il numero esatto memorizzato nel computer è uguale al valore decimale **0.1000000000000000055511151231257827021181583404541015625**. Invece di visualizzare il valore decimale pieno, molti linguaggi (comprese le vecchie versioni di Python), arrotondano il risultato a 17 cifre significative:

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

I moduli `fractions` e `decimal`, permettono una facile gestione dei decimali:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.1000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17')
'0.10000000000000001'
```