Il manuale di riferimento di Python

Versione 2.3.4

Guido van Rossum
Fred L. Drake, Jr., editor

19 marzo 2005

Python Software Foundation
Email: docs@python.org
Traduzione presso
http://www.zonapython.it
Email: zap@zonapython.it

Sommario

Python è un linguaggio di programmazione di alto livello, interpretato, orientato agli oggetti e con una semantica dinamica. Il suo alto livello di costrutti nelle strutture dati, combinato con la tipizzazione ed i binding dinamici, lo rende molto interessante per lo sviluppo rapido di applicazioni, così come per l'utilizzo come linguaggio di scripting o come linguaggio collante per connettere assieme componenti esistenti. La sintassi semplice e facile da apprendere di Python enfatizza la leggibilità e riduce il costo di mantenimento dei programmi. Python supporta moduli e package, incoraggiando così la programmazione modulare ed il riutilizzo del codice. L'interprete Python e l'estesa libreria standard sono disponibili sia come sorgente che in forma binaria, senza costo per le maggiori piattaforme, possono inoltre essere ridistribuiti liberamente.

Il manuale di riferimento descrive la sintassi e la "core semantics" del linguaggio. È chiara e semplice, ma cerca di essere esatta e completa. La semantica dei tipi di oggetto built-in non essenziali, delle funzioni built-in e dei moduli, viene descritta nella *Libreria Python di riferimento*. Per un'introduzione informale al linguaggio, si puó consultare il *Tutorial di Python*. Per i programmatori C o C++, esistono due manuali aggiuntivi: *Extending and Embedding the Python Interpreter* dà una visione ad alto livello di come scrivere un modulo di estensione Python ed il *Python/C API Reference Manual* descrive in dettaglio le interfacce disponibili ai programmatori C/C++.

Traduzione in italiano

Per quanto riguarda la traduzione di questo documento hanno collaborato molti volontari, nel più puro spirito del “free software”, con l’intento di rendere disponibile a tutti questo manuale nella lingua italiana.

Traduzione in Italiano del *Manuale di riferimento di Python*, sotto la coordinazione di Ferdinando Ferranti zap[at]zonapython.it, hanno collaborato a tradurre, in ordine alfabetico:

- Antonio Cangiano antonio[at]visualcsharp.it
- Antonio Vitale lampilux[at]interfree.it,
- Davide Bozza dbozza[at]avmap.it
- Davide Muzzarelli public[at]dav-muz.net
- Enrico Morelli morelli[at]cerm.unifi.it
- Paolo Mossino paolo.mossino[at]gmail.com,
- Paolo Caldana verbal[at]teppisti.it,

Un ringraziamento particolare ad Enrico Morelli morelli[at]cerm.unifi.it che ha contribuito in modo determinante al rilascio di questo manuale effettuando una minuziosa revisione.

INDICE

CAPITOLO

UNO

Introduzione

Questo manuale di riferimento descrive il linguaggio di programmazione Python. Non è inteso come un tutorial.

Mentre sto provando ad essere il più preciso possibile, ho scelto di usare un linguaggio comune anziché specificazioni formali per tutto, ad eccezione della sintassi e dell'analisi lessicale. Questo dovrebbe rendere il documento più facile da capire per il lettore medio, ma lascia adito ad ambiguità. Conseguentemente, se venite da Marte e avete provato a reimplementare Python solamente da questo documento, dovrete indovinare alcune cose e, alla prova dei fatti, probabilmente otterrete un'implementazione leggermente diversa del linguaggio. D'altro canto, se state usando Python e vi domandate in cosa consistono le regole precise circa una particolare area del linguaggio, dovreste essere in grado di trovarle qui. Se volete vedere una definizione più formale del linguaggio, potreste offrire volontariamente il vostro tempo — o inventare una macchina per clonare :-).

È pericoloso aggiungere troppi dettagli implementativi in un documento di riferimento per un linguaggio: l'implementazione può cambiare ed altre implementazioni dello stesso linguaggio possono operare in modo differente. D'altra parte, c'è attualmente solo un'implementazione di Python largamente usata (anche se ne esiste una seconda!) ed i suoi particolari capricci sono a volte degni di essere menzionati, specialmente l'implementazine impone ulteriori limitazioni. Perciò, troverete brevi "Note implementative" a sprazzi nel testo.

Ogni implementazione di Python viene fornita di un certo numeri di built-in e moduli standard. Questi non sono documentati qui, ma in un documento separato: *La libreria di riferimento di Python*. Alcuni moduli built-in vengono menzionati dove questi interagiscono in modo significativo con la definizione del linguaggio.

1.1 Notazione

Le descrizioni dell'analisi lessicale e della sintassi, utilizzano una notazione per le grammatiche BNF. Utilizzano il seguente stile di definizione:

```
name:           lc_letter (lc_letter | "_")*
lc_letter:      "a"..."z"
```

La prima riga dice che un nome, `name`, è una lettera_minuscola, `lc_letter`, seguita da una sequenza di zero o più altre lettere_minuscole, `lc_letter`, o caratteri di sottolineatura. Una '`a`' è un singolo carattere dell'alfabeto da '`a`' a '`z`'. (Queste regole aderiscono alle definizioni delle regole lessicali e grammaticali in questo documento.)

Ogni regola inizia con un nome (che è lo stesso nome definito dalla regola) e un punto. Una sbarra verticale (`|`) viene usata per separare le alternative; è l'operatore con la precedenza più bassa in questa notazione. Un asterisco (`*`) significa zero o più ripetizioni dell'oggetto precedente, un più (`+`) significa una o più ripetizioni e una frase racchiusa tra parentesi quadre (`[ ]`) significa zero o una occorrenza (in altre parole, la frase è facoltativa). Gli operatori `*` e `+` si applicano il più strettamente possibile (in termini di numero di oggetti); le parentesi tonde vengono utilizzate per raggruppare (un gruppo si comporta come un singolo oggetto). Le stringhe costanti letterali sono incluse tra caratteri di quotatura. Gli spazi hanno solo il significato di separare i token. Le regole vengono normalmente contenute in una singola riga; le righe con molte alternative possono essere formattate alternativamente facendo iniziare tutte le righe dopo la prima con una sbarra verticale.

Nelle definizioni lessicali (come nell'esempio sopra), vengono utilizzate due ulteriori convenzioni. Due caratteri

costanti letterali, separati da tre punti significano una scelta di un singolo carattere nell'intervallo indicato (estremi inclusi) di caratteri ASCII. Una frase tra parentesi angolari (`<...>`) da una descrizione informale del simbolo definito; per esempio, questo può essere utilizzato nel descrivere la nozione di carattere di controllo, se necessario.

Anche se la notazione utilizzata è pressoché la stessa, c'è una grande differenza tra il significato delle definizioni lessicali e quelle sintattiche: una definizione lessicale opera su caratteri individuali del sorgente in input, mentre una definizione sintattica opera sulla sequenza di token generati dall'analisi lessicale. Tutti gli utilizzi di BNF nel prossimo capitolo ("Analisi Lessicale") sono definizioni lessicali; quelle nei capitoli successivi sono definizioni sintattiche.

CAPITOLO

DUE

Analisi lessicale

Un programma Python viene letto da un *parser*. L'input da analizzare è una sequenza di *token*, generati dall'*analizzatore lessicale*. Questo capitolo descrive come l'analizzatore lessicale divide il file in token.

Python utilizza il set di caratteri ASCII a 7 bit per il testo del programma. Nuovo nella versione 2.3: può essere dichiarata una codifica per indicare che le stringhe costanti manifeste ed i commenti sono in una codifica diversa da ASCII. uer compatibilità con vecchie versioni, Python vi avvisa solo se trova caratteri ad 8 bit; questi avvertimenti devono essere corretti utilizzando una specifica codifica o utilizzando delle sequenza di escape (NdT: di protezione) se questi byte sono dati binari anziché caratteri.

L'insieme di caratteri a runtime dipende dal dispositivo di I/O connesso al programma, ma è generalmente un superinsieme di ASCII.

Note di compatibilità futura: si può essere tentati di assumere che l'insieme di caratteri per i caratteri ad 8 bit sia ISO Latin-1 (un superinsieme ASCII che copre la maggior parte dei linguaggi dell'ovest che usano l'alfabeto latino), ma è possibile che in futuro gli editor di testo Unicode diventino di uso comune. Questi generalmente utilizzano la codifica UTF-8, che è sempre un superinsieme ASCII, ma utilizza in modo diverso i caratteri con un valore numerico tra 128 e 255. Mentre non c'è ancora consenso su questo, è sconsigliabile dare per scontato Latin-1 o UTF-8, anche se l'implementazione corrente appare in favore di Latin-1. Questo si applica sia al codice sorgente che all'insieme di caratteri a runtime.

2.1 Struttura delle righe

Un programma Python è diviso in un certo numero di *righe logiche*.

2.1.1 Righe logiche

La fine di una riga fisica è rappresentato dal token NEWLINE. Le dichiarazioni non possono attraversare i limiti delle righe logiche ad eccezione di dove NEWLINE è ammesso dalla sintassi (per esempio tra istruzioni in "compound statements"). Una riga logica è costituita da una o più *righe fisiche* seguendo le regole implicite o esplicite di *unione tra righe*.

2.1.2 Righe fisiche

Una riga fisica termina dove si trova il terminatore di riga, secondo la convenzione della piattaforma corrente. Su UNIX, è il carattere ASCII LF (linefeed). Su Windows è la sequenza ASCII CR LF (return seguito da linefeed). Su Macintosh è il carattere ASCII CR (return).

2.1.3 Commenti

Un commento inizia con un carattere cancelletto (#) che non sia parte di una stringa costante manifesta e termina alla fine della riga fisica. Un commento indica la fine di una riga logica, a meno che non siano invocate speciali regole di unione. I commenti vengono ignorati dalla sintassi; non sono token.

2.1.4 Dichiarazioni di codifica

Se un commento nella prima o nella seconda riga di uno script Python corrisponde all'espressione regolare `coding[=:]\s*([\w-_.]+)`, questo commento viene processato come una dichiarazione di codifica; il primo gruppo di questa espressione nomina la codifica del file contenente il codice sorgente. Le forme raccomandate di questa espressione sono:

```
# -*- coding: <encoding-name> -*-
```

che viene riconosciuta anche da GNU Emacs e

```
# vim:fileencoding=<encoding-name>
```

che viene riconosciuta da VIM di Bram Moolenaar.

In aggiunta, se il primo byte del file è la maschera di byte-order di UTF-8 (`'\xef\xbb\xbf'`), il file di codifica dichiarata è UTF-8 (questo è supportato, tra gli altri, dal **notepad** di Microsoft).

Se viene dichiarata una codifica, il nome di questa codifica deve essere riconosciuta da Python. La codifica viene utilizzata per tutte le analisi lessicali, in particolare per trovare la fine di una stringa e per interpretare il contenuto di stringhe costanti Unicode. Le stringhe costanti vengono convertite in Unicode per le analisi sintattiche, poi riconvertite nella codifica originale prima che inizi l'interpretazione. La dichiarazione di codifica deve apparire sulla prima riga di codice.

2.1.5 Unione esplicita di righe

Due o più righe fisiche possono essere unite in righe logiche, utilizzando il carattere backslash (\), come di seguito: quando una riga fisica termina con un backslash che non è parte di una stringa costante o un commento, viene unita alla seguente, formando una singola riga logica, cancellando il backslash ed il seguente terminatore di riga. Per esempio:

```
if 1900 < year < 2100 and 1 <= month <= 12 \
   and 1 <= day <= 31 and 0 <= hour < 24 \
   and 0 <= minute < 60 and 0 <= second < 60:   # Per date valide
        return 1
```

Una linea che termina in backslash non può avere un commento. Un backslash non continua un commento. Un backslash non continua un token, ad eccezione delle stringhe costanti (token che non siano stringhe costanti non possono essere divisi tra righe fisiche utilizzando un backslash). Un backslash è illegale in altre posizioni in una riga, al di fuori di una stringa costante.

2.1.6 Unione implicita di righe

Le espressioni in parentesi tonde, quadre o graffe possono essere divise su più righe fisiche senza utilizzare backslash. Per esempio:

```
month_names = ['Januari', 'Februari', 'Maart',      # Questi sono i
               'April',   'Mei',      'Juni',       # nomi olandesi
               'Juli',    'Augustus', 'September',  # per i mesi
               'Oktober', 'November', 'December']   # dell'anno
```

Le righe che continuano implicitamente possono avere dei commenti. L'indentazione delle righe di continuazione non è importante. Sono ammesse righe di continuazione vuote. Non c'è un token NEWLINE tra le righe di con-

tinuazione implicite. Le righe di continuazione implicite possono capitare anche in stringhe con tripla quotatura (vedete sotto); in questo caso non possono contenere commenti.

2.1.7 Righe vuote

Una riga logica che contiene solo spazi, tabulazioni, avanzamenti di pagina e facoltativamente un commento, viene ignorata (cioè non viene generato un token NEWLINE). Durante sessioni interattive di istruzioni in input, la gestione delle righe vuote può differire a seguito dell'implementazione del ciclo di lettura-valutazione-stampa. Nell'implementazione standard, una riga logica interamente vuota (cioè priva di caratteri di spaziatura o commenti) termina un'istruzione multiriga.

2.1.8 Indentazione

I caratteri di spaziatura (spazi e tab) all'inizio di una riga logica, vengono utilizzati per computare il livello di indentazione di questa riga; in pratica determina il raggruppamento delle istruzioni.

Come prima cosa, i tab vengono sostituiti (da sinistra a destra) da un numero di spazi variabile da uno ad 8, in modo che il numero totale di caratteri incluso la sostituzione sia un multiplo di otto (questo si intende come la regola utilizzata dai sistemi UNIX). Il numero totale di spazi che precedono il primo carattere che non sia di spaziatura determinano quindi l'indentazione della riga. L'indentazione non può essere divisa su più righe fisiche utilizzando i backslash; gli spazi fino al primo backslash determinano il livello di indentazione.

Note di compatibilità su diverse piattaforme: a causa della natura degli editor di testo sui sistemi non UNIX, non è consigliato utilizzare un misto di spazi e tab per indentare una singola riga di codice sorgente. Si deve anche notare che differenti piattaforme possono limitare il numero massimo di livelli di indentazione.

Un carattere di avanzamento di pagina può essere presente all'inizio di una riga; verrà ignorato per il calcolo dell'indentazione. Caratteri di avanzamento di pagina che occorrano altrove avranno un effetto non specificato (per esempio, possono azzerare il contatore degli spazi).

Livelli di indentazione di righe consecutive vengono utilizzati per generare i token INDENT e DEDENT, utilizzando uno stack, come di seguito.

Prima che venga letta la prima riga del file, un singolo zero viene messo sullo stack; questo non verrà mai rimosso. Il numero messo nello stack sarà sempre strettamente crescente, dal fondo alla cima. All'inizio di ogni riga logica, il livello di indentazione della riga viene confrontato con la cima dello stack. Se uguale non accade niente. Se è maggiore, viene messo nello stack e viene generato un token INDENT. Se minore, *deve* essere uno dei numeri presenti nello stack; tutti i numeri maggiori vengono eliminati dallo stack e, per ognuno di questi, viene generato un token DEDENT. Alla fine del file viene generato un token DEDENT per ogni numero rimasto nello stack che sia maggiore di zero.

Ecco un esempio di una porzione di codice python correttamente indentato, anche se fuorviante:

```
def perm(l):
        # Compute the list of all permutations of l
    if len(l) <= 1:
                  return [l]
    r = []
    for i in range(len(l)):
             s = l[:i] + l[i+1:]
             p = perm(s)
             for x in p:
              r.append(l[i:i+1] + x)
    return r
```

L'esempio seguente mostra vari errori di indentazione:

```
 def perm(l):                       # errore: prima linea indentata
for i in range(len(l)):             # errore: non indentato
    s = l[:i] + l[i+1:]
        p = perm(l[:i] + l[i+1:])   # errore: indentazione inaspettata
        for x in p:
                r.append(l[i:i+1] + x)
            return r                # errore: deindentazione inconsistente
```

In verità, i primi tre errori vengono intercettati dal parser; solo l'ultimo viene intercettato dall'analizzatore lessicale — l'indentazione di `return r` non corrisponde ad un livello estratto dallo stack.

2.1.9 Spaziature tra i token

Ad eccezione dell'inizio di una riga logica o una stringa costante, i caratteri di spaziatura (spazi, tab e avanzamento di pagina) possono essere utilizzati in modo intercambiabile per separare i token. Una spaziatura è necessaria tra due token solo se la loro concatenazione possa altrimenti essere interpretata come un token differente (per esempio ab è un token, ma a e b sono due token).

2.2 Altri token

A parte NEWLINE, INDENT e DEDENT, esistono le seguenti categorie di token: *identifiers*, *keywords*, *literals* , *operators* e *delimiters* (NdT: identificatori, parole chiave, costanti manifeste, operatori e delimitatori). I caratteri di spaziatura (a parte i terminatori di riga, come discusso precedentemente) non sono token, ma servono per delimitare i token. Quando esiste un'ambiguità, un token comprende la maggior porzione di stringa che possa formare un token legale, quando letto da sinistra a destra.

2.3 Identificatori e keywords

Gli identificatori (anche chiamati *nomi*) sono descritti dalle seguenti definizioni lessicali:

```
identifier  ::=  (letter|_) (letter | digit | _)*
letter      ::=  lowercase | uppercase
lowercase   ::=  a...z
uppercase   ::=  A...Z
digit       ::=  0...9
```

Gli identificatori possono essere di lunghezza arbitraria. La distinzione tra lettere maiuscole e minuscole è significativa.

2.3.1 Parole chiave

I seguenti identificatori vengono utilizzati come parole riservate o *parole chiave* del linguaggio e non possono essere utilizzati come identificatori ordinari. Devono essere scritti esattamente come sono scritti qui:

```
and       del       for       is        raise
assert    elif      from      lambda    return
break     else      global    not       try
class     except    if        or        while
continue  exec      import    pass      yield
def       finally   in        print
```

Notate che anche se l'identificatore `as` può essere utilizzato come parte della sintassi di una dichiarazione di `import`, non è attualmente una parola riservata.

Nelle prossime versioni di Python, gli identificatori `as` e `None` diventeranno parole chiave.

2.3.2 Classi riservate di identificatori

Alcune classi di identificatori (a parte le parole chiave) hanno un significato speciale. Queste classi vengono identificate come un modello caratterizzato da un carattere di sottolineatura all'inizio o alla fine:

`_*` Non importato da '`from` *module* `import *`'. L'identificatore speciale '_' viene utilizzato nell'interprete interattivo per immagazzinare il risultato dell'ultima valutazione; viene immagazzinato nel modulo `__builtin__`. Quando non siamo in modalità interattiva, '_' non ha significati speciali e non viene definito. Vedere la sezione 6.12, "L'istruzione `import`".

Note: Il nome '_' viene spesso utilizzato in congiunzione con l'internazionalizzazione; fate riferimento alla documentazione del modulo `gettext` per maggiori informazioni su questa convenzione.

`__*__` Nome definito dal sistema. Questi nomi vengono definiti dall'interprete e la sua implementazione (includendo la libreria standard); le applicazioni non dovrebbero definire nomi supplementari utilizzando questa convenzione. L'insieme di nomi di questa categoria, definiti da Python, potrebbe estendersi nelle prossime versioni. Vedere la sezione 3.3 "Nomi speciali dei metodo".

`__*` Nomi privati di classe. I nomi in questa categoria, quando utilizzati nel contesto di una definizione di classe, vengono riscritti per evitare interferenze (NdT: name clashes) con gli attributi "privati" delle classi base e derivate. Vedere la sezione 5.2.1, "Identificatori (Nomi)".

2.4 Literals

Le costanti manifeste sono le notazioni per i valori costanti di alcuni tipi built-in.

2.4.1 Stringhe costanti manifeste

Le stringhe costanti vengono descritte dalle seguenti definizioni lessicali:

```
stringliteral    ::=  [stringprefix](shortstring | longstring)
stringprefix     ::=  r | u | ur | R | U | UR | Ur | uR
shortstring      ::=  ' shortstringitem* ' | '' shortstringitem* ''
longstring       ::=  ''' longstringitem* '''
                      | '"' longstringitem* '"'
shortstringitem  ::=  shortstringchar | escapeseq
longstringitem   ::=  longstringchar | escapeseq
shortstringchar  ::=  <ogni carattere ASCII con l'eccezione di \, fine riga o quotatu
longstringchar   ::=  <ogni carattere ASCII con l'eccezione di \»
escapeseq        ::=  \ <ogni carattere ASCII>
```

Una restrizione sintattica non indicata da queste produzioni è che non sono ammesse spaziature tra `stringprefix` ed il resto della stringa costante.

In Italiano: le stringhe costanti possono essere racchiuse in caratteri di quotatura singola (') o doppia (). Possono anche essere racchiusi in gruppi di *stringhe composte da tuple di tre elementi di quotatura* (NdT: queste sono in genere indicate come "triple-quoted strings"). Il carattere backslash (\) viene utilizzato come carattere di escape, che altrimenti avrebbe un significato speciale, come il fine riga, la stessa backslash o il carattere di quotatura. Le stringhe costanti manifeste possono facoltativamente avere come prefisso le lettere '`r`' o '`R`'; queste stringhe sono dette *stringhe raw* e utilizzano differenti regole per l'interpretazione dei backslash nelle sequenze di escape. Un prefisso '`u`' o '`U`' rende la stringa una stringa Unicode. Le stringhe Unicode vengono definite dal consorzio Unicode e ISO 10646. Alcune sequenze di escape aggiuntive, descritte sotto, sono disponibili nelle stringhe Unicode. I due caratteri di prefisso possono essere combinati; in questo caso, '`u`' può apparire prima di '`r`'.

Nelle stringhe a quotatura tripla, vengono ammessi e mantenuti i fine riga e le quotature singole senza escape, ad eccezione del fatto che tre quotature senza escape in una riga terminano la stringa. (Un carattere di "quotatura" è il carattere utilizzato per aprire la stringa, cioè sia ' che).

A meno che il prefisso 'r' o 'R' non sia presente, le sequenze di escape in una stringa vengono interpretate in accordo a regole simili a quelle utilizzate nello standard C. Le sequenze di escape riconosciute sono:

Sequenza di escape	**Significato**	**Note**
newline	Ignorato	
\\	Backslash (\)	
\'	Carattere di quotatura singola (')	
\	Carattere di quotatura doppia ()	
\a	ASCII Bell (BEL)	
\b	ASCII Backspace (BS)	
\f	ASCII Formfeed (FF)	
\n	ASCII Linefeed (LF)	
\N{*name*}	Carattere chiamato *name* nel database Unicode (solamente in Unicode)	
\r	ASCII Carriage Return (CR)	
\t	ASCII Tab orizzontale (TAB)	
\u*xxxx*	Carattere con valore esadecimale a 16 bit *xxxx* (valido solo per Unicode)	(1)
\U*xxxxxxxx*	Carattere con valore esadecimale a 32 bit *xxxxxxxx* (valido solo per Unicode)	(2)
\v	ASCII Tab verticale (VT)	
ooo	Carattere ASCII con valore ottale *ooo*	(3)
\x*hh*	Carattere ASCII con valore esadecimale *hh*	(4)

Note:

(1) Unità di codice individuali che formano parte di un surrogato di una coppia, possono essere codificate utilizzando questa sequenza di escape.

(2) Ciascun carattere Unicode può essere rappresentato in questo modo, ma i caratteri al di fuori del Basic Multilingual Plane (BMP) verranno codificati per utilizzare il surrogato di una coppia se Python è compilato per utilizzare unità di codice a 16 bit (il valore predefinito). Le unità di codice individuali che formano parti del surrogato di una coppia possono essere codificate utilizzando questa sequenza di escape.

(3) Come nello Standard C, fino a tre cifre ottali vengono accettate.

(4) Al contrario dello standard C, vengono accettate almeno 2 cifre esadecimali.

Al contrario dello standard C, tutte le sequenze di escape non riconosciute vengono lasciate nella stringa senza apportare modifiche, cioè *il backslash viene lasciato nella stringa*. (Questo comportamento è utile in fase di debug: se una sequenza di escape viene digitata male, dall'output risultante si riconosce più facilmente come errata). È anche importante notare che le sequenze di escape segnate come "(valide solamente per Unicode)" nella tabella sopra, finiscono nella categoria degli escape non riconosciuti per le stringhe costanti non Unicode.

Quando il prefisso 'r' o 'R' è presente, un carattere che segue un backslash viene incluso nella stringa senza subire modifiche e tutte i backslash vengono lasciati nella stringa. Per esempio, la stringa costante r\n consiste di due caratteri: un backslash e una 'n' minuscola. I caratteri di quotature possono essere soggetti ad escape con un backslash, ma il backslash rimane nella stringa; per esempio, r\" è una stringa costante valida che consiste di due caratteri: un backslash e un carattere di quotatura doppio; r\ non è una stringa costante valida (anche una stringa raw non può terminare con un numero dispari di backslash). Specificamente, *una stringa raw non può terminare con un singolo backslash* (poiché il backslash farebbe l'escape dell'ultimo carattere di quotatura). Notare anche che un singolo backslash seguito da un fine riga viene interpretato come se questi due caratteri facessero parte della stringa, *non* come una continuazione di riga.

Quando un prefisso 'r' o 'R' viene usato in congiunzione con un prefisso 'u' o 'U', la sequenza di escape \uXXXX viene elaborata mentre tutte gli altri backslash vengono lasciati invariati nella stringa. Per esempio, la stringa costante ur\u0062\n consiste di tre caratteri Unicode: 'LATIN SMALL LETTER B', 'REVERSE SOLIDUS' e 'LATIN SMALL LETTER N'. Si può effettuare l'escape di un backslash con un'altro backslash; comunque, entrambi rimangono nella stringa. Come risultato, le sequenze di escape \uXXXX vengono solo riconosciute quando c'è un numero dispari di backslash.

2.4.2 Concatenazione di stringhe costanti manifeste

Stringhe costanti manifeste multiple ed adiacenti (delimitate da caratteri di spaziatura), che possibilmente utilizzano differenti convenzioni di quotatura, sono ammesse ed hanno lo stesso significato di una concatenazione. Perciò `hello 'world'` è equivale ad `helloworld`. Questa funzionalità può essere utilizzata per ridurre il numero di backslash necessari, per suddividere stringhe lunghe in modo conveniente, attraverso lunghe righe o anche per aggiungere commenti ad alcune parti della stringa; per esempio:

```
re.compile("[A-Za-z_]"       # lettere o sottolineature
           "[A-Za-z0-9_]*"   # lettere, numeri o sottolineature
          )
```

Notate che questa funzionalità viene definita a livello sintattico, ma implementata al momento della compilazione. L'operatore '+' deve essere utilizzato per concatenare stringhe a runtime. Notate anche che la concatenazione tra stringhe costanti manifeste può utilizzare diversi stili di quotatura per ogni componente (anche mischiando stringhe raw e stringhe a quotatura tripla).

2.4.3 Costanti numeriche

Ci sono quattro tipi di costanti manifeste numeriche: interi, interi long, numeri in virgola mobile e numeri immaginari. Non ci sono costanti complesse (i numeri complessi possono essere formati sommando un numero reale ed un numero immaginario).

Si noti che le costanti numeriche non includono il segno; una frase come `-1` è in realtà un'espressione composta dall'operatore unario '`-`' e dalla costante numerica `1`.

2.4.4 Costanti di tipo intero e long

Le costanti di tipo intero e long vengono descritte dalle seguenti definizioni lessicali:

```
longinteger     ::=  integer (l | L)
integer         ::=  decimalinteger | octinteger | hexinteger
decimalinteger  ::=  nonzerodigit digit* | 0
octinteger      ::=  0 octdigit+
hexinteger      ::=  0 (x | X) hexdigit+
nonzerodigit    ::=  1...9
octdigit        ::=  0...7
hexdigit        ::=  digit | a...f | A...F
```

Anche se si puó utilizzare sia il suffisso '`l`' che '`L`' per indicare una costante long, è altamente raccomandato di utilizzare sempre '`L`', poiché la lettera '`l`' è troppo simile alla cifra '`1`'.

Le costanti intere che sono maggiori del più grande numero intero rappresentabile (per esempio 2147483647 quando si utilizza un'aritmetica a 32 bit) vengono accettate come se fossero costanti long. [1] Non c'é limite alla lunghezza delle costanti long, a parte la limitazione dovuta alla dimensione della memoria disponibile.

Alcuni esempi di costanti numeriche intere (prima riga) e long (seconda e terza riga):

```
7     2147483647                        0177
3L    79228162514264337593543950336L    0377L   0x100000000L
      79228162514264337593543950336             0xdeadbeef
```

[1] Nelle versioni di Python precedenti alla 2.4, le costanti numeriche ottali ed esadecimali nell'intervallo subito sopra il più grande intero rappresentabile, ma inferiore al più grande numero senza segno a 32 bit (su macchine con aritmetica a 32 bit), 4294967296, vengono considerate come il numero intero negativo ottenuto sottraendo 4294967296 al loro valore senza segno.

2.4.5 Costanti numeriche in virgola mobile

Le costanti numeriche in virgola mobile vengono descritte dalle seguenti definizioni lessicali:

```
floatnumber    ::=  pointfloat | exponentfloat
pointfloat     ::=  [intpart] fraction | intpart .
exponentfloat  ::=  (intpart | pointfloat) exponent
intpart        ::=  digit+
fraction       ::=  .  digit+
exponent       ::=  (e | E) [+ | -] digit+
```

Si noti che la parte intera e l'esponente di un numero in virgola mobile possono sembrare come interi in notazione ottale, ma vengono interpretati utilizzando la base 10. Per esempio, '`077e010`' è legale e denota lo stesso numero come '`77e10`'. La gamma di costanti numeriche in virgola mobile ammessa dipende dall'implementazione. Alcuni esempi di costanti in virgola mobile:

```
3.14    10.    .001    1e100    3.14e-10    0e0
```

Si noti che le costanti numeriche non includono il segno; una frase come `-1` è in realtà un'espressione composta dell'operatore unario `-` e della costante numerica `1`.

2.4.6 Costanti numeriche immaginarie

Le costanti numeriche immaginarie vengono descritte dalla seguente definizione lessicale:

```
imagnumber  ::=  (floatnumber | intpart) (j | J)
```

Una costante immaginaria restituisce un numero complesso con una parte reale di 0.0. I numeri complessi vengono rappresentati come una coppia di numeri in virgola mobile ed hanno le stesse restrizioni sulla gamma di valori possibili. Per creare un numero complesso con una parte reale diversa da zero, sommarvi un numero in virgola mobile, per esempio `(3+4j)`. Alcuni esempi di costanti numeriche immaginarie:

```
3.14j   10.j    10j     .001j   1e100j  3.14e-10j
```

2.5 Operatori

I seguenti token sono operatori:

```
+       -       *       **      /       //      %
<<      >>      &       |       ^       ~
<       >       <=      >=      ==      !=      <>
```

Gli operatori di confronto `<>` e `!=` sono forme alternative dello stesso operatore. La forma preferita è `!=`, mentre `<>` è obsoleta.

2.6 Delimitatori

I seguenti token operano come delimitatori grammaticali:

```
(       )       [       ]       {       }
,       :       .       `       =       ;
+=      -=      *=      /=      //=     %=
&=      |=      ^=      >>=     <<=     **=
```

Il punto occorre anche nelle costanti numeriche reali ed immginarie. Una sequenza di tre punti ha un significato speciale come un'ellissi in una fetta. La seconda metà della lista, gli operatori di assegnamento e modifica, operano lessicalmente come delimitatori, ma effettuano anche delle operazioni.

I seguenti caratteri ASCII stampabili hanno un significato speciale come parte di altri token o sono altrimenti significativi per l'analizzatore lessicale:

```
'       "       #       \
```

I seguenti caratteri ASCII stampabili non vengono utilizzati in Python. La loro presenza fuori da stringhe costanti manifeste e commenti è un errore:

```
@       $       ?
```

CAPITOLO

TRE

Modello dei dati

3.1 Oggetti, valori e tipi

Gli *oggetti* sono delle astrazioni di Python per i dati. Tutti i dati in Python vengono rappresentati da oggetti o da relazioni tra oggetti. (In un certo senso, e conformemente al modello di Von Neumann di "stored program computer", anche il codice viene rappresentato come un oggetto).

Ogni oggetto ha un'identità, un tipo ed un valore. L'*identità* di un oggetto non cambia mai dopo che è stato creato; potete pensare a questa come l'indirizzo di memoria dell'oggetto. L'operatore '`is`' confronta l'identità di 2 oggetti; la funzione `id()` restituisce un intero che rappresenta l'identità dell'oggetto (attualmente implementato come il suo indirizzo). Il *tipo* di un oggetto è anch'esso immutabile.[1] Il tipo di un oggetto determina le operazioni che l'oggetto supporta (per esempio, "è possibile determinarne la lunghezza?") e definisce anche i possibili valori per gli oggetti di quel tipo. La funzione `type()` restituisce il tipo dell'oggetto (che è a sua volta un oggetto). Il *valore* di alcuni oggetti può cambiare. Gli oggetti il cui valore può cambiare vengono detti *mutabili*; gli oggetti il cui valore non può cambiare vengono detti *immutabili*. (Il valore di un oggetto contenitore immutabile che contiene un riferimento ad un oggetto mutabile può cambiare quando cambia il valore di quest'ultimo; comunque il contenitore viene ancora considerato immutabile, poiché la collezione di oggetti che contiene non può essere cambiata. Perciò immutabile non è strettamente equivalente ad un valore non modificabile, la differenza è più sottile). La mutabilità di un oggetto viene determinata dal suo tipo; per esempio numeri, stringhe e tuple sono immutabili, mentre dizionari e liste sono mutabili.

Gli oggetti non vengono mai esplicitamente distrutti; comunque, quando diventano irrangiungibili, possono essere distrutti tramite il Garbage collector. All'implementazione viene concesso di posticipare il garbage collector oppure ometterlo totalmente, il modo in cui opera il Garbage Collector è una questione di qualità dell'implementazione, finché non viene raccolto un oggetto che è ancora raggiungibile. Nota implementativa: l'implementazione corrente utilizza uno schema di conteggio dei riferimenti che (facoltativamente) ritarda l'intercettazione di oggetti non più referenziati, ma non garantisce di intercettare le dipendenze cicliche. Vedere la *Libreria Python di riferimento* per informazioni sul controllo del garbage collector per dipendenze cicliche.

Notare che l'utilizzo dell'implementazione delle facility di tracing e debugging può mantenere vivi oggetti che normalmente sarebbero riciclabili. Notare anche che l'intercettazione di un'eccezione con l'istruzione '`try...except`' può tenere in vita un oggetto.

Alcuni oggetti contengono riferimenti a risorse "esterne" come file aperti o finestre. Bisogna comprendere che queste risorse vengono liberate quando l'oggetto viene riciclato tramite il GC, ma poiché nulla garantisce che il GC intervenga, questi oggetti forniscono anche un metodo esplicito per rilasciare le risorse esterne, generalmente un metodo `close()`. Ai programmi è caldamente raccomandato chiudere questi oggetti. L'istruzione '`try...finally`' fornisce un modo conveniente per farlo.

Alcuni oggetti contengono riferimenti ad altri oggetti; questi vengono chiamati *contenitori*. Sono esempi di contenitori tuple, liste e dizionari. Questi riferimenti sono parte del valore del contenitore. In molti casi, quando parliamo del valore del contenitore, implichiamo il valore, non l'identità degli oggetti contenuti; comunque, quando parliamo di mutabilità di un contenitore, viene implicata solo l'identità dell'oggetto immediatamente contenuto.

[1] Da Python 2.2, è iniziato un graduale lavoro di unificazione di tipi e classi, che rendono alcune delle affermazioni in questo manuale non accurate al 100% ed incomplete: per esempio, adesso *è* possibile, in alcuni casi, cambiare il tipo di un oggetto, sotto alcune condizioni controllate. Finché questo manuale non subirà un'intensa revisione, deve essere considerato come autoritativo solo circa le "classi classiche", che sono ancora il comportamento predefinito, per ragioni di compatibilità, in Python 2.2 e 2.3.

Perciò, se un contenitore immutabile (come una tupla) contiene un riferimento ad un oggetto mutabile, il suo valore può cambiare se cambia questo oggetto mutabile.

I tipi coinvolgono quasi tutti gli aspetti del comportamento di un oggetto. Anche l'importanza dell'identità degli oggetti ne viene coinvolta in qualche modo: per i tipi immutabili, le operazioni che computano nuovi valori possono in realtà restituire un riferimento ad un oggetto esistente con lo stesso valore, mentre per gli oggetti mutabili non è ammesso. Per esempio, dopo '`a = 1; b = 1`', `a` e `b` possono fare riferimento allo stesso oggetto con il valore uno, a seconda dell'implementazione, ma dopo '`c = []; d = []`' è garantito che `c` e `d` faranno riferimento a due liste differenti, uniche e vuote (notare invece che '`c = d = []`' assegna lo stesso oggetto a `c` e `d`).

3.2 La gerarchia dei tipi standard

Di seguito una lista di tutti i tipi che vengono costruiti in Python. I moduli di estensione (scritti in C, Java o altri linguaggi, in funzione dell'implementazione) possono definire ulteriori tipi. Le future versioni di Python potranno aggiungere tipi alla gerarchia dei tipi (per esempio numeri razionali, array di interi memorizzati efficientemente, etc.).

Alcuni dei tipi descritti contengono un paragrafo che elenca gli 'attributi speciali'. Questi sono attributi che forniscono accesso all'implemenatzione e non sono intesi per l'uso generale. La loro definizione potrà cambiare in futuro.

None Questo tipo ha un singolo valore. C'è un solo oggetto con questo valore. Questo oggetto è accessibile attraverso il nome built-in `None`. Viene utilizzato per indicare l'assenza di un valore in molte situazioni, per esempio, viene restituito dalle funzioni che non restituiscono esplicitamente qualcosa. Il suo valore di verità è "Falso".

NotImplemented Questo tipo ha un singolo valore. C'è un solo oggetto con questo valore. Questo oggetto è accessibile attraverso il nome built-in `NotImplemented`. I metodi numerici ed i metodi di confronto complessi possono restituire questo valore se non implementano l'operazione per l'operando fornito. (L'interprete proverà quindi a riflettere l'operazione, o altri metodi di fallback, a seconda dell'operatore in questione.) Il suo valore di verità è "Vero".

Ellipsis Questo tipo ha un singolo valore. C'è un solo oggetto con questo valore. Questo oggetto è accessibile con il nome built-in `Ellipsis`. Viene utilizzato per indicare la presenza di '`...`' nella sintassi di una sequenza. Il suo valore di verità è "Vero".

Numbers Questi vengono creati da costanti numeriche e restuituiti come risultato di operazioni aritmetiche o funzioni aritmetiche built-in. Gli oggetti numerici sono immutabili; una volta creati, il loro valore non verrà mai cambiato. I numeri Python sono ovviamente strettamente correlati con i numeri matematici, ma soggetti alle limitazioni numeriche della rappresentazione su computer.

Python distingue tra interi, numeri in virgola mobile e numeri complessi:

Integers Questi rappresentano elementi dall'insieme matematico di tutti i numeri:

Ci sono tre tipi di interi:

Interi Plain Questi rappresentano numeri nell'intervallo da -2147483648 a 2147483647. (L'intervallo può essere maggiore su architetture con una dimensione più grande, ma mai minore.) Quando il risultato di un'operazione cade al di fuori di questo intervallo, il risultato viene normalmente restituito come un intero di tipo long (in alcuni casi, viene invece sollevata l'eccezione `OverflowError`). Per scopi volti ad operazioni di scorrimento o per maschere, si assume che gli interi abbiano una notazione binaria in complemento a 2, utilizzando 32 bit o più e nascondendo alcuni bit all'utente (cioé, tutti i 4294967296 differenti modelli di bit corrispondono a differenti valori).

Interi Long Questi rappresentano i numeri in un intervallo illimitato, soggetti solamente alla memoria (virtuale) disponibile. Per scopi volti ad operazioni di scorrimento o per maschere, si assume una rappresentazione binaria e i numeri negativi vengono rappresentati in una variante della notazione in complemento a due che dà l'illusione di una stringa infinita di bit con segno che si estende verso sinistra.

Booleani Questi rappresentano i valori di verità "Vero" e "Falso". I due oggetti che rappresentano "Vero" e "Falso" sono gli unici oggetti booleani. Il tipo Boolean è un sottotipo degli interi plain ed i valori di Boolean si comportano come i valori 0 e 1 rispettivamente, in quasi tutti i contesti, ad eccezione di quando vendono convertiti in stringa, dove vendono restituite, rispettivamente, le stringhe `False` e `True`.

Il risultato per la rappresentazione degli interi è intesa per fare l'interpretazione più utile delle operazioni di scorrimento e per le maschere che riguardano i numeri negativi e le minori sorprese possibili quando si effettua un cambiamento tra il dominio degli interi plain e gli interi long. Ogni operazione, ad eccezione dello scorrimento a sinistra, se genera un risultato nel dominio degli interi plain senza generare overflow, genera lo stesso risultato nel dominio degli interi long o quando si utilizzano operandi misti.

Numeri in virgola mobile Questi rappresentano i numeri in virgola mobile in precisione doppia (rappresentazione finita disponibile su un computer). Siete lasciati alla misericordia dell'architettura sottostante (e delle implementazioni C e Java) per l'intervallo accettato e la gestione dell'overflow. Python non supporta i numeri in virgola mobile in precisione singola; il guadagno in utilizzo di processore e memoria, che è in genere il motivo per utilizzarli, viene sminuito dall'overhead di utilizzare oggetti in Python, perciò non c'è ragione di complicare il linguaggio con due tipi di numeri in virgola mobile.

Numeri complessi Rappresentano numeri complessi come una coppia di numeri in virgola mobile in precisione doppia. Valgono gli stessi avvertimenti dati per i numeri in virgola mobile. Le parti reali ed immaginarie di un numero complesso `z` possono essere recuperate attraverso degli attributi in sola lettura `z.real` e `z.imag`.

Sequenze Rappresentano degli insiemi ordinati e finiti di elementi indicizzati tramite numeri non negativi. La funzione built-in `len()` restituisce il numero di elementi di una sequenza. Quando la lunghezza di una sequenza è *n*, l'indice contiene i numeri 0, 1, ..., *n*-1. L'elemento *i* di una sequenza *a* viene selezionata tramite *a*`[`*i*`]`.

Le sequenza supportano anche l'affettamento: *a*`[`*i*`:`*j*`]` seleziona tutti gli elementi con indice *k* tale che vari `<=` *k* < *j*. Quando utilizzata come un'espressione, una fetta è una sequenza dello stesso tipo. Questo implica che l'insieme degli indici viene rinumerato in modo che inizi da 0.

Alcune sequenze supportano anche l'"affettazione estesa" con un terzo parametro "passo": *a*`[`*i*`:`*j*`:`*k*`]` seleziona tutti gli elementi di *a* con un indice *x* dove *x* `=` *i* `+` *n*`*`*k*, *n* `>= 0` e *i* `<=` *x* < *j*.

Le sequenze si distinguono in base alla mutabilità:

Sequenze immutabili Un oggetto di una sequenza di tipo immutabile non può cambiare una volta che è stato creato. (Se l'oggetto contiene un riferimento ad un'altro oggetto, quest'ultimo può essere mutabile e può cambiare; comunque, la collezione di oggetti direttamente referenziati da un'oggetto immutabile non può cambiare.)

I seguenti tipi sono sequenze immutabili:

Stringhe Gli elementi di una stringa sono caratteri. Non c'è un tipo separato per i caratteri; un carattere viene rappresentato da una stringa di un elemento. I caratteri rappresentano (almeno) un byte di 8 bit. Le funzioni built-in `chr()` e `ord()` effettuano le conversioni tra i caratteri e degli interi non negativi che rappresentano il valore del byte. I byte con i valori 0-127 rappresentano generalmente i valori ASCII corrispondenti, ma l'interpretazione dei valori viene lasciata al programma. Il tipo di dato stringa viene anche utilizzato per rappresentare array di byte, per esempio, per gestire i dati letti da un file.
(Nei sistemi in cui l'insieme di caratteri nativo non è ASCII, le stringhe possono utilizzare EBCDIC nella loro rappresentazione interna e le funzioni `chr()` ed `ord()` implementano una mappa tra ASCII e EBCDIC ed il confronto tra stringhe mantiene l'ordine ASCII. (O forse qualcuno può proporre una regola migliore?)

Unicode Gli elementi di un oggetto Unicode sono unità di codice Unicode. Un'unità di codice Unicode viene rappresentata da n oggetto unicode di un elemento e può contenere un valore a 16 o 32 bit che rappresenta un ordinale Unicode (il massimo valore per gli ordinali Unicode viene indicato in `sys.maxunicode` e dipende da come è stato configurato Python al momento della compilazione). Negli oggetti Unicode possono essere presenti delle coppie surrogate e verranno segnalate come due elementi separati. Le funzioni built-in `unichr()` ed `ord()` convertono tra

le unità di codice ed interi non negativi rappresentanti gli ordinali Unicode, come definito nello Standard Unicode 3.0. Le conversioni da e verso le altre codifiche sono possibili attraverso i metodi di codifica `encode` degli oggetti Unicode e la funzione built-in `unicode()`..

Tuple Gli elementi di una tupla sono oggetti Python arbitrari. Le tuple di due o più elementi vengono formattate in una lista di espressioni separate da virgole. Una tupla di un elemento (un 'singleton') può essere formato aggiungendo una virgola ad un'espressione (un'espressione da sola non crea una tupla, poiché le parentesi devono essere utilizzabili per raggruppare le espressioni). Una tupla vuota può essere resa da una coppia vuota di parentesi.

Sequenze mutabili Le sequenze mutabili possono essere modificate dopo la loro creazione. Le notazioni subscription e slice possono essere utilizzate come obiettivi di un assegnamento o di un'istruzione `del` (cancella).

C'è attualmente un solo tipo intrinseco di sequenze mutabili:

Liste Gli elementi di una lista sono oggetti Python arbitrari. Le liste sono formate da una lista di espressioni separate da virgola in una coppia di parentesi quadre. (Notare che non c'è un caso speciale per le liste di lunghezza 0 o 1).

Il modulo di estensione `array` fornisce un'esempio aggiuntivo di sequenze mutabili.

Mappe Rappresentano un insieme finito di oggetti indicizzati da un insieme arbitrario di indici. La notazione di subscription `a[k]` seleziona l'elemento indicizzato `k` dalla mappa `a`; questo può essere utilizzato nelle espressioni come obiettivo di un assegnamento o di un'istruzione `del` (cancella). La funzione built-in `len()` restituisce il numero di elementi in una mappa.

C'è attualmente un solo tipo intrinseco di mappe:

Dizionari Rappresentano un insieme finito di oggetti indicizzati da valori praticamente arbitrari. Gli unici tipi di valori non accettabili come chiavi sono i valori contenenti liste o dizionari o altri oggetti di tipo mutabile che vengono confrontati per valore piuttosto che per identità, la ragione di questo è che l'implementazione efficiente dei dizionari richiede un valore di hash per la chiave costante nel tempo. I tipi numerici utilizzati come chiavi obbediscono alle normali regole per il confronto numerico: se due numeri sono uguali (per esempio `1` e `1.0`) possono essere usati in modo intercambiabile per indicizzare lo stesso elemento del dizionario.

I dizionari sono mutabili; possono essere creati con la notazione `{...}` (si veda la sezione 5.2.5, "Visualizzazione dei dizionari").

I moduli di estensione `dbm`, `gdbm`, `bsddb` forniscono esempi aggiuntivi di mappe.

Tipi eseguibili Sono i tipi a cui si può applicare l'operazione di chiamata a funzione (si veda la sezione 5.3.4, "Chiamate"):

Funzioni definite dall'utente Un oggetto funzione definita dall'utente viene creato dalla definizione di funzione (si veda la sezione 7.5, "Definizioni di funzione"). Deve essere chiamato con una lista di argomenti che contiene lo stesso numero di oggetti della lista formale di parametri ammessi per la funzione.

Attributi speciali: `func_doc` o `__doc__` è la stringa di documentazione (docstring) della funzione, o `None` se non disponibile; `func_name` o `__name__` è il nome della funzione; `__module__` è il nome del modulo in cui la funzione è stata definita o `None` se non disponibile; `func_defaults` è una tupla che contiene i valori predefiniti per gli argomenti che hanno un valore predefinito oppure `None` se nessun argomento ha un valore predefinito; `func_code` è un oggetto di tipo codice che rappresenta il codice compilato del corpo della funzione; `func_globals` è (un riferimento a) il dizionario che mantiene le variabili globali della funzione — questo definisce lo spazio dei nomi globale del modulo in cui la funzione viene definita; `func_dict` o `__dict__` contiene lo spazio dei nomi che sostiene gli attributi arbitrari di funzione; `func_closure` è `None` oppure una tupla di celle che contengono le associazioni per le variabili libere della funzione.

Di queste, `func_code`, `func_defaults`, `func_doc`/`__doc__` e `func_dict`/`__dict__` possono essere scrivibili; le altre non possono mai cambiare. Informazioni aggiuntive circa la definizione della funzione, possono essere recuperate dal suo oggetto di tipo codice; si veda la descrizione dei tipi interni sotto.

Metodi definiti dall'utente Un oggetto di tipo metodo definito dall'utente combina una classe, un'istanza di classe (o `None`) ed un qualunque oggetto chiamabile (normalmente una funzione definita dall'utente).

Attributi speciali in sola lettura: `im_self` è l'oggetto che rappresenta l'istanza della classe; `im_func` è l'oggetto che rappresenta la funzione; `im_class` è la classe di `im_self` per i metodi associati (bound) o la classe a cui chiedere il metodo per i metodi non associati (unbound); `__doc__` è la documentazione del metodo (lo stesso di `im_func.__doc__`); `__name__` è il nome del metodo (lo stesso di `im_func.__name__`); `__module__` è il nome del modulo in cui il metodo è stato definito o `None` se non disponibile. Modificato nella versione 2.2: `im_self` referenzia la classe che definisce il metodo.

I metodi supportano anche l'accesso (ma non l'impostazione) agli argomenti arbitrari della funzione nell'oggetto sottostante che rappresenta la funzione.

I metodi definiti dall'utente possono essere creati quando si recupera l'attributo di una classe (forse tramite un'istanza di quella classe), se questo attributo è una funzione definita dall'utente, un metodo definito dall'utente non associato o un metodo di classe. Quando l'attributo è un metodo definito dall'utente, un nuovo oggetto di tipo metodo viene creato solo se la classe da cui viene recuperato è la stessa o deriva dalla stessa classe o dalla classe immagazzinata nel metodo originale; altrimenti viene utilizzato il metodo originale.

Quando un metodo definito dall'utente viene creato recuperando una funzione definita dall'utente da una classe, il suo attributo `im_self` è `None` e l'oggetto metodo si dice essere non associato. Quando viene creato tramite una funzione definita dall'utente da una classe attraverso una sua istanza, il suo attributo `im_self` è l'istanza e l'oggetto metodo si dice essere associato. In entrambi i casi, l'attributo `im_class` del nuovo metodo è la classe da cui il recupero ha luogo e il suo attributo `im_func` è l'oggetto funzione originale.

Quando un metodo definito dall'utente viene creato recuperando un'altro metodo da una classe o un'istanza, il comportamento è lo stesso che per un oggetto funzione, ad accezione del fatto che l'attributo `im_func` della nuova istanza non è l'oggetto che rappresenta il metodo originale, ma il suo attributo `im_func`.

Quando un metodo definito dall'utente viene creato recuperando un metodo di classe da una classe o un'istanza, il suo attributo `im_self` è la classe stessa (la stessa dell'attributo `im_class`) e il suo attributo `im_func` è l'oggetto funzione sottostante al metodo di classe.

Quando un metodo non associato definito dall'utente viene chiamato, la funzione sottostante (`im_func`) viene chiamata, con la restrizione che il primo argomento deve essere un'istanza della classe appropriata (`im_class`) o di un suo tipo derivato.

Quando un metodo associato definito dall'utente viene chiamato, la funzione sottostante (`im_func`) viene chiamata, inserendo l'istanza di classe (`im_self`) all'inizio della lista di argomenti. Per esempio, quando `C` è una classe che contiene una definizione per una funzione `f()` e `x` è un'istanza di `C`, chiamare `x.f(1)` è equivalente a chiamare `C.f(x, 1)`.

Quando un metodo definito dall'utente deriva da un metodo di classe, "l'istanza di classe" memorizzata in `im_self` sarà la classe stessa, così chiamare `x.f(1)` o `C.f(1)` è equivalente a chiamare `f(C,1)`, dove `f` è la funzione sottostante.

Notate che la trasformazione dalla funzione al metodo (associato o non associato) avviene ogni volta che l'attributo viene recuperato da una classe o un'istanza. In alcuni casi, un'ottimizzazione fruttuosa è assegnare l'attributo ad una variabile locale e chiamare la variabile locale. Notare anche che questa trasformazione avviene solo per una funzione definita dall'utente; gli altri oggetti eseguibili (e tutti gli oggetti non eseguibili) vengono recuperati senza trasformazione. È importante notare che le funzioni definite dall'utente che sono attributi di un'istanza di classe non vengono convertiti in metodi associati; questo avviene *solamente* quando la funzione è un'attributo della classe.

Funzioni generatore Una funzione od un metodo che usa l'istruzione `yield` (vedere la sezione 6.8, "L'istruzione `yield`") viene chiamata *funzione generatore*. Come una funzione, una volta invocata, restituisce sempre un oggetto iteratore che può essere usato per eseguire il corpo della funzione: attraverso il metodo iteratore `next()` la funzione verrà eseguita finché continuerà a provvedere un valore usando l'istruzione `yield`. Quando la funzione esegue un'istruzione `return` o raggiunge la fine, viene sollevata un'eccezione `StopIteration` e l'iteratore che avrà raggiunto la fine della serie di valori verrà restituito.

Funzioni built-in Una funzione built-in è un wrapper per le funzioni C. Esempi di funzioni built-in sono `len()` e `math.sin()` (`math` è un modulo built-in standard). Il numero ed il tipo degli argomenti

vengono determinati dalla funzione C. Attributi speciali di sola lettura: `__doc__` è la stringa di documentazione della funzione, o `None` se non è disponibile; `__name__` è il nome della funzione; `__self__` viene impostato a `None` (vedere comunque il prossimo paragrafo); `__module__` è il nome del modulo nel quale la funzione è stata definita o `None` se non è disponibile.

Metodi built-in Questo è il travestimento di una funzione built-in, in questo caso contenente un oggetto passato alla funzione C come un implicito argomento extra. Un esempio di un metodo built-in è *alist*`.append()`, assumendo *alist* come un oggetto lista. In questo caso, l'attributo speciale di sola lettura `__self__` viene impostato all'oggetto rappresentato dalla lista, *list*.

Tipi di classi Le classi tipo, o "classi di nuovo stile", sono invocabili. Questi oggetti agiscono normalmente come generatori di nuove istanze di se stesse, ma sono possibili variazioni per le classi tipo che sovrascrivono `__new__()`. Gli argomenti della chiamata vengono passati a `__new__()` e, in casi tipici, a `__init__()` per l'inizializzazione della nuova istanza.

Classi classiche Gli oggetti classe vengono descritti di seguito. Quando un oggetto classe viene chiamato, viene creata e restituita una nuova istanza della classe (descritta anch'essa di seguito). Questo implica una chiamata al metodo `__init__()` della classe, se esiste. Qualsiasi argomento viene passato al metodo `__init__()`. Se non esiste un metodo `__init__()`, la classe deve essere invocata senza argomenti.

Istanze di una classe Le istanze di una classe vengono descritte di seguito. Le istanza di una classe sono invocabili solo quando la classe ha un metodo `__call__()`; `x(argomenti)` è una scorciatoia per `x.__call__(argomenti)`.

Moduli I moduli vengono importati attraverso l'istruzione `import` (vedere la sezione 6.12, "L'istruzione `import`"). Un oggetto modulo ha uno spazio dei nomi implementato come fosse un dizionario (questo è il dizionario referenziato dall'attributo func_globals delle funzioni definite nel modulo). I riferimenti agli attributi vengono tradotti in controlli in questo dizionario, per esempio, `m.x` è equivalente a `m.__dict__[ x]`. Un oggetto modulo non contiene il codice oggetto usato per inizializzare il modulo (dato che è già stato inizializzato una volta).

L'assegnamento di attributi aggiorna il dizionario dello spazio dei nomi del modulo, per esempio, '`m.x = 1`' equivale a '`m.__dict__[x] = 1`'.

Attributi speciali di sola lettura: `__dict__` è il dizionario dello spazio dei nomi del modulo.

Attributi predefiniti (modificabili): `__name__` è il nome del modulo; `__doc__` è la stringa di documentazione del modulo o `None` se non è disponibile; `__file__` è il percorso del file da cui è stato caricato il modulo, se è stato caricato da un file. L'attributo `__file__` non è presente per i moduli C che sono linkati staticamente nell'interprete; per estensioni di moduli caricati dinamicamente da librerie condivise, è il percorso del file della libreria condivisa.

Classi Gli oggetti classe vengono creati dalla definizione class (vedere la sezione 7.6, "Definizioni di classe"). Una classe ha uno spazio dei nomi implementato attraverso un dizionario. I riferimenti agli attributi della classe vengono tradotti in controlli in questo dizionario, per esempio, '`C.x`' viene convertito in '`C.__dict__[x]`'. Quando l'attributo non viene trovato, la sua ricerca continua nelle classi di base. La ricerca è basata prima sulla profondità, da sinistra a destra e nell'ordine delle occorrenze nella lista delle classi di base.

Quando un riferimento ad un attributo di classe (per una classe `C`) deve produrre una funzione definita dall'utente o uno slegamento di un metodo definito dall'utente la cui classe associata è sia `C` che ad una delle sue classi di base, viene trasformato in un metodo slegato definito dall'utente il cui attributo `im_class` è `C`. Nel caso dovesse produrre un metodo di classe, viene trasformato in un metodo legato definito dall'utente i cui attributi `im_class` e `im_self` sono entrambi `C`. Mentre nel caso dovesse produrre un metodo statico, viene trasformato in un oggetto wrapped dal metodo statico. Vedere la sezione 3.3.2 per altri modi in cui gli attributi ricevuti da una classe possono differire da quelli attualmente contenuti nel proprio `__dict__`.

L'assegnamento di attributi di una classe aggiornano il dizionario della classe e mai il dizionario della classe di base.

Un oggetto classe può essere chiamato (vedere sopra) per produrre un'istanza di una classe (vedere di seguito).

Attributi speciali: `__name__` è il nome della classe; `__module__` è il nome del modulo in cui è definita la classe; `__dict__` è il dizionario contenente lo spazio dei nomi della classe; `__bases__` è una tupla

(forse vuota o un singleton) contenente le classi di base nell'ordine delle loro occorrenze nella lista della classe di base; `__doc__` è la stringa di documentazione della classe o `None` se non è definita.

Istanze di classe L'istanza di una classe viene creata invocando un oggetto class (vedere sopra). L'istanza di una classe ha uno spazio dei nomi implementato come un dizionario che è anche il primo posto in cui verranno cercati i riferimenti agli attributi. Quando non vi viene trovato un attributo e l'istanza della classe non ha attributi con questo nome, la ricerca continua con gli attributi della classe. Se viene trovato un attributo della classe, cioè una funzione definita dall'utente o un metodo non legato definito dall'utente la cui classe associata è la classe (la si chiami `C`) dell'instanza per cui il riferimento all'attributo è stato inizializzato o uno delle sue basi, viene trasformato in un metodo legato definito dall'utente il cui metodo `im_class` è `C` e l'attributo `im_self` è l'istanza. Metodi statici o metodi di classe vengono anch'essi trasformati, come se non fossero derivati dalla classe `C`; vedere "Classi" esposto in precedenza. Si veda la sezione 3.3.2 per un altro modo in cui gli attributi di una classe, ritrovati attraverso le loro istanze, possono differire dall'oggetto attualmente memorizzato nel `__dict__` della classe. Se non viene trovato nessun attributo della classe e la classe possiede un metodo `__getattr__()`, questo viene chiamato per soddisfare la ricerca.

L'assegnamento e la rimozione di attributi aggiornano il dizionario dell'istanza e mai il dizionario della classe. Se la classe possiede un metodo `__setattr__()` o `__delattr__()`, viene invocato questo metodo invece di aggiornare direttamente il dizionario dell'istanza.

Istanze di classe possono pretendere di essere numeri, sequenze o mappe se hanno metodi con certi nomi speciali. Vedere la sezione 3.3, "Nomi di metodi speciali".

Attributi speciali: `__dict__` è il dizionario degli attributi; `__class__` è l'istanza della classe.

File Un oggetto file rappresenta un file aperto. Oggetti file vengono creati dalla funzione built-in `open()` ed anche da `os.popen()`, `os.fdopen()` e dal metodo `makefile()` di oggetti socket (e forse da altre funzioni o metodi forniti da moduli di estensione). Gli oggetti `sys.stdin`, `sys.stdout` e `sys.stderr` vengono inizializzati da oggetti file corrispondenti ai flussi input, output ed error dell'interprete standard. Si veda la *Libreria Python di riferimento* per la documentazione completa di oggetti file.

Tipi interni Alcuni tipi usati internamente dall'interprete vengono resi disponibili all'utente. Le loro definizioni possono cambiare in versioni future dell'interprete, ma vengono menzionati per completezza.

Codice oggetto Il codice oggetto rappresenta il codice Python eseguibile, *byte-compilato*, o *bytecode*. La differenza tra codice oggetto ed oggetto funzione è che la funzione contiene un esplicito riferimento alle funzioni globali (il modulo in cui è stata definita), mentre un codice oggetto non contiene del contesto; anche i valori degli argomenti predefiniti vengono memorizzati nella funzione, non nel codice oggetto (dato che i valori rappresentati vengono calcolati al momento dell'esecuzione). A differenza delle funzioni, il codice oggetto è immutabile e non contiene riferimenti (diretti o indiretti) ad oggetti mutabili.

Attributi speciali di sola lettura: `co_name` fornisce il nome della funzione; `co_argcount` indica il numero degli argomenti posizionali (inclusi gli argomenti con valori predefiniti); `co_nlocals` indica il numero delle variabili locali usate per la funzione (inclusi gli argomenti); `co_varnames` è una tupla contenente i nomi delle variabili locali (partendo con i nomi degli argomenti); `co_cellvars` è una tupla contenente i nomi delle variabili locali che vengono referenziate da funzioni annidate; `co_freevars` è una tupla contenente i nomi delle variabili libere; `co_code` è una stringa che rappresenta la sequenza di istruzioni bytecode; `co_consts` è una tupla contenente le costanti usate dal bytecode; `co_names` è una tupla contentente i nomi usati dal bytecode; `co_filename` è il nome del file da cui è stato compilato il codice; `co_firstlineno` indica il numero della prima riga della funzione; `co_lnotab` è una stringa che codifica la mappatura degli offset del bytecode in numeri di riga (per dettagli vedere il codice sorgente dell'interprete); `co_stacksize` è la dimenszione richiesta dello stack (variabili locali incluse); `co_flags` è un intero che codifica un numero di flag per l'interprete.

I seguenti bit flag vengono definiti per `co_flags`: il bit `0x04` viene impostato se la funzione usa la sintassi '`*argomenti`' per accettare un numero arbitrario di argomenti posizionali; il bit `0x08` viene impostato se la funzione usa la sintassi '`**keywords`' per accettare parole chiavi arbitrarie; il bit `0x20` viene impostato se la funzione è un generatore.

Anche le dichiarazioni di caratteristiche future ('`from __future__ import division`') usano bit in `co_flags` per indicare se un codice oggetto è stato compilato con una particolare caratteristica abilitata: il bit `0x2000` viene impostato se la funzione è stata compilata con la funzionalità division di future abilitata; i bit `0x10` e `0x1000` sono stati usati in precedenti versioni di Python.

Altri bit in `co_flags` vengono riservati per un uso interno.

Se un codice oggetto rappresenta un funzione, la prima voce in `co_consts` è la stringa di documentazione della funzione o `None` se non è definita.

Oggetti frame Gli oggetti frame rappresentano i frame di esecuzione. Si possono presentare in oggetti traceback (vedere sotto).

Attributi speciali di sola lettura: `f_back` punta al precedente frame dello stack (riguardo al chiamante), o `None` se questo è l'ultimo frame dello stack; `f_code` è il codice oggetto eseguito in questo frame; `f_locals` è il dizionario usato per cercare le variabili locali; `f_globals` viene usato per variabili globali; `f_builtins` viene usato per nomi (reali) built-in; `f_restricted` è un'opzione che indica se la funzione viene eseguita con talune restrizioni (restricted execution mode); `f_lasti` fornisce l'istruzione precisa (questo è un indice nella striga bytecode del codice oggetto).

Attributi speciali modificabili: `f_trace`, se non è `None`, è una funzione invocabile all'avvio di ogni riga di codice sorgente (usata dal debugger); `f_exc_type`, `f_exc_value`, `f_exc_traceback` rappresentano le eccezioni più recenti sollevate in questo frame; `f_lineno` è il numero di riga corrente del frame — se lo si scrive all'interno di una funzione trace si salta alla data riga (solo per gli ultimi frame). Un debugger può implementare un comando Jump (anche conosciuto come Set Next Statement) scrivendo in `f_lineno`.

Oggetti traceback Gli oggetti traceback rappresentano lo stack trace di un'eccezione. Un oggetto traceback viene creato quando viene sollevata un'eccezione. Quando la ricerca di un'eccezione tratta lo stack di esecuzione, viene inserito un oggetto traceback davanti al corrente traceback per ogni livello. Quando viene inserito un gestore di eccezioni, la traccia dello stack viene resa disponibile al programma. (Vedere la sezione 7.4, "L'istruzione `try`."). È accessibile come `sys.exc_traceback` ed anche come il terzo elemento della tupla restituita da `sys.exc_info()`. Quest'ultima è l'interfaccia preferita dato che funziona correttamente quando il programma sta usando thread multipli. Quando il programma non contiene gestori disponibili, la traccia dello stack viene scritta (formattata con precisione) sullo standard error; se l'interprete è interattivo viene comunque resa disponibile all'utente come `sys.last_traceback`.

Attributi speciali in sola lettura: `tb_next` è il prossimo livello nella traccia dello stack (riguardo al frame dove si è verificata l'eccezione), o `None` se non esiste un ulteriore livello; `tb_frame` punta al frame in esecuzione del livello corrente; `tb_lineno` fornisce il numero di riga dove è stata sollevata l'eccezione; `tb_lasti` indica precisamente l'istruzione. Il numero di riga e l'ultima istruzione del traceback possono differire dal numero di riga del proprio oggetto frame se l'eccezione viene sollevata in un'istruzione `try` con una clausola except non raggiunta o con una clausola `finally`.

Oggetti fetta Gli oggetti fetta vengono usati per rappresentare delle fette quando viene usata una *sintassi di affettazione estesa*. Questa è una fetta che usa i due punti o fette multiple o ellissi separate da virgole, per esempio, `a[i:j:step]`, `a[i:j, k:l]` o `a[..., i:j]`. Vengono anche create dalla funzione built-in `slice()`.

Attributi speciali in sola lettura: `start` è il legame più basso; `stop` è il legame più alto; `step` è il valore del passo; ciascuno di questi valori viene considerato `None` se omesso. Questi attributi possono essere di qualsiasi tipo.

Gli oggetti fetta supportano un metodo:

`indices`(*self, length*)

Questo metodo accetta come singolo argomento un intero (*length*) e calcola l'informazione circa la fetta estesa che l'oggetto fetta dovrebbe descrivere se applicato ad una sequenza di *length* elementi. Restituisce una tupla di tre interi; rispettivamente questi sono gli indici di *start*, *stop* e lo *step* o la lunghezza del passo della fetta. Indici omessi o più grandi del consentito vengono gestiti in modo consistente con fette regolari. Nuovo nella versione 2.3.

Metodi statici degli oggetti I metodi statici forniscono un modo per annullare la trasformazione di funzioni in metodi come descritto in precedenza. Un metodo statico è un involucro (wrapper) attorno ad altri oggetti, di solito metodi definiti dall'utente. Quando un metodo statico viene richiesto da una classe o dall'istanza di una classe, l'oggetto attualmente restituito è un oggetto involucro (wrapper), che non è soggetto ad altre trasformazioni. I metodi statici non sono essi stessi invocabili, benché gli oggetti siano essi stessi involucri (wrap). I metodi statici vengono creati dal costruttore built-in `staticmethod()`.

Metodi di classe degli oggetti Un metodo di classe, come un metodo statico, è un involucro (wrapper) attorno ad un altro oggetto che altera il modo nel quale questo oggetto viene richiamato dalla clas-

se e dalle instanze della classe. Come richiamare un metodo di classe è stato descritto in precedenza, in "Metodi definiti dall'utente". I metodi di classe vengono creati dal costruttore built-in `classmethod()`.

3.3 Nomi di metodi speciali

Una classe può implementare certe operazioni che vengono invocate da sintassi speciali (come operazioni aritmetiche o subscript e affettamenti) definendo metodi con speciali nomenclature. Questo è l'approccio di Python all'*overload degli operatori*, permettendo alle classi di definire il proprio comportamento rispettando altresì gli operatori del linguaggio. Per esempio, se una classe definisce un metodo chiamato `__getitem__()` ed `x` è un'istanza di questa classe, `x[i]` equivale a `x.__getitem__(i)`. Eccetto dove menzionato, i tentativi di eseguire un'operazione quando non sono stati definiti metodi appropriati, sollevano un'eccezione.

Quando si implementa una classe che emula qualsiasi tipo built-in, è importante che l'emulazione sia implementata solo al livello che abbia un senso per l'oggetto che si è preso a modello. Per esempio, alcune sequenze possono funzionare bene per il recupero di elementi individuali, ma l'estrazione di una fetta può non avere senso. (Un esempio di questo è l'interfaccia `NodeList` nel Document Object Model del W3C.)

3.3.1 Personalizzazioni di base

`__init__`(*self*[, ...])
: Chiamata quando viene creata un'istanza. Gli argomenti vengono passati al costruttore della classe. Se una classe base ha un metodo `__init__()`, il metodo `__init__()` della classe derivata, se esiste, deve esplicitamente richiamarlo per assicurare l'appropriata inizializzazione relativa alla classe base dell'istanza; per esempio: '`BaseClass.__init__(`*self*`, [`*args*`...])`'. Come una speciale costrizione sui costruttori, non può essere restituito alcun valore; facendolo verrà sollevata un'eccezione `TypeError` al runtime.

`__del__`(*self*)
: Chiamata quando l'istanza può essere distrutta. Viene anche chiamata distruttore. Se una classe base ha un metodo `__del__()`, il metodo `__del__()` della classe derivata, se esiste, deve esplicitamente richiamarlo per assicurare l'appropriata distruzione relativa alla classe di base dell'istanza. Si noti che è possibile (benché non raccomandato) posporre la distruzione dell'istanza da parte del metodo `__del__()` creando un nuovo riferimento all'istanza. Questa può quindi essere chiamata in un secondo tempo quando questo nuovo riferimento viene cancellato. Non viene garantito che i metodi `__del__()` vengano chiamati per gli oggetti che esistono quando si esce dall'interprete.

  Note: '`del x`' non chiama direttamente `x.__del__()` — il primo decrementa il contatore di riferimenti per `x` di uno, mentre l'altro viene chiamato solo quando il contatore di riferimenti per `x` raggiunge zero. Alcune situazioni comuni che possono prevenire il raggiungimento di zero da parte del contatore di riferimenti di un oggetto include: riferimenti circolari tra oggetti (per esempio, una lista doppiamente linkata o una struttura dati ad albero con parenti e figli puntatori); un riferimento all'oggetto nel frame dello stack di una funzione che può sollevare un'eccezione (il traceback memorizzato in `sys.exc_traceback` mantiene il frame dello stack attivo); o un riferimento all'oggetto che nel frame dello stack raggiunge un eccezione non gestita in modo interattivo (il traceback memorizzato in `sys.last_traceback` mantiene il frame dello stack attivo). La prima situazione può essere rimediata solo da un'esplicita rottura dei cicli; le ultime due situazioni possono essere risolte memorizzando `None` in `sys.exc_traceback` o `sys.last_traceback`. Riferimenti circolari inutili vengono scoperti quando viene abilitata l'opzione cycle detector (è attiva in modo predefinito), ma possono essere ripuliti solo se non ci sono metodi `__del__()` coinvolti a livello Python. Far riferimento alla documentazione del modulo `gc` per maggiori informazioni su come i metodi `__del__()` vengono gestiti dai cycle detector, in particolare la descrizione del valore della `garbage`.

  A causa delle precarie circostanze sotto le quali i metodi `__del__()` vengono invocati, le eccezioni sollevate durante la loro esecuzione vengono ignorate, al loro posto viene stampato un avviso su `sys.stderr`. Inoltre, quando `__del__()` viene invocato come risposta ad un modulo che deve essere rimosso (per esempio, quando l'esecuzione di un programma raggiunge il termine), altri riferimenti globali referenziati dal metodo `__del__()` possono dover essere cancellati. Per questa ragione, i metodi `__del__()`

dovrebbero fare il minimo indispensabile in assoluto per mantenere l'esterno invariato. Dalla versione 1.5, Python garantisce che questi riferimenti globali i cui nomi iniziano con un singolo carattere di sottolineatura siano rimossi dai propri moduli prima che altri riferimenti globali vengano rimossi; il fatto che non esistano altri riferimenti globali assicura che i moduli importati siano ancora disponibili al momento della chiamata al metodo `__del__()`.

`__repr__`(*self*)
Chiamata dalla funzione built-in `repr()` e da conversioni in stringa (virgolette rovesciate) per calcolare la rappresentazione "ufficiale" di un oggetto stringa. Se tutto è possibile, questa dovrebbe essere vista come una valida espressione Python che potrebbe essere usata per ricreare un oggetto con lo stesso valore (stabilendo un ambiente appropriato). Se questo non è possibile, dovrebbe essere restituita una stringa nella forma '<*...qualche utile descrizione...*>'. Il valore restituito deve essere un oggetto stringa. Se una classe definisce `__repr__()` ma non `__str__()`, il metodo `__repr__()` verrà usato anche quando verrà richiesta una rappresentazione di stringa "informale" di istanze richieste da questa classe.

Questo è il tipico uso per il debugging, pertanto è importante che la rappresentazione sia ricca di informazioni e non ambigua.

`__str__`(*self*)
Chiamato dalla funzione built-in `str()` e dall'istruzione `print` per calcolare la rappresentazione di stringa "informale" di un oggetto, che differisce da `__repr__()` in quanto non deve essere un'espressione valida Python: può, invece, venire usata una rappresentazione più adatta e concisa. Il valore restituito deve essere un oggetto stringa.

`__lt__`(*self, other*)
`__le__`(*self, other*)
`__eq__`(*self, other*)
`__ne__`(*self, other*)
`__gt__`(*self, other*)
`__ge__`(*self, other*)
Nuovo nella versione 2.1. Questi sono i cosiddetti metodi di "confronto ricco" e vengono usati per confrontare degli operatori con priorità a `__cmp__()` preso in esame di seguito. La corrispondenza tra operatori e metodi è la seguente: *x*<*y* chiama *x*`.__lt__(`*y*`)`, *x*<=*y* chiama *x*`.__le__(`*y*`)`, *x*==*y* chiama *x*`.__eq__(`*y*`)`, *x*`!=`*y* e *x*<>*y* chiama *x*`.__ne__(`*y*`)`, *x*>*y* chiama *x*`.__gt__(`*y*`)` e *x*>=*y* chiama *x*`.__ge__(`*y*`)`. Questi metodi possono restituire qualsiasi valore, ma se l'operatore di confronto viene usato in un contesto booleano, il valore restituito dovrebbe essere interpretabile come un valore booleano, altrimenti viene sollevata un'eccezione `TypeError`. Per convenzione, `False` viene usato per falso e `True` per vero.

Non ci sono relazioni implicite tra gli operatori di confronto. Il fatto che *x*==*y* sia vero non implica che *x*`!=`*y* sia falso. Per questo, definendo `__eq__`, si dovrebbe anche definire `__ne__` in modo che gli operatori si comportino come ci si aspetta.

Non ci sono versioni riflesse (con argomenti scambiati) di questi metodi (quando cioè l'argomento di sinistra non supporta l'operazione mentre lo supporta l'argomento di destra); piuttosto, `__lt__()` e `__gt__()` sono l'una il contrario dell'altra, la stessa cosa per `__le__()` e `__ge__()` ed anche `__eq__()` e `__ne__()` riflettono la medesima situazione.

Gli argomenti dei metodi di confronto ricco non vengono mai forzati. Un metodo di confronto può restituire `NotImplemented` se l'operazione tra la coppia di argomenti non viene implementata.

`__cmp__`(*self, other*)
Chiamato da operazioni di confronto se un metodo di confronto (si veda sopra) non è stato definito. Dovrebbe restituire un intero negativo se `self < other`, zero se `self == other`, un intero positivo se `self > other`. Se non viene definita alcuna operazione `__cmp__()`, `__eq__()` o `__ne__()`, istanze di classe vengono confrontate tramite oggetti identità ("indirizzi"). Si veda anche la descrizione di `__hash__()` per alcune note sulla creazione di oggetti che supportino operazioni di confronto personalizzate e che siano utilizzabili come chiavi di dizionari. (Nota: la restrizione che le eccezioni non vengano propagate da `__cmp__()` è stata rimossa a partire dalla versione 1.5 di Python.)

`__rcmp__`(*self, other*)
Modificato nella versione 2.1: Non più supportato.

`__hash__`(*self*)

Chiamato per oggetti chiave in operazioni su dizionari e dalla funzione built-in `hash()`. Dovrebbe restituire un intero a 32-bit utilizzabile come un valore di hash per operazioni su dizionari. L'unica proprietà richiesta è che oggetti uguali abbiano lo stesso valore di hash; viene consigliato di mischiare insieme, in qualche modo (per esempio usando un'or esclusivo), i valori di hash per i componenti dell'oggetto che gioca anche un ruolo nel confronto di oggetti. Se in una classe non viene definito un metodo `__cmp__()` non dovrebbe venire definita neanche un'operazione `__hash__()`; se viene definito `__cmp__()` o `__eq__()` ma non `__hash__()`, la sua istanza non potrà essere usata come chiave di un dizionario. Se in una classe vengono definiti oggetti mutabili e viene implementato un metodo `__cmp__()` o `__eq__()`, non si dovrebbe implementare `__hash__()` dato che l'implementazione di un dizionario richiede che un valore hash di una chiave sia immutabile (se il valore hash dell'oggetto cambia, si otterrà un hash errato).

`__nonzero__`(*self*)
Chiamato per implementare un test su un valore vero o dall'operazione built-in `bool()`; dovrebbe restituire `False` o `True`, o il loro equivalente numerico (`0` o `1`). Quando questo metodo non viene definito, viene richiamato `__len__()` se definito (si veda di seguito). Se in una classe non vengono definiti né `__len__()` né `__nonzero__()`, tutte le loro istanze vengono considerate vere.

`__unicode__`(*self*)
Chiamato per implementare la funzione built-in functionunicode(); dovrebbe restituire un oggetto Unicode. Quando questo metodo non viene definito, viene tentata la conversione in stringa ed il risultato della conversione in stringa viene convertito in Unicode usando il sistema di codifica predefinito.

3.3.2 Personalizzare gli attributi di accesso

Si possono definire i seguenti metodi per personalizzare il significato di accesso agli attributi (l'uso, l'assegnamento o la cancellazione di `x.name`) delle istanze di classe.

`__getattr__`(*self, name*)
Viene chiamato quando la ricerca nelle posizioni usuali di un attributo ha dato esito negativo (per esempio non è un attributo d'istanza e non c'è nemmeno nell'albero di classe per `self`). Il nome dell'attributo è `name`. Questo metodo restituisce il valore (calcolato) dell'attributo oppure solleva l'eccezione `AttributeError`.

Notare che se l'attributo viene individuato mediante il solito meccanismo, la chiamata a `__getattr__()` non avviene (si tratta di una asimmetria voluta tra `__getattr__()` e `__setattr__()`). Questo, allo stesso tempo, succede per ragioni di efficienza e perché altrimenti `__setattr__()` non avrebbe nessun modo per accedere agli altri attributi dell'istanza. Poi, almeno per le variabili di istanza, se ne può alterare interamente il controllo, non inserendo alcun valore del dizionario degli attributi d'istanza (collocarli invece in un altro oggetto). Vedere più avanti il metodo `__getattribute__()` per come attualmente si ha il controllo assoluto nelle classi di nuovo stile.

`__setattr__`(*self, name, value*)
Chiamato quando viene tentato l'assegnamento di un attributo. È chiamato al posto del normale meccanismo (per esempio memorizzare il valore nel dizionario delle istanze). Il nome dell'attributo è *name*, *value* è il valore da assegnargli.

Se `__setattr__()` vuole assegnare un attributo d'istanza, non basta la semplice esecuzione di '`self.`*name* `= value`' — ciò causerà una chiamata ricorsiva a se stesso. Invece, deve inserire il valore nel dizionario degli attributi d'istanza: ad esempio '`self.__dict__[`*name*`] = value`'. Le classi di nuovo stile, piuttosto che accedere al dizionario delle istanze, devono chiamare il metodo della classe base avente lo stesso nome: per esempio, '`object.__setattr__(self, name, value)`'.

`__delattr__`(*self, name*)
Funziona come `__setattr__()`, ma per la cancellazione dell'attributo invece dell'assegnamento. Dovrebbe essere implementato solo se per l'oggetto è significativo '`del obj.`*name*'.

Altri modi per accedere agli attributi delle classi di nuovo stile

I seguenti metodi si applicano solamente alle classi di nuovo stile.

`__getattribute__`(*self, name*)

Viene chiamato per implementare gli accessi agli attributi riguardanti le istanze di una classe. Non verrà mai chiamato (a meno che sia fatto esplicitamente) se la classe ha già un metodo `__getattr__`. Questo metodo dovrebbe restituire il valore (calcolato) dell'attributo oppure sollevare l'eccezione `AttributeError`. Affinché in tale metodo sia evitata una ricorsione infinita, la sua implementazione, per l'accesso agli attributi che necessita, deve sempre richiamare il metodo base della classe con lo stesso nome; per esempio '`object.__getattribute__(self, name)`'.

Implementazione dei descrittori

I metodi che seguono si applicano solo quando un'istanza di classe contenente un metodo (chiamato anche classe *descrittore*) compare in una classe dizionario di un'altra classe di nuovo stile, conosciuta come classe di *appartenenza*. Nell'esempio sotto, "l'attributo" si riferisce a quello il cui nome costituisce la proprietà chiave nella classe di appartenenza `__dict__`. I descrittori stessi possono essere solo implementati come classi di nuovo stile.

`__get__`(*self, instance, owner*)
Chiamato per ottenere l'attributo della classe di appartenenza (accesso all'attributo di classe) o di un'istanza di quella classe (accesso all'attributo d'istanza). *owner* è sempre la classe di appartenenza, mentre *instance* è l'istanza, il cui attributo ha avuto accesso, oppure `None` quando si ottiene l'attributo mediante la classe di appartenenza. Questo metodo deve restituire il valore (calcolato) dell'attributo o sollevare l'eccezione `AttributeError`.

`__set__`(*self, instance, value*)
Chiamato per impostare ad un nuovo valore *value*, l'attributo in un'istanza *instance* della classe di appartenenza.

`__delete__`(*self, instance*)
Viene chiamato per cancellare l'attributo in un'istanza *instance* della classe di appartenenza.

Invocare i descrittori

Generalmente, un descrittore è un oggetto attributo che ha un "comportamento limitato", cioè accessi ad attributi sovrascritti dai metodi del suo protocollo: `__get__()`, `__set__()` e `__delete__()`. Si definisce descrittore l'oggetto che ha qualcuno di questi metodi definito.

Il comportamento predefinito per l'accesso all'attributo è di ottenere "get", impostare "set", o cancellare "delete" l'attributo dal dizionario dell'oggetto. Per esempio `a.x` esegue una ricerca vincolata che parte da `a.__dict__['x']`, procede con `type(a).__dict__['x']` e continua attraversando le classi base del `type(a)`, escludendo le metaclassi.

Tuttavia, se il valore cercato è chiaramente un oggetto con definito uno dei metodi descrittori, Python allora può sovrascrivere il comportamento predefinito ed invocare, invece, il metodo descrittore. La precedenza con cui ciò avviene dipende da quali metodi descrittori sono stati definiti e da come sono chiamati. Notare che i descrittori vengono invocati solo per gli oggetti o le classi di nuovo stile (quelle che derivano da `object()` o `type()`).

L'invocazione più comune di un descrittore è il legame `a.x`. Come gli argomenti vengono assemblati dipende da `a`:

Chiamata diretta
: La chiamata più semplice e più comune avviene quando il programmatore invoca direttamente un metodo descrittore: `x.__get__(a)`.

Legame ad un'istanza
: nel caso di un legame ad un'istanza di un oggetto di nuovo stile, `a.x` viene trasformato in chiamata: `type(a).__dict__['x'].__get__(a, type(a))`.

Legame ad una classe
: nel caso di un legame ad una classe di nuovo stile, `A.x` viene trasformato in chiamata: `A.__dict__['x'].__get__(None, A)`.

Super legame
: Se `a` è un'istanza di super, allora il legame `super(B, obj).m()` ricerca `obj.__class__.__mro__` per la classe base `A` immediatamente prima di `B` e poi invoca il descrittore mediante la chiamata: `A.__dict__['m'].__get__(obj, A)`.

Per i legami d'istanza, la precedenza con cui avviene l'invocazione del descrittore dipende da come sono definiti i suoi metodi. I descrittori di dati definiscono entrambi `__get__()` e `__set__()`. Gli altri descrittori hanno solo il metodo `__get__()`. I descrittori di dati sovrascrivono sempre una ridefinizione in un dizionario d'istanza. D'altro canto, tutti gli altri descrittori possono essere sovrascritti dalle istanze.

I metodi di Python (compresi `staticmethod()` e `classmethod()`) non sono implementati come descrittori di dati. Quindi, le istanze possono ridefinire e sovrascrivere i metodi. Ciò permette alle singole istanze di acquisire comportamenti diversi dalle altre istanze della stessa classe.

La funzione `property()` viene implememtata come descrittore di dati. Perciò le istanze non possono sovrascrivere il comportamento di una proprietà.

__slots__

Per definizione, le istanze di classe sia di vecchio che di nuovo stile, hanno un dizionario per la memorizzazione degli attributi. Questo porta ad uno spreco di memoria per quegli oggetti che hanno pochissime variabili d'istanza. Tale consumo può essere notevole quando il numero di istanze create è enorme.

Il dizionario predefinito può essere sovrascritto definendo *__slots__* in una classe di nuovo stile. La dichiarazione di *__slots__* accetta una sequenza di variabili d'istanza e riserva per ognuna di esse spazio a sufficienza per conservare il valore di ciascuna variabile. Risparmia spazio perché *__dict__* non viene creato per ogni istanza.

`__slots__`
: A questa variabile di classe si può attribuire una stringa, un tipo iterabile o una sequenza di stringhe aventi i nomi delle variabili utilizzati dalle istanze. Se *__slots__* viene definito in una classe di nuovo stile, riserverà spazio per le variabili dichiarate ed eviterà la creazione automatica di *__dict__* e *__weakref__* per ogni istanza. Nuovo nella versione 2.2.

Note d'utilizzo di *__slots__*

- Quando manca la variabile *__dict__*, alle istanze non possono venire attribuite variabili nuove che non siano state elencate nella definizione *__slots__*. Il tentativo di assegnarne una non elencata solleverà l'eccezione `AttributeError`. Se è necessaria l'attribuzione dinamica di una nuova variabile, allora si aggiunge `'__dict__'` alla sequenza di stringhe presenti nella dichiarazione *__slots__*.

  Modificato nella versione 2.3: `'__dict__'` aggiunto alla dichiarazione *__slots__* non abiliterà l'assegnamento di nuove variabili non elencate specificatamente nella sequenza dei nomi delle variabili d'istanza..

- Le classi che definiscono *__slots__* non supportano i riferimenti deboli alle sue istanze, senza che ci sia una variabile *__weakref__* per ogni istanza. Se tali riferimenti sono necessari, allora bisogna aggiungere `'__weakref__'` alla sequenza di stringhe della dichiarazione *__slots__*. Modificato nella versione 2.3: precendentemente, l'aggiunta di `'__weakref__'` alla dichiarazione *__slots__* non abilitava i riferimenti deboli..

- Creando i descrittori (3.3.2) per ciascun nome di variabile, si implementano a livello di classe gli *__slots__*. Come risultato, gli attributi di classe non possono essere usati per impostare i valori predefiniti per le variabili d'istanza definite da *__slots__*; altrimenti, l'attributo di classe sovrascrive l'assegnamento del descrittore.

- Se una classe definisce uno slot che è già definito in una classe base, la variabile d'istanza definita da quest'ultima non è accessibile (eccetto recuperando il suo descrittore direttamente dalla classe base). Ciò non rende definito il senso del programma. Per prevenire questo, in futuro potrebbe essere aggiunta una funzione di verifica.

- La dichiarazione *__slots__* ha un effetto limitato alla classe dove viene definita. Conseguentemente, le sottoclassi avranno *__dict__* eccetto che definiscano anche *__slots__*.

- *__slots__* non funziona con le classi derivate dai tipi built-in a "lunghezza variabile" come `long`, `str` e `tuple`.

- Ogni tipo che non sia una stringa iterabile può essere attribuito a *__slots__*. Si possono usare anche le mappe; comunque, nelle prossime versioni, si potrà atribuire un significato speciale ai valori corrispondenti ad ogni proprietà.

3.3.3 Personalizzare la creazione delle classi

Per definizione, le classi di nuovo stile vengono costruite usando `type()`. Una definizione di classe viene letta in uno spazio dei nomi separato ed il nome della classe viene collegato al risultato di `type(name, bases, dict)`.

Durante la lettura della definizione di classe, se è stata definita *__metaclass__*, allora la funzione di chiamata attribuitagli verrà chiamata al posto di `type()`. Le classi permesse o le funzioni scritte che controllano o alterano il processo di creazione della classe:

- Modificano il dizionario di classe prima che la classe venga creata;
- Restituiscono un'istanza di un'altra classe – essenzialmente eseguono il ruolo di una funzione integrata.

`__metaclass__`
: Questa variabile può essere qualunque funzione chiamabile che accetti gli argomenti `name`, `bases` e `dict`. Sulla creazione della classe, la funzione chiamabile viene usata al posto della funzione built-in `type()`. Nuovo nella versione 2.2.

La metaclasse appropriata viene determinata in base alle seguenti regole di precedenza:

- Se esiste `dict['__metaclass__']` viene usato questo.
- Altrimenti, se c'è almeno una classe base, viene usata la sua metaclasse (cerca per primo l'attributo *__class__* e se non lo trova, usa il suo tipo).
- Altrimenti se esiste una variabile denominata *__metaclass__*, viene usata questa.
- Altrimenti viene impiegata la metaclasse classica, di vecchio stile (types.ClassType).

I potenziali utilizzi delle metaclassi sono illimitati. Sono state esaminate varie idee compresi sistemi di registrazione (logging), controllo di interfaccia, automatizzazione di deleghe, creazione automatica di proprietà, proxy, framework e sincronizzazione/locking di risorse.

3.3.4 Emulazione di oggetti "invocabili"

`__call__`(*self*[, *args...*])
: Viene chiamato quando un'istanza viene "chiamata" come una funzione; se questo metodo viene definito, *x*`(arg1, arg2, ...)` è un'abbreviazione di *x*`.__call__(arg1, arg2, ...)`.

3.3.5 Emulare i tipi contenitore

I metodi seguenti possono essere definiti per implementare gli oggetti contenitore. I contenitori sono solitamente sequenze (come le liste o le tuple) o mappe (come i dizionari), ma possono rappresentare altri contenitori altrettanto bene. Il primo insieme di metodi viene usato sia per emulare una sequenza che per emulare una mappa; la differenza è che per una sequenza, le chiavi ammissibili dovrebbero essere numeri interi *k* in cui `0 <=` *k* `<` *N* dove *N* è la lunghezza della sequenza, od oggetti fetta, che definiscono una gamma di articoli. (Per compatibilità all'indietro, il metodo `__getslice__()` (si veda più avanti) può essere definito anche per maneggiare fette semplici, ma non estese.) Si raccomanda inoltre che quelle mappe forniscano i metodi `keys()`, `values()`, `items()`, `has_key()`, `get()`, `clear()`, `setdefault()`, `iterkeys()`, `itervalues()`, `iteritems()`, `pop()`, `popitem()`, `copy()` e `update()`, con un comportamento simile a quello per gli oggetti dizionari standard di Python. Il modulo `UserDict` fornisce una classe `DictMixin` per aiutare la creazione di quei metodi da un insieme base di `__getitem__()`, `__setitem__()`, `__delitem__()` e `keys()`. Le sequenze mutabili dovrebbero fornire i metodi `append()`, `count()`, `index()`, `extend()`, `insert()`, `pop()`, `remove()`, `reverse()` e `sort()`, come gli oggetti lista standard di Python. Infine, i tipi di sequenze dovrebbero implementare l'addizione (intendendo la concatenazione) e la moltiplicazione (intendendo la ripetizione) definendo i metodi `__add__()`, `__radd__()`, `__iadd__()`, `__mul__()`, `__rmul__()` e `__imul__()` descritti sotto; essi non dovrebbero definire `__coerce__()` o altri operatori numerici. Si raccomanda che sia

le mappe che le sequenze implementino il metodo `__contains__()` per consentire un uso efficiente dell'operatore `in`; per le mappe, `in` dovrebbe essere equivalente ad`has_key()`; per le sequenze, esso dovrebbe cercare attraverso i valori. Si raccomanda inoltre che sia le mappe che le sequenze implementino il metodo `__iter__()` per consentire un'iterazione efficiente attraverso il contenitore; per le mappe, `__iter__()` dovrebbe essere lo stesso di `iterkeys()`; per le sequenze, dovrebbe iterare attraverso i valori.

`__len__`(*self*)
Chiamato per implementare la funzione built-in `len()`. Dovrebbe restituire la lunghezza dell'oggetto, un numero intero >= 0. Inoltre, un oggetto che non definisce un metodo `__nonzero__()` e del cui metodo `__len__()` restituisce zero viene considerato come falso in un contesto booleano.

`__getitem__`(*self*, *key*)
Chiamato ad implementare una valutazione di *self*`[`*key*`]`. Per i tipi sequenza, le chiavi accettate dovrebbero essere numeri interi ed oggetti fetta. Si noti che l'interpretazione speciale degli indici negativi (se la classe desidera emulare un tipo sequenza) spetta al metodo `__getitem__()`. Se la chiave *key* è di un tipo inappropriato, potrebbe essere sollevata l'eccezione `TypeError`; se un valore risulta al di fuori dell'insieme di indici per la sequenza (dopo qualsiasi interpretazione speciale dei valori negativi), potrebbe essere sollevata l'eccezione `IndexError`. **Note:** Per i cicli `for` ci si aspetta che venga sollevata un'eccezione `IndexError` per indici illegali, in modo da consentire una corretta rilevazione della fine della sequenza.

`__setitem__`(*self*, *key*, *value*)
Chiamato ad implementare assegnamenti a *self*`[`*key*`]`. Stessa nota per `__getitem__()`. Questo dovrebbe essere implementato solo per le mappe se l'oggetto supporta cambi ai valori per chiavi, o se nuove chiavi possono essere aggiunte, o per sequenze se gli elementi possono essere sostituiti. Le stesse eccezioni dovrebbero essere sollevate per valori chiave *key* impropri per quanto riguarda il metodo `__getitem__()`.

`__delitem__`(*self*, *key*)
Chiamato ad implementare l'eliminazione di *self*`[`*key*`]`. Stessa nota per `__getitem__()`. Questo dovrebbe essere implementato solo per le mappe se gli oggetti supportano la rimozione delle chiavi, o per le sequenze se gli elementi possono essere rimossi dalla sequenza. Le stesse eccezioni dovrebbero essere sollevate per valori chiave, *key*, impropri per quanto riguarda il metodo `__getitem__()`.

`__iter__`(*self*)
Questo metodo viene chiamato quando un iteratore viene richiesto per un contenitore. Questo metodo dovrebbe restituire un nuovo oggetto iteratore che può iterare su tutti gli oggetti nel contenitore. Per le mappe, dovrebbe iterare sopra le chiavi del contenitore, e dovrebbe anche essere reso disponibile come il metodo `iterkeys()`.

Gli oggetti iteratori devono necessariamente implementare anche questo metodo; vengono richiesti per restituire loro stessi. Per maggiori informazioni sugli oggetti iteratore, si veda "Tipi iteratori" nella *Libreria di riferimento di Python*.

Gli operatori per le verifiche di appartenenza (`in` e `not in`) vengono usualmente implementati come un'iterazione per mezzo di una sequenza. Tuttavia, gli oggetti contenitore possono fornire i seguenti metodi speciali di un'implementazione di maggiore efficienza, che inoltre non richiede che l'oggetto sia una sequenza.

`__contains__`(*self*, *item*)
Chiamato per implementare operatori che verifichino l'appartenenza. Dovrebbe restituire vero se l'elemento *item* è all'interno di *self*, altrimenti falso. Per gli oggetti mappa, questo dovrebbe considerare le chiavi della mappa piuttosto che i valori o le coppie elementi-chiave.

3.3.6 Metodi aggiuntivi per emulare tipi sequenza

I seguenti metodi facoltativi possono venire definiti per favorire l'emulazione di oggetti sequenza. Metodi per sequenze immutabili dovrebbero al massimo definire `__getslice__()`; sequenze mutabili possono definire tutti e tre i metodi.

`__getslice__`(*self*, *i*, *j*)
Deprecato dalla versione 2.0. Supporta oggetti fetta come parametri per il metodo `__getitem__()`.

Chiamato per implementare la valutazione di *self*`[`*i*`:`*j*`]`. L'oggetto restituito dovrebbe essere dello stesso tipo di *self*. Si noti che l'assenza di *i* o *j* nell'espressione di affettamento viene sostituita rispettivamente da

zero o `sys.maxint`. Se vengono usati indici negativi nell'affettamento, viene aggiunta all'indice la lunghezza della sequenza. Se l'istanza non implementa il metodo `__len__()`, viene sollevata un'eccezione `AttributeError`. Non viene data alcuna garanzia che gli indici modificati in questo modo non siano ancora negativi. Indici che sono maggiori della lunghezza della sequenza non vengono modificati. Se non viene trovato `__getslice__()`, viene creato un oggetto fetta e passato invece a `__getitem__()`.

`__setslice__`(*self, i, j, sequence*)
Chiamato per implementare l'assegnamento di *self*`[`*i*`:`*j*`]`. Stesse osservazioni fatte su *i* e *j* per `__getslice__()`.

Questo metodo è deprecato. Se non viene trovato `__setslice__()`, o per affettamenti estesi nella forma *self*`[`*i*`:`*j*`:`*k*`]`, viene creato un oggetto fetta e passato a `__setitem__()` invece di richiamare `__setslice__()`.

`__delslice__`(*self, i, j*)
Chiamato per implementare la rimozione di *self*`[`*i*`:`*j*`]`. Stesse osservazioni su *i* e *j* per `__getslice__()`. Questo metodo è deprecato. Se non viene trovato `__delslice__()` o per affettamenti estesi nella forma *self*`[`*i*`:`*j*`:`*k*`]`, viene creato un oggetto fetta e passato a `__delitem__()` invece di richiamare `__delslice__()`.

Da notare che questi metodi vengono invocati solo quando viene usata un'affettamento singolo con un solo carattere due punti ed il metodo di affettamento è disponibile. Per operazioni di affettamento viene richiamato `__getitem__()`, `__setitem__()` o `__delitem__()` con un oggetto fetta come argomento.

Il seguente esempio dimostra come creare un programma o un modulo compatibile con versioni precedenti di Python (assumendo che i metodi `__getitem__()`, `__setitem__()` e `__delitem__()` supportino oggetti fetta come argomenti):

```
class MyClass:
    ...
    def __getitem__(self, index):
        ...
    def __setitem__(self, index, value):
        ...
    def __delitem__(self, index):
        ...

    if sys.version_info < (2, 0):
        # They won't be defined if version is at least 2.0 final

        def __getslice__(self, i, j):
            return self[max(0, i):max(0, j):]
        def __setslice__(self, i, j, seq):
            self[max(0, i):max(0, j):] = seq
        def __delslice__(self, i, j):
            del self[max(0, i):max(0, j):]
    ...
```

Si notino le chiamate a `max()` che sono necessarie a causa della gestione degli indici negativi prima della chiamata ai metodi `__*slice__()`. Quando vengono usati indici negativi, i metodi `__*item__()` li ricevono come vengono forniti ma i metodi `__*slice__()` ottengono una forma "contraffatta" degli indici. Per ogni indice negativo, la lunghezza della sequenza viene aggiunta all'indice prima di richiamare il metodo (anche se può risultare ancora un indice negativo); questa è la gestione abituale degli indici negativi fatta dalle sequenze built-in ed è il comportamento che ci si aspetta anche dai metodi `__*item__()`. Comunque, dato che dovrebbero già fare tutto questo, gli indici negativi non possono essere passati loro; devono essere costretti alla legatura della sequenza prima di venire passati ai metodi `__*item__()`. La chiamata a `max(0, i)` restituisce un valore appropriato.

3.3.7 Emulazione dei tipi numerici

Per emulare gli oggetti numerici si possono definire i seguenti metodi. Non bisognerebbe definire i metodi corrispondenti ad operazioni che non sono supportate da una data tipologia di numero implementato (per esempio per i numeri non integrali le operazioni bit per bit).

__add__(*self, other*)
__sub__(*self, other*)
__mul__(*self, other*)
__floordiv__(*self, other*)
__mod__(*self, other*)
__divmod__(*self, other*)
__pow__(*self, other*[, *modulo*])
__lshift__(*self, other*)
__rshift__(*self, other*)
__and__(*self, other*)
__xor__(*self, other*)
__or__(*self, other*)

Questi metodi vengono chiamati per il calcolo delle operazioni di aritmetica binaria (`+`, `-`, `*`, `//`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`). Per esempio, si chiama *x*`.__add__(`*y*`)` per calcolare l'espressione *x*+*y*, dove *x* è un'istanza di classe con un metodo `__add__()`. Il metodo `__divmod__()` è equivalente all'uso di `__floordiv__()` e `__mod__()`; da non correlare a `__truediv__()` (descritto qui sotto). Notare che, se bisogna supportare la versione ternaria della funzione built-in `pow()`, `__pow__()` dovrebbe essere definito in modo da accettare un terzo argomento.

__div__(*self, other*)
__truediv__(*self, other*)

L'operazione di divisione (`/`) viene effettuata mediante questi metodi. Il metodo `__truediv__()` viene utilizzato quando `__future__.division` è in uso, altrimenti viene usato `__div__()`. Se viene definito solamente uno dei due metodi, l'oggetto non effettuerà la divisione nel diverso contesto; verrà invece sollevata l'eccezione `TypeError`.

__radd__(*self, other*)
__rsub__(*self, other*)
__rmul__(*self, other*)
__rdiv__(*self, other*)
__rtruediv__(*self, other*)
__rfloordiv__(*self, other*)
__rmod__(*self, other*)
__rdivmod__(*self, other*)
__rpow__(*self, other*)
__rlshift__(*self, other*)
__rrshift__(*self, other*)
__rand__(*self, other*)
__rxor__(*self, other*)
__ror__(*self, other*)

Questi metodi vengono richiamati per compiere le operazioni di aritmetica binaria (`+`, `-`, `*`, `/`, `%`, `divmod()`, `pow()`, `**`, `<<`, `>>`, `&`, `^`, `|`) con gli operandi riflessi (NdT: scambiati). Tali funzioni vengono chiamate solamente se l'operando di sinistra non supporta l'operazione corrispondente. Ad esempio, per valutare l'espressione *x*`-`*y*, dove *y* è un'istanza di classe con un metodo `__rsub__()`, viene chiamata *y*`.__rsub__(`*x*`)`. Notare che la funzione ternaria `pow()` non tenterà la chiamata a `__rpow__()` (le regole coercitive diventano troppo complesse).

__iadd__(*self, other*)
__isub__(*self, other*)
__imul__(*self, other*)
__idiv__(*self, other*)
__itruediv__(*self, other*)
__ifloordiv__(*self, other*)
__imod__(*self, other*)

__ipow__(*self, other*[, *modulo*])
__ilshift__(*self, other*)
__irshift__(*self, other*)
__iand__(*self, other*)
__ixor__(*self, other*)
__ior__(*self, other*)

Questi metodi sono chiamati per compiere le operazioni aritmetiche di aumento (`+=`, `-=`, `*=`, `/=`, `%=`, `**=`, `<<=`, `>>=`, `&=`, `^=`, `|=`). Dovrebbero svolgere l'operazione appropriata (modificando *self*) e restituendo il risultato (che potrebbe, ma non per forza, essere *self*). Se non viene definito un metodo specifico, l'operazione di aumento fa ricorso ai metodi consueti. Per esempio, per il calcolo dell'espressione *x*`+=`*y*, dove *x* è un'istanza di classe con un metodo `__iadd()`, richiama *x*`.__add__(`*y*`)`. Se *x* è un'istanza di classe che non ha un metodo `__iadd()`, per il calcolo di *x*`+`*y*, considera i metodi *x*`.__add__(`*y*`)` e *y*`.__radd__(`*x*`)`.

__neg__(*self*)
__pos__(*self*)
__abs__(*self*)
__invert__(*self*)

Richiamati per il calcolo delle operazioni matematiche unarie (`-`, `+`, `abs()` e `~`).

__complex__(*self*)
__int__(*self*)
__long__(*self*)
__float__(*self*)

Questi metodi vengono usati per valutare le funzioni built-in `complex()`, `int()`, `long()` e `float()`. Dovrebbero restituire un valore del tipo appropriato.

__oct__(*self*)
__hex__(*self*)

Chiamati per il calcolo delle funzioni built-in `oct()` ed `hex()`. Dovrebbero restituire un valore stringa.

__coerce__(*self, other*)

Viene chiamato per il calcolo matematico numerico a "modalità misto". Deve restituire una doppia tupla che contiene *self* ed *other* convertiti in un tipo numerico comune, oppure `None` se tale conversione non è fattibile. Quando il tipo comune diventa quello che era *other*, è sufficiente che venga restituito `None`, siccome l'interprete richiederà anche l'altro l'oggetto per tentare la forzatura a quel tipo (ma talvolta, è utile a questo punto effettuare la conversione nell'altro tipo, se la sua implementazione non può essere modificata). Un valore di ritorno pari a `NotImplemented` è come se fosse restituito `None`.

3.3.8 Regole coercitive

Questa sezione illustrerà le regole di coercizione. Man mano che il linguaggio si è evoluto, è diventato più difficile documentarle con esattezza; non è il massimo spiegare che cosa fa una versione di una particolare implementazione. In Python 3.0, le regole di coercizione non verranno più supportate.

- Se l'operando sinistro dell'operatore % è una stringa o un oggetto Unicode non viene applicata nessuna coercizione, ma vengono invocate le operazioni di formattazione stringa.
- Non è più raccomandabile definire un'operazione con regole coercitive. Le operazioni compiute in modalità mista sui tipi, che non determinano una coercizione, passano all'operazione l'argomento iniziale.
- Le classi di nuovo stile (quelle che derivano dalla classe `object`) come risposta ad un operatore binario, non invocheranno mai il metodo `__coerce__()`; `__coerce__()` viene invocato solamente quando si richiama la funzione built-in `coerce()`.
- Nella maggioranza dei casi, un operatore che restituisce `NotImplemented` viene trattato come se non sia stato implementato per niente.
- Sotto, `__op__()` e `__rop__()` vengono impiegati per mostrare l'uso dei dei metodi generici corrispondenti ad un operatore; `__iop__` viene usato per il corrispondente operatore adatto. Per esempio, per l'ope-

ratore '+', `__add__()` e `__radd__()` vengono utilizzati per le varianti sinistra e destra dell'operatore binario e `__iadd__` per la variante appropriata.

- Per oggetti *x* e *y*, viene testato per prima *x*`.__op__(`*y*`)`. Se non è supportato oppure restituisce `NotImplemented`, continua con *y*`.__rop__(`*x*`)`. Se anche questo non è implementato o restituisce `NotImplemented`, viene sollevata l'eccezione `TypeError`. Ma si vedano le seguenti eccezioni:

- Eccezioni riguardanti l'elemento precedente: se l'operando sinistro è un'istanza di un tipo built-in o una classe di nuovo stile, e l'operando destro è un'istanza di una speciale sottoclasse di quel tipo di classe, viene testato il metodo `__rop__()` dell'operando destro *prima* del metodo `__op__()` dell'operando sinistro. Questo avviene affinché una sottoclasse possa completamente sovrascrivere gli operatori binari. Altrimenti, il metodo `__op__` dell'operando sinistro accetta sempre l'operando destro: quando viene attesa un'istanza di una data classe, viene sempre accettata anche un'istanza di una sua sottoclasse.

- Non appena un operando specifica una coercizione, questa viene chiamata prima che il metodo `__op__()` oppure `__rop__()`, relativo al suo tipo venga chiamato. Se tale coercizione restituisce un oggetto di tipo differente dall'operando le cui regole coercitive siano state invocate, parte del processo viene rieseguito usando il nuovo oggetto.

- Quando viene usato un operatore locale (come '`+=`'), se l'operando sinistro supporta `__iop__()`, questo metodo viene invocato senza alcuna coercizione. Quando il processo di calcolo ritorna a `__op__()` e/o `__rop__()`, si applicano le normali regole coercitive.

- Quando in *x*+*y*, *x* è una sequenza che supporta la concatenazione di sequenze, viene appunto invocata la concatenazione di sequenze.

- QUando in *x*`*`*y*, un operatore è una sequenza che supporta la ripetizione di sequenze, e l'altro è costituito da un intero (`int` o `long`), viene invocata la ripetizione di sequenze.

- I confronti ricchi (implementati dai metodi `__eq__()` e via dicendo) non usano mai la coercizione. Il confronto a tre (supportato da `__cmp__()`) utilizza le regole coercitive alle stesse condizioni di come sono impiegate nelle operazioni binarie.

- Nell'attuale implementazione, i tipi numerici built-in `int`, `long` e `float` non usano la coercizione; i tipi numerici complessi, `complex`, comunque la utilizzano. La differenza appare evidente quando si sottoclassano questi tipi. Eccezionalmente, il tipo numerico complesso, `complex`, può essere determinato in modo da evitare le regole coercitive. Tutti questi tipi implementano un metodo `__coerce__()`, per l'uso della funzione built-in `coerce()`.

CAPITOLO

QUATTRO

Modello di Esecuzione

4.1 Nomi e Associazioni

I *Nomi* si riferiscono agli oggetti e vengono introdotti da operazioni di associazione. Ogni occorrenza di un nome nel testo di un programma fa riferimento all'*associazione* di tale nome, stabilita all'interno del blocco funzionale che lo usa.

Un *blocco* è una parte di un programma Python che viene eseguito come una singola unità. Esempi di blocchi sono: un modulo, il corpo di una funzione, una definizione di classe. Ogni comando scritto in modalità interattiva è un blocco. Un file di script (un file passato come standard input all'interprete o specificato come primo parametro sulla riga di comando dell'interprete) è un blocco di codice. Un comando di script (un comando specificato sulla riga di comando dell'interprete con l'opzione '**-c**') è un blocco di codice. Un file letto dalla funzione predefinita `execfile()` è un blocco di codice. L'argomento di tipo stringa passato alla funzione predefinita `eval()` e all'istruzione `exec` è un blocco di codice. L'espressione letta e valutata dalla funzione predefinita `input()` è un blocco di codice.

Un blocco di codice viene eseguito all'interno di una *cornice di esecuzione*. Una cornice contiene delle informazioni amministrative (utilizzate per il debug) e determina dove ed in che modo l'esecuzione può continuare una volta terminata l'esecuzione del blocco di codice corrente.

Lo *scope* (NdT: campo, ambito di visibilità) definisce la visibilità di un nome all'interno di un blocco. Se una variabile locale viene definita in un blocco, il suo ambito di visibilità è quel blocco. Se una definizione viene fatta all'interno di un blocco funzionale, il campo si estende a tutti i blocchi contenuti all'interno di quello in cui è stata realizzata la definizione, a meno che uno dei blocchi contenuti non assegni un oggetto differente allo stesso nome. Il campo di un nome definito all'interno di un blocco di tipo classe è limitato a tale blocco; non si estende ai blocchi di codice dei suoi metodi.

Quando un nome viene utilizzato in un blocco di codice, questo viene risolto all'interno del campo più vicino che lo racchiude. L'insieme di tutti questi ambiti di visibilità di un blocco di codice vengono definiti come l'*ambiente* del blocco di codice.

Se un nome è legato ad un blocco, è una variabile locale di quel blocco. Se un nome è legato a livello di modulo, è una variabile globale. (Le variabili del blocco di codice di tipo modulo sono locali e globali.) Se una variabile viene usata in un blocco di codice ma non viene definita in tale blocco, viene chiamata *variabile libera*.

Quando un nome non viene trovato affatto, viene sollevata un'eccezione `NameError`. Se il nome si riferisce ad una variabile locale che non è stata associata, viene sollevata un'eccezione `UnboundLocalError`. `UnboundLocalError` è una sottoclasse di `NameError`.

I seguenti costrutti collegano i nomi: parametri formali a funzioni, istruzioni `import`, classi e definizioni di funzioni (queste collegano il nome della classe o della funzione nel blocco definito), obiettivi che rappresentano identificatori se trovati in un assegnamento, cicli `for` di intestazioni o nella seconda posizione in un'intestazione di una clausola `except`. L'istruzione `import` nella forma "'`from ...import *`"', collega tutti i nomi definiti nel modulo importato, ad eccezione di quelli che iniziano con un carattere di sottolineatura. Questa forma può essere usata solo a livello di modulo.

Un obiettivo trovato in una dichiarazione `del` viene considerato collegato a quella funzione (sebbene le semantiche attuali svincolino il nome). È illegale svincolare un nome che si riferisce ad un campo collegato; il compilatore riporterà un `SyntaxError`.

Ogni assegnazione o istruzione import avverrà all'interno di un blocco definito da una definizione di classe o funzione o a livello del modulo (il blocco di codice al livello più alto).

Se un nome che collega una operazione si presenta all'interno di un blocco di codice, tutti gli usi del nome all'interno del blocco verranno trattati come riferimenti al blocco corrente. Questo può condurre ad errori quando un nome viene usato all'interno di un blocco prima che questo venga collegato. Questa regola è subdola. Python manca di dichiarazioni e consente operazioni di collegamento di nomi ovunque all'interno di un blocco di codice. Le variabili locali di un blocco di codice possono essere determinate dalla scansione dell'intero testo del blocco per le operazioni di collegamento dei nomi.

Se l'istruzione globale avviene all'interno di un blocco, tutti gli usi del nome specificato nell'istruzione si riferiscono al collegamento di quel nome nello spazio dei nomi di alto livello. I nomi vengono risolti nello spazio dei nomi di alto livello, ricercandoli nello spazio dei nomi globale, per esempio lo spazio dei nomi del modulo che contiene il blocco di codice e lo spazio dei nomi built-in, lo spazio dei nomi del modulo `__builtin__`. Viene ricercato prima nello spazio dei nomi globale. Se il nome non viene trovato lì, viene cercato nello spazio dei nomi built-in. La dichiarazione globale deve precedere tutti gli usi del nome.

Lo spazio dei nomi built-in, associato all'esecuzione di un blocco di codice si trova attualmente cercando il nome `__builtins__` nel suo spazio dei nomi globale; questo dovrebbe essere un dizionario o un modulo (nel caso di lettere viene usato il dizionario dei moduli). Normalmente, lo spazio dei nomi `__builtins__` è il dizionario del modulo built-in `__builtin__` (notare: senza 's'). Se non c'è, viene attivata una modalità di esecuzione limitata.

Lo spazio dei nomi per il modulo viene creato automaticamente la prima volta che il modulo viene importato. Il modulo principale per uno script viene chiamato sempre `__main__`.

La dichiarazione globale ha la stessa capacità dell'operazione di collegamento dei nomi nello stesso blocco. Se il più vicino ambito di visibilità per una variabile libera contiene una dichiarazione globale, la variabile libera viene trattata come una globale.

Una definizione di classe è una dichiarazione eseguibile che può essere usata e definire nomi. Questi riferimenti seguono le normali regole per le definizione dei nomi. Lo spazio dei nomi della definizione di classe diventa l'attributo dizionario della classe. I nomi definiti per l'uso della classe non sono visibili nei metodi.

4.1.1 Interazione con caratteristiche dinamiche

Ci sono diversi casi dove le dichiarazioni di Python sono illegali quando usate in congiunzione con campi annidati che contengono variabili libere.

Se una variabile è riferita ad un campo incluso, è illegale cancellare il nome. Un errore verrà riportato al momento della compilazione.

Se la forma con il carattere jolly di import — '`import *`' — viene usata in una funzione, e la funzione contiene o è un blocco annidato con variabili libere, il compilatore solleverà un'eccezione SyntaxError.

Se viene usato `exec` in una funzione, e la funzione contiene o è un blocco annidato con variabili libere, il compilatore solleverà un'eccezione SyntaxError, a meno che l'`exec` non specifichi esplicitamente lo spazio dei nomi locale per l'`exec`. (In altre parole, '`exec obj`' dovrebbe essere illegale, ma '`exec obj in ns`' dovrebbe essere legale.)

Le funzioni `eval()`, `execfile()`, `input()` e l'istruzione `exec` non hanno accesso all'ambiente completo per la risoluzione dei nomi. I nomi potrebbero essere risolti nello spazio dei nomi locale o globale del chiamante. Le variabili libere non vengono risolte nel più vicino spazio dei nomi, ma nello spazio dei nomi globale.[1] L'istruzione `exec` e le funzioni `eval()` ed `execfile()` hanno argomenti facoltativi per sovrascrivere lo spazio dei nomi locale e globale. Se un solo spazio dei nomi viene specificato, questo viene utilizzato da entrambi.

4.2 Eccezioni

Le eccezioni sono un modo per interrompere il normale flusso di controllo del blocco di codice per gestire errori o altre condizioni eccezionali. Un'eccezione viene *sollevata* nel punto in cui viene rilevato l'errore; può essere

[1] Questa limitazione si verifica poiché il codice che viene eseguito da queste operazioni non è disponibile quando il modulo viene compilato.

gestita da un blocco di codice circostante o da ogni altro blocco di codice dove avviene l'errore.

L'interprete Python solleva un'eccezione quando rileva un errore a run-time (come una divisione per zero). Un programma Pyrhon può anche sollevare esplicitamente un'eccezione con l'istruzione `raise`. I gestori di eccezioni vengono specificati con l'istruzione `try` ... `except`. L'istruzione `try` ... `finally` specifica in modo pulito il codice che non gestisce l'eccezione, ma viene eseguito se viee sollevata o meno un'eccezione nel codice precedente.

Python usa il modello "termination" della gestione di errori: un gestore di errori può estrarre quello che è successo e continuare l'esecuzione ad un altro livello, ma non può riparare la causa dell'errore e riprovare l'operazione fallita, eccetto reimmettere il codice dannoso dall'inizio.

Quando un'eccezione non viene gestita del tutto, l'interprete termina l'esecuzione del programma, o lo riporta all'esecuzione interattiva del ciclo principale. In ogni caso, stampa una lista, eccetto quando l'eccezione è una `SystemExit`.

Le eccezioni vengono identificate per istanze di classe. La selezione di una corrispondenza di una clausola except è basata sull'identità dell'oggetto. La clausola `except` deve riferirsi alla stessa classe o una classe di base della stessa.

Quando viene sollevata un'eccezione, un oggetto (potrebbe essere anche `None`) viene passato come *valore* dell'eccezione; questo oggetto non influisce sulla selezione di un gestore di errori, ma viene passato al gestore di errori selezionato come informazione aggiuntiva. Per le classi di eccezioni, questo oggetto deve essere un'istanza della classe di eccezioni sollevata inizialmente.

> **Avvertenze:** i messaggi degli errori non sono parte della API Python. Il loro contenuto può cambiare da una versione di Python ad una nuova senza che vi siano avvertimenti e non dovrebbe essere rilasciata su quel codice che verrà eseguito sotto versioni multiple dell'interprete.

Si veda anche la descrizione dell'istruzione `try` nella sezione 7.4 e l'istruzione `raise` nella sezione 6.9.

CAPITOLO

CINQUE

Espressioni

Questo capitolo spiega il significato degli elementi delle espressioni in Python.

Note sulla sintassi: in questo e nei prossimi capitoli, la notazione BNF estesa verrà usata per descrivere la sintassi, non l'analisi lessicale. Quando (un'alternativa di) una regola sintattica ha la forma

```
name  ::=  othername
```

e non viene fornita alcuna semantica, le semantiche di questa forma di `name` (NdT: nome) sono le stesse di `othername` (NdT: altro nome).

5.1 Conversioni aritmetiche

Quando in una descrizione di un operatore aritmetico si userà la frase "gli argomenti numerici vengono convertiti in un tipo comune" gli argomenti vengono forzati ad usare regole di coercizione, elencate alla fine del capitolo 3. Se entrambi gli argomenti sono tipi numerici standard, vengono applicate le seguenti coercizioni:

- Se uno degli argomenti è un numero complesso, l'altro viene converito a complesso;
- altrimenti, se uno degli argomenti è un numero in virgola mobile, l'altro viene convertito in virgola mobile;
- altrimenti, se uno degli argomenti è un intero long, l'altro viene convertito ad un intero long;
- altrimenti, entrambi devono essere semplici interi e non è necessaria alcuna conversione.

Alcune altre regole si applicano per certi operatori (ad esempio, una stringa argomento sinistro dell'operatore '%'). Le estensioni possono definire le proprie coercizioni.

5.2 Atomi

Gli atomi sono gli elementi basilari delle espressioni. I più semplici atomi sono identificatori o costanti manifeste. Forme racchiuse tra apici inversi o tra parentesi tonde, quadre o graffe vengono classificate sintatticamente come atomi. La sintassi per gli atomi è:

```
atom       ::=  identifier | literal | enclosure
enclosure  ::=  parenth_form | list_display
                | dict_display | string_conversion
```

5.2.1 Identificatori (Names)

Un identificatore che si presenta come un atomo è un nome. Si veda la sezione 4.1 per la documentazione circa le convenzioni su nomi ed associazioni.

Quando il nome è associato ad un oggetto, la valutazione dell'atomo restituisce quell'oggetto. Quando un nome non è associato, un tentativo di valutarlo solleva un'eccezione `NameError`.

Private name mangling:

Quando un identificatore viene rinvenuto testualmente nella definizione di una classe e comincia con due o più caratteri underscore ma non termina con due o più underscore, viene considerato un *nome privato* di quella classe. I nomi privati vengono trasformati nella forma estesa dopo che è stato generato il codice per questi. La trasformazione inserisce il nome della classe davanti al nome, prima rimuovendo gli underscore e poi inserendo un singolo underscore prima del nome. Per esempio, l'identificatore `__spam` viene rinvenuto in una classe chiamata `Ham` che verrà trasformata in `_Ham__spam`. Questa trasformazione è indipendente dal contesto sintattico in cui viene usato l'identificatore. Se la trasformazione del nome è estremamente lunga (più lunga di 255 caratteri), l'implementazione tronca il nome dove capita. Se il nome della classe si compone solamente di underscore, non avviene alcuna trasformazione.

5.2.2 Costanti manifeste

Python supporta stringhe costanti manifeste e varie costanti numeriche:

```
literal  ::=  stringliteral | integer | longinteger
              | floatnumber | imagnumber
```

La valutazione di tali costanti produce un oggetto del tipo dato (stringa, intero, intero long, in virgola mobile, numero complesso) assieme al suo valore. Quest'ultimo può venire approssimato nel caso di costanti formate da un numero in virgola mobile o dalla parte immaginaria di un numero complesso. Vedere la sezione 2.4 per i dettagli.

Tutte le costanti corrispondono a tipi di dati immutabili, per cui l'identità dell'oggetto non è importante quanto il suo valore. Dalla valutazione di più costanti che hanno un valore identico (la stessa occorrenza nel programma oppure diverse occorrenze) si ottiene l'oggetto stesso oppure un diverso oggetto avente lo stesso valore.

5.2.3 Forme tra parentesi

La forma tra parentesi è una lista di espressioni facoltative racchiusa tra parentesi:

```
parenth_form  ::=  ( [expression_list] )
```

Una lista di espressioni tra parentesi produce ciò che tale lista contiene: se ha almeno una virgola, si ottiene una tupla; altrimenti produce una singola espressione costituita dalla lista.

Due parentesi vuote creano un oggetto formato da una tupla vuota. Siccome le tuple sono immutabili, si applicano le regole per le costanti (per esempio due occorrenze di una tupla vuota possono produrre o meno lo stesso oggetto).

Notare che le tuple non sono costituite da parentesi, ma piuttosto hanno la virgola come operatore. Una tupla vuota è l'eccezione che *richiede* le parentesi — permettere nelle espressioni "qualunque cosa" senza racchiuderlo tra parentesi causa ambiguità e permette ai comuni errori di passare l'interprete senza che siano gestiti.

5.2.4 Visualizzare le liste

Una lista può essere rappresentata da una serie vuota di espressioni racchiuse tra parentesi quadre:

```
test          ::=  and_test ( or and_test )* | lambda_form
testlist      ::=  test ( , test )* [ , ]
list_display  ::=  [ [listmaker] ]
listmaker     ::=  expression ( list_for | ( , expression )* [,] )
list_iter     ::=  list_for | list_if
list_for      ::=  for expression_list in testlist [list_iter]
list_if       ::=  if test [list_iter]
```

Visualizzare una lista produce un nuovo oggetto lista. I suoi contenuti vengono specificati fornendo ad entrambi una lista delle espressioni o una costruzione di lista. Quando viene fornita una lista di espressioni separata da virgole, i suoi elementi vengono valutati da sinistra a destra e posti nell'oggetto lista in quell'ordine. Quando viene fornita una costruzione di lista, questa consiste in una singola espressione seguita da almeno una clausola

`for` e zero o più clausole `for` o `if`. In questo caso, gli elementi della nuova lista sono quelli che si vuole siano prodotti considerando un blocco ognuna delle clausole `for` o `if`, annidando da sinistra a destra e valutando che l'espressione produca un elemento della lista ogni volta che il blocco più interno è stato raggiunto.

5.2.5 Visualizzare i dizionari

Un dizionario può essere rappresentato da una serie vuota di chiave/dato racchiusi tra parentesi graffe:

```
dict_display    ::=  { [key_datum_list] }
key_datum_list  ::=  key_datum (, key_datum)* [,]
key_datum       ::=  expression :  expression
```

Visualizzare un dizionario produce un nuovo oggetto dizionario.

Le coppie chiave/dato vengono valutate da sinistra a destra per definire le varie voci del dizionario: ogni oggetto chiave viene usato come chiave nel dizionario per memorizzare il dato corrispondente.

Le restrizioni nei tipi dei valori delle chiavi sono stati elencati precedentemente nella sezione 3.2. (Per riassumere, il tipo chiave dovrebbe essere hashable, questo esclude tutti gli oggetti mutabili.) Contrasti fra chiavi duplicate non vengono rilevati; prevale l'ultimo dato (testualmente mostrato all'estrema destra) memorizzato per una data chiave.

5.2.6 Conversioni di stringhe

Una conversione di stringa è un'espressione di lista racchiusa tra apici inversi:

```
string_conversion  ::=  ‘ expression_list ‘
```

Una conversione di stringa valuta il contenuto dell'espressione di lista e converte l'oggetto risultante in una stringa secondo le regole specifiche del suo tipo.

Se l'oggetto è una stringa, un numero, `None`, o una tupla, lista o dizionario che contiene soltanto gli oggetti il cui tipo è uno di questi, la stringa risultante è un'espressione di Python valida che può essere passata alla funzione built-in `eval()` per produrre un'espressione con lo stesso valore (o un'approssimazione, se sono implicati i numeri in virgola mobile).

(In particolare, per convertire una stringa si dovranno aggiungere alla stessa i caratteri di quotatura e convertire i caratteri "strani" in sequenze di escape che risultino adatte alla stampa.)

Gli oggetti ricorsivi (per esempio, liste o dizionari che contengono un riferimento a sé stessi, direttamente o indirettamente) usano '`...`' per indicare un riferimento ricorsivo ed il risultato non può essere passato ad `eval()` per ottenere un valore equivalente (verrà invece sollevata un'eccezione `SyntaxError`).

La funzione built-in `repr()` esegue esattamente la stessa conversione, racchiudendo l'espressione ed i suoi argomenti tra parentesi ed apici inversi. La funzione built-in `str()` esegue una simile, ma più amichevole, conversione.

5.3 Primitive

Le primitive rappresentano le operazioni maggiormente legate al linguaggio. La loro sintassi è:

```
primary  ::=  atom | attributeref | subscription | slicing | call
```

5.3.1 Riferimenti ad attributo

Un riferimento ad attributo è una primitiva seguita da un punto ed un nome:

```
attributeref  ::=  primary .  identifier
```

La primitiva deve valutare che un oggetto di un tipo supporti riferimenti ad attributo, per esempio un modulo, una lista o un'istanza. Questo oggetto deve produrre l'attributo il cui nome è l'identificativo. Se questo attributo

non è disponibile, viene sollevata un'eccezione `AttributeError`. Altrimenti, il tipo ed il valore dell'oggetto vengono determinati dall'oggetto. Valutazioni multiple dello stesso riferimento ad attributo possono produrre oggetti differenti.

5.3.2 Subscriptions

Una subscription seleziona un elemento di una sequenza (stringa, tupla o lista) o di un oggetto mappa (dizionario):

```
subscription  ::=  primary [ expression_list ]
```

La primitiva deve valutare se l'oggetto è una sequenza o una mappa.

Se la primitiva è una mappa, la lista di espressioni deve valutare un oggetto il cui valore è una delle chiavi della mappa e la subscription seleziona il valore nella mappa che corrisponde a questa chiave. (La lista di espressioni è una tupla, eccetto se ha esattamente un solo elemento.)

Se la primitiva è una sequenza, la lista di espressioni deve valutare un intero naturale. Se questo valore è negativo, gli viene aggiunta la lunghezza della sequenza (in modo che, per esempio, `x[-1]` selezioni l'ultimo elemento di `x`). Il valore risultante deve essere un intero non negativo minore del numero di elementi della sequenza e la subscription seleziona l'elemento il cui indice è questo valore (contando da zero).

Un elemento stringa è formato da caratteri. Un carattere non è un tipo di dati separati ma una stringa di esattamente un carattere.

5.3.3 Fette

Una fetta seleziona un intervallo di elementi in una sequenza (per esempio, una stringa, una tupla o una lista). L'affettamento può essere usato come un'espressione o come identificativo in istruzioni di assegnamento o cancellazione. La sintassi per un'affettamento è:

```
slicing           ::=  simple_slicing | extended_slicing
simple_slicing    ::=  primary [ short_slice ]
extended_slicing  ::=  primary [ slice_list ]
slice_list        ::=  slice_item (, slice_item)* [,]
slice_item        ::=  expression | proper_slice | ellipsis
proper_slice      ::=  short_slice | long_slice
short_slice       ::=  [lower_bound] :  [upper_bound]
long_slice        ::=  short_slice :  [stride]
lower_bound       ::=  expression
upper_bound       ::=  expression
stride            ::=  expression
ellipsis          ::=  ...
```

C'è ambiguità nella sintassi formale: tutto quello che sembra un lista di espressioni sembra anche una lista di fette, così ogni subscription può essere interpretata come un'affettamento. Invece di complicare la sintassi, l'ambiguità viene evitata definendo che in questo caso l'interpretazione come una subscription ha la priorità sull'affettamento (questo è il caso in cui la lista di fette non contiene fette appropriate né elementi ellittici). In modo simile, quando la lista di fette ha esattamente una fetta concisa e nessuna virgola a seguire, l'interpretazione come semplice affettamento ha la priorità come se fosse un'affettamento esteso.

Segue la semantica per un'affettamento semplice. La primitiva deve valutare una sequenza. Le espressioni di partenza e di arrivo, se presenti, devono esprimere un intero; i valori predefiniti sono rispettivamente zero e `sys.maxint`. Se entrambi sono negativi, gli viene aggiunta la lunghezza della sequenza. L'affettamento seleziona quindi tutti gli elementi con indice *k* cosicché *i* `<=` *k* `<` *j* dove *i* e *j* sono i legami di partenza ed arrivo specificati. Questa può essere una sequenza vuota. Non è un errore se *i* o *j* oltrepassano l'intervallo di indici valido (se gli elementi non esistono, non vengono selezionati).

Segue la semantica per un'affettamento esteso. La primitiva deve valutare una mappa che viene indicizzata con una chiave che è costituita almeno da una virgola, la chiave è una tupla contenente la conversione degli elementi della fetta; altrimenti, la conversione della sola fetta è la chiave. La conversione di un elemento di una fetta ellittica

è l'oggetto built-in `Ellipsis`. La conversione di una fetta opportuna è un oggetto fetta (vedete la sezione 3.2) i cui attributi di partenza, arrivo e passo (NdT: `start`, `stop` e `step`) sono i valori dell'espressione data come lower_bound, upper_bound e stride, rispettivamente, sostituendo `None` ad espressioni omesse.

5.3.4 Chiamate

Una chiamata richiede un oggetto invocabile (per esempio una funzione) con una possibile lista vuota di argomenti:

```
call                 ::=  primary ( [argument_list [,]] )
argument_list        ::=  positional_arguments [, keyword_arguments]
                           [, * expression]
                           [, ** expression]
                          | keyword_arguments [, * expression]
                           [, ** expression]
                          | * expression [, ** expression]
                          | ** expression
positional_arguments ::=  expression (, expression)*
keyword_arguments    ::=  keyword_item (, keyword_item)*
keyword_item         ::=  identifier = expression
```

Dopo una lista di argomenti può essere presente una virgola che non ha comunque effetto sulla semantica.

La primitiva deve valutare un oggetto invocabile (funzioni definite dall'utente, funzioni built-in, metodi di oggetti built-in, classi, metodi di istanze di classe e alcune istanze di classe che sono esse stesse invocabili; attraverso estensioni si possono definire tipi aggiuntivi di oggetti invocabili). Tutti gli argomenti vengono valutati prima di effettuare la chiamata. Fare riferimento alla sezione 7.5 per la sintassi formale delle liste di parametri.

Se sono presenti argomenti chiave, vengono prima convertiti in argomenti posizionali, some segue. Prima viene creata una lista di slot vuoti per i parametri formali. Se ci sono N argomenti posizionali, vengono posti nei primi N slot. Quindi, per ogni argomento chiave, l'identificativo viene usato per determinare lo slot corrispondente (se l'identificativo ha lo stesso nome del primo parametro formale, viene usato il primo slot e così via). Se lo slot è stato già riempito, viene sollevata un'eccezione `TypeError`, altrimenti, il valore dell'argomento viene posto nello slot, riempiendolo (anche se l'argomento è `None`). Quando tutti gli argomenti sono stati processati, gli slot che sono ancora vuoti vengono riempiti con i corrispondenti valori predefiniti nella definizione della funzione. (Valori predefiniti vengono calcolati non appena viene definita la funzione; così un oggetto mutabile come una lista o un dizionario, usati come valori predefiniti, verranno condivisi da tutte le chiamate che non specificano un argomento per lo slot corrispondente; di solito questo dovrebbe essere evitato.) Se ci sono degli slot vuoti per i quali non sono stati specificati dei valori predefiniti, viene sollevata un'eccezione `TypeError`. Altrimenti, la lista di slot pieni viene usata come lista di argomenti per la chiamata.

Se ci sono più argomenti posizionali degli slot di parametri formali, viene sollevata un'eccezione `TypeError`, a meno che non sia presente un parametro formale che usi la sintassi '`*identificativo`'. In questo caso, il parametro formale riceve una tupla contenente gli argomenti posizionali in eccesso (o una tupla vuota se non ci sono stati argomenti posizionali in eccesso).

Se qualche argomento chiave non corrisponde al nome del parametro formale, viene sollevata un'eccezione `TypeError`, a meno che il parametro formale non usi la sintassi '`**identificativo`' nel qual caso, questo parametro formale riceve un dizionario corrispondente contenente gli argomenti chiave in eccesso (usando le parole come chiavi e gli argomenti come i valori corrispondenti), o un (nuovo) dizionario vuoto se non vi sono stati argomenti chiave in eccesso.

Se appare la sintassi '`*espressione`' nella chiamata a funzione, l'`espressione`' deve essere valutata come una sequenza. Gli elementi di questa sequenza vengono trattati come se fossero argomenti posizionali aggiuntivi; se ci sono argomenti posizionali *x1*,...,*xN* ed '`espressione`' viene valutata come una sequenza *y1*,...,*yM*, è equivalente ad una chiamata con M+N argomenti posizionali *x1*,...,*xN*,*y1*,...,*yM*.

Una conseguenza di questo è che anche se la sintassi '`*espressione`' appare *dopo* ogni argomento chiave, viene processato *prima* l'argomento chiave (e quindi l'argomento '`**espressione`', se è presente – vedere sotto). Così:

```
>>> def f(a, b):
...  print a, b
...
>>> f(b=1, *(2,))
2 1
>>> f(a=1, *(2,))
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: f() got multiple values for keyword argument 'a'
>>> f(1, *(2,))
1 2
```

È insolito usare gli argomenti chiave e la sintassi '`*espressione`' nella stessa chiamata e quindi, in pratica, non c'è confusione.

Se è presente la sintassi '`**espressione`' nella chiamata a funzione, '`espressione`' deve essere valutata come un (o una sottoclasse di un) dizionario, il cui contenuto viene trattato come argomenti chiave aggiuntivi. Nel caso in cui una parola chiave appaia sia in '`espressione`' che come una parola chiave di uno specifico argomento, viene sollevata un'eccezione `TypeError`.

Parametri formali che usano la sintassi '`*identificativo`' o '`**identificativo`' non possono essere usati come slot per argomenti posizionali o come nomi di argomenti chiave. Parametri formali che usano la sintassi '`(sublist)`' non possono essere usati come nomi di argomenti chiave; la sublist più lontana corrisponde ad un singolo slot senza nome ed il valore viene assegnato alla sublist usando le tipiche regole di assegnamento per le tuple, dopo che tutti gli altri parametri sono stati processati.

Una chiamata restituisce sempre qualche valore, possibilmente `None`, a meno che non venga sollevata un'eccezione. Il modo in cui viene calcolato questo valore dipende dal tipo di oggetto invocabile.

Se è—

una funzione definita dall'utente: viene eseguito il blocco di codice della funzione, passandole la lista di argomenti. La prima cosa che fa il blocco di codice sarà legare i parametri formali agli argomenti; questo viene descritto nella sezione 7.5. Quando il blocco di codice esegue un'istruzione `return` viene specificato il valore di ritorno della funzione chiamata.

una funzione built-in o un metodo: il risultato dipende dall'interprete; vedere la *Libreria di riferimento di Python* per la descrizione delle funzioni built-in ed i metodi.

un oggetto classe: viene restituita una nuova istanza di questa classe.

un metodo istanza di una classe: viene chiamata la corrispondente funzione definita dell'utente con una lista di argomenti che è più lunga di un elemento della lista originale: l'istanza diviene il primo argomento.

un'istanza di classe: la classe deve definire un metodo `__call__()`; l'effetto è lo stesso, come se venisse chiamato questo metodo.

5.4 L'operatore di potenza

L'operatore di potenza lega più saldamente di un operatore unario sulla propria sinistra e lega meno saldamente di un operatore unario sulla propria destra. La sintassi è:

```
power  ::=  primary [** u_expr]
```

In una sequenza di operatori di potenza e unari senza parentesi, gli operatori vengono valutati da destra a sinistra (questo non forza l'ordine di valutazione per gli operandi).

L'operatore di potenza ha la stessa semantica della funzione built-in `pow()` invocata con due argomenti: causa l'elevazione a potenza dell'argomento di sinistra con l'argomento di destra. Gli argomenti numerici vengono prima convertiti in un tipo comune. Il tipo risultante è quello degli argomenti dopo la coercizione.

Con tipi di operandi misti, si applicano le regole di coercizione per operatori aritmetici binari. Per operandi int e long int, il risultato è dello stesso tipo degli operandi (dopo la coercizione) a meno che il secondo argomento non sia negativo, nel qual caso, tutti gli argomenti vengono convertiti in virgola mobile e viene fornito un risultato in virgola mobile. Per esempio, `10**2` restituisce `100`, ma `10**-2` restituisce `0.01`. (Quest'ultima caratteristica è stata aggiunta in Python 2.2. In Python 2.1 e precedenti, se entrambi gli argomenti erano interi ed il secondo era negativo, veniva sollevata un'eccezione).

L'elevazione di `0.0` ad una potenza negativa produce il risultato `ZeroDivisionError`. L'elevazione di un numero negativo ad una potenza frazionaria solleva un'eccezione `ValueError`.

5.5 Operazioni aritmetiche unarie

Tutte le operazioni aritmetiche unarie (e bit per bit) hanno la stessa priorità:

```
u_expr  ::=  power | - u_expr | + u_expr | ~ u_expr
```

L'operatore unario `-` (meno) produce la negazione del proprio argomento numerico.

L'operatore unario `+` (più) restituisce il proprio argomento numerico invariato.

L'operatore unario `~` (inverso) produce l'inversione bit per bit del proprio argomento numerico intero semplice o intero long. L'inversione bit per bit di `x` viene definita come `-(x+1)` e si applica solo a numeri interi.

In tutti e tre i casi, se l'argomento non è del tipo appropriato viene sollevata un'eccezione `TypeError`.

5.6 Operazioni aritmetiche binarie

Le operazioni aritmetiche binarie hanno il livello di priorità convenzionale. Da notare che alcune di queste operazioni si applicano anche a certi tipi non numerici. A parte l'operatore di potenza, ci sono solo due livelli, uno per gli operatori di moltiplicazione ed uno per gli operatori di addizione:

```
m_expr  ::=  u_expr | m_expr * u_expr | m_expr // u_expr | m_expr / u_expr
             | m_expr % u_expr
a_expr  ::=  m_expr | a_expr + m_expr | a_expr - m_expr
```

L'operatore `*` (moltiplicazione) restituisce il prodotto dei suoi argomenti. Gli argomenti devono essere o entrambi numeri o un argomento deve essere un intero (semplice o long) e l'altro deve essere una sequenza. Nel primo caso, i numeri vengono convertiti in un tipo comune e quindi moltiplicati tra loro. Nell'altro caso, viene eseguita una ripetizione della sequenza; un fattore di ripetizione negativo produce una sequenza vuota.

Gli operatori `/` (divisione) e `//` (divisione con arrotondamento) producono il quoziente dei propri argomenti. Gli argomenti numerici vengono prima convertiti in un tipo comune. Divisioni tra numeri interi semplici o long producono un intero dello stesso tipo; il risultato è quello della divisione matematica con la funzione 'floor' applicata al risultato. Divisioni per zero sollevano l'eccezione `ZeroDivisionError`.

L'operatore `%` (modulo) produce il resto della divisione tra il primo argomento ed il secondo. Gli argomenti numerici vengono prima convertiti in un tipo comune. Se il secondo operando è zero viene sollevata l'eccezione `ZeroDivisionError`. Gli argomenti possono essere numeri in virgola mobile, per esempio `3.14%0.7` è uguale a `0.34` (dato che `3.14` è uguale a `4*0.7 + 0.34`.) L'operatore modulo produce sempre un risultato con lo stesso segno del secondo operando (o zero); il valore assoluto del risultato è rigorosamente più piccolo del valore assoluto del secondo operando[1].

Gli operatori di divisione e modulo per gli interi sono legati dalla seguente caratteristica: `x == (x/y)*y + (x%y)`. La divisione ed il modulo per gli interi sono legati anche dalla funzione built-in `divmod()`: `divmod(x, y) == (x/y, x%y)`. Queste caratteristiche non si applicano ai numeri in virgola mobile; carat-

[1] Mentre `abs(x%y) < abs(y)` è matematicamente vero, per i numeri in virgola mobile può non essere numericamente vero a causa dell'arrotondamento. Per esempio, assumendo una piattaforma in cui i numeri in virgola mobile di Python siano IEEE 754 in precisione doppia, affinché `-1e-100 %1e100` abbia lo stesso segno di `1e100`, il risultato computato `-1e-100 + 1e100` che è numericamente esattamente uguale a `1e100`. La funzione `fmod()` del modulo `math` restuisce un risultato il cui segno corrisponde al segno del primo argomento e perciò, in questo caso, restituisce `-1e-100`. Quale approccio sia il più appropriato, dipende dall'applicazione.

teristiche simili si applicano approssimativamente dove `x/y` viene sostituito da `floor(x/y)` o `floor(x/y) - 1`[2].

Deprecato dalla versione 2.3. L'operatore di divisione con arrotondamento, l'operatore modulo e la funzione `divmod()` non vengono più definite per numeri complessi. È appropriato invece, convertire i numeri complessi in numeri in virgola mobile usando la funzione `abs()`.

L'operatore `+` (addizione) produce la somma dei propri argomenti. Gli argomenti devono essere o entrambi numeri o entrambi sequenze dello stesso tipo. Nel primo caso, i numeri vengono convertiti in un tipo comune e quindi sommati tra loro. Nell'altro caso invece, le sequenze vengono concatenate.

L'operatore `-` (sottrazione) produce la differenza tra i propri argomenti. Gli argomenti numerici vengono prima convertiti in un tipo comune.

5.7 Operazioni di scorrimento

Le operazioni di scorrimento hanno una priorità minore rispetto alle operazioni aritmetiche:

```
shift_expr  ::=  a_expr | shift_expr ( «< |»> ) a_expr
```

Questi operatori accettano come argomento numeri interi semplici o long. Gli argomenti vengono convertiti in un tipo comune. Essi scorrono il primo argomento a sinistra o a destra di un numero di bit dato dal secondo argomento.

Uno scorrimento a destra di *n* bit viene definito come la divisione di `pow(2,`*n*`)`. Uno scorrimento a sinistra di *n* bit viene definito come la moltiplicazione di `pow(2,`*n*`)`; per interi semplici non c'è controllo sull'overflow, così in questo caso l'operazione diminuisce i bit e inverte il segno se il risultato non è minore di `pow(2,31)` in valore assoluto. Scorrimenti di valori negativi sollevano l'eccezione `ValueError`.

5.8 Operazioni binarie bit per bit

Ognuna delle tre operazioni bit per bit ha livelli di priorità differenti:

```
and_expr  ::=  shift_expr | and_expr & shift_expr
xor_expr  ::=  and_expr | xor_expr ^ and_expr
or_expr   ::=  xor_expr | or_expr  xor_expr
```

L'operatore `&` produce l'AND bit per bit dei propri argomenti, che devono essere interi semplici o long. Gli argomenti vengono convertiti in un tipo comune.

L'operatore `^` produce lo XOR (OR esclusivo) bit per bit dei propri argomenti, che devono essere interi semplici o long. Gli argomenti vengono convertiti in un tipo comune.

L'operatore `|` produce l'OR (inclusivo) bit per bit dei propri argomenti, che devono essere interi semplici o long. Gli argomenti vengono convertiti in un tipo comune.

5.9 Confronti

A differenza del C, tutte le operazioni di confronto in Python hanno la stessa priorità, che è più bassa rispetto ad ogni operazione aritmetica, di scorrimento o bit per bit. Inoltre, sempre a differenza del C, espressioni come `a < b < c` vengono interpretate con la convenzione matematica:

```
comparison     ::=  or_expr ( comp_operator or_expr )*
comp_operator  ::=  « |» | == |»= | «= | «> | !=
                    | is [not] | [not] in
```

Confronti booleani producono i valori: `True` o `False`.

[2] Se `x` è molto vicino ad un intero esatto multiplo di `y`, è possibile che `floor(x/y)` sia più grande di `(x-x%y)/y` a causa dell'arrotondamento. In questo caso, Python restituisce quest'ultimo risultato per garantire che `divmod(x,y)[0] * y + x % y` sia il più vicino possibile ad `x`.

I confronti possono essere concatenati arbitrariamente, per esempio, `x < y <= z` è equivalente a `x < y and y <= z`, eccetto per il fatto che `y` viene valutata una sola volta (ma in entrambi i casi `z` non viene valutata se `x < y` è falsa).

Formalmente, se *a*, *b*, *c*, ..., *y*, *z* sono espressioni e *opa*, *opb*, ..., *opy* sono operatori di confronto, allora *a opa b opb c* ...*y opy z* è equivalente a *a opa b* `and` *b opb c* `and` ...*y opy z*, eccetto per il fatto che ogni espressione viene valutata almeno una volta.

Da notare che *a opa b opb c* non implica ogni tipo di comparazione tra *a* e *c*, cioè, per esempio, `x < y > z` è perfettamente legale (anche se non è elegante).

Le forme `<>` e `!=` sono equivalenti; per consistenza col C, `!=` è da preferire; viene anche accettato `!=` menzionato sotto `<>`. La sintassi `<>` sta diventando obsoleta.

Gli operatori `<`, `>`, `==`, `>=`, `<=` e `!=` confrontano i valori di due oggetti. Gli oggetti non devono essere necessariamente dello stesso tipo. Se entrambi sono numeri, vengono convertiti in un tipo comune. Altrimenti, confronti tra oggetti di tipo differente sono *sempre* differenti, e vengono ordinati in modo consistente ma arbitrario.

(Questa definizione inusuale di confronto è stata usata per semplificare la definizione di operazioni come l'ordinamento, l'`in` ed il `not in` negli operatori. In futuro, le regole di confronto per oggetti di tipo differente verranno probabilmente modificate.)

I confronti di oggetti dello stesso tipo dipendono dal tipo:

- I numeri vengono confrontati aritmeticamente.
- Le stringhe vengono confrontate lessicograficamente usando le equivalenze numeriche (il risultato della funzione built-in `ord()`) dei loro caratteri. In questo modo, stringhe unicode e 8-bit sono completamente interoperabili.
- Le tuple e le liste vengono confrontate lessicograficamente usando il confronto degli elementi corrispondenti. Questo significa che per essere uguali, ogni elemento deve essere uguale e le due sequenze devono essere dello stesso tipo e avere la stessa lunghezza.

  Se non sono uguali, le sequenze vengono ordinate in base ai loro primi elementi differenti. Per esempio, `cmp([1,2,x], [1,2,y])` restituisce lo stesso risultato di `cmp(x,y)`. Se l'elemento corrispondente non esiste, viene prima ordinata la sequenza più corta (per esempio, `[1,2] < [1,2,3]`).
- Mappe (dizionari) risultano uguali se e solo se la loro lista ordinata (chiave, valore) risulta uguale.[3] Altri risultati oltre l'uguaglianza vengono risolti in modo consistente, ma non vengono altrimenti definiti.[4]
- Molti altri tipi risultano disuguali a meno che non siano lo stesso oggetto; la scelta di considerare un oggetto più piccolo o più grande di un altro è arbitraria ma consistente all'interno dell'esecuzione di un programma.

Gli operatori `in` e `not in` provano l'appartenenza ad un insieme. *x* `in` *s* valuta se è vero che *x* è membro dell'insieme *s* e falso altrimenti. *x* not in *s* restituisce la negazione di *x* in *s*. Il test sull'appartenenza ad un insieme è stato tradizionalmente legato alle sequenze; un oggetto è membro di un insieme se l'insieme è una sequenza e contiene un elemento uguale a tale oggetto. Comunque è possibile per un oggetto supportare il test di appartenenza ad un insieme senza essere una sequenza. In particolare, i dizionari supportano tale test come un modo comodo di scrivere chiavi in un dizionario; altri tipi di mappe possono seguire i seguenti comportamenti.

Per tipi lista e tupla, *x* `in` *y* risulta vero se e solo se esiste un indice *i* così che *x* `==` *y*`[`*i*`]` sia vero.

Per tipi Unicode e stringa, *x* `in` *y* risulta vero se e solo se *x* è una sottostringa di *y*. Un test equivalente è `y.find(x) != -1`. Da notare che *x* e *y* non devono necessariamente essere dello stesso tipo; di conseguenza, `u'ab' in 'abc'` restituirà `True`. Stringhe vuote vengono sempre considerate sottostringhe di qualsiasi altra stringa, così `'' in 'abc'` restituirà `True`. Modificato nella versione 2.3: Precedentemente, era richiesto che *x* fosse una stringa di lunghezza `1`.

Per classi definite dall'utente che definiscono il metodo `__contains__()`, *x* `in` *y* è vero se e solo se *y*`.__contains__(`*x*`)` risulta vero.

[3] L'implementazione lo calcola efficientemente, senza costruire liste o ordinamenti.

[4] Le prime versioni di Python utilizzavano il confronto lessicofrafico delle liste ordinate (chiave, valore), ma questo era molto costoso per la maggiorparte dei casi di confronto di uguaglianza. Una versione ancora più vecchia di Python, confrontava i dizionari solo tramite le identità, ma questo causava sorprese perché le persone si aspettavano di essere in grado di verificare se un dizionario è vuoto, confrontandolo con `{}`.

Per classi definite dall'utente che non definiscono `__contains__()` ma definiscono `__getitem__()`, *x* `in` *y* è vero se e solo se c'è un indice *i* intero non negativo in modo tale che *x* `==` *y*`[`*i*`]`, e tutti gli indici interi inferiori non sollevino l'eccezione `IndexError`. (Se viene sollevata qualsiasi altra eccezione, è come se `in` avesse sollevato tale eccezione).

L'operatore `not in` viene definito per avere il valore vero inverso di `in`.

Gli operatori `is` e `is not` controllano l'identità di un oggetto: *x* `is` *y* è vero se e solo se *x* e *y* sono lo stesso oggetto. *x* `is not` *y* produce una valore vero inverso.

5.10 Operazioni booleane

Le operazioni booleane hanno la priorità più bassa rispetto a tutte le operazioni di Python:

```
expression   ::=  or_test | lambda_form
or_test      ::=  and_test | or_test or and_test
and_test     ::=  not_test | and_test and not_test
not_test     ::=  comparison | not not_test
lambda_form  ::=  lambda [parameter_list]:  expression
```

Nel contesto delle operazioni booleane, ed anche quando le espressioni vengono usate dal flusso di controllo delle istruzioni, i seguenti valori vengono interpretati come falso: `None`, il numero zero in tutti i tipi, sequenze vuote (stringhe, tuple e liste) e mappe vuote (dizionari). Tutti gli altri valori vengono interpretati come vero.

Gli operatori `not` producono `True` se il proprio argomento è falso, altrimenti producono `False`.

L'espressione *x* `and` *y* valuta prima *x*; se *x* è falso viene restituito il suo valore; altrimenti viene valutata *y* e viene restituito il valore risultante.

L'espressione *x* `or` *y* valuta prima *x*; se *x* è vero viene restituito il suo valore; altrimenti viene valutato *y* e viene restituito il valore risultante.

(Da notare che né `and` e né `or` limitano il valore e il tipo che restituiscono come `False` o `True`, ma piuttosto restituiscono l'ultimo argomento valutato. Questo è utile in certe occasioni, per esempio se `s` è una stringa che dovrebbe essere sostituita con un valore predefinito se vuota, l'espressione `s or 'foo'` produce il valore desiderato. Dato che `not` non inventa comunque un valore, la restituzione di un valore dello stesso tipo del proprio argomento non confonde, così per esempio, `not 'foo'` produce `False`, non `''`.)

5.11 Lambda

Le forme lambda (espressioni lambda) hanno la stessa posizione sintattica delle espressioni. Sono una scorciatoia per creare funzioni anonime; gli argomenti delle espressioni lambda producono oggetti funzione, `lambda` *arguments*`:` *expression*. L'oggetto senza nome si comporta come una funzione definita con

```
def name(arguments):
    return expression
```

Vedere la sezione 7.5 per la sintassi della lista dei parametri. Da notare che le funzioni create in forma lambda non possono contenere istruzioni.

5.12 Liste di espressioni

```
expression_list  ::=  expression ( , expression )* [,]
```

Una lista di espressioni contiene almeno una virgola, producendo così una tupla. La lunghezza della tupla è il numero delle espressioni nella lista. Le espressioni vengono valutate da sinistra a destra.

La virgola finale viene richiesta solo per creare una singola tupla (conosciuta anche come *singleton*); è facoltativa

in tutti gli altri casi. Un'espressione singola non seguita da una virgola non crea una tupla, ma piuttosto produce il valore di tale espressione. (Per creare una tupla vuota, usare un paio di parentesi vuote: `()`.)

5.13 Ordine di valutazione

Python valuta le espressioni da sinistra a destra. Da notare che durante la valutazione di un assegnamento, la parte destra viene valutata prima della parte sinistra.

Nelle righe seguenti, le espressioni verranno valutate nell'ordine aritmetico dei loro suffissi:

```
expr1, expr2, expr3, expr4
(expr1, expr2, expr3, expr4)
{expr1: expr2, expr3: expr4}
expr1 + expr2 * (expr3 - expr4)
func(expr1, expr2, *expr3, **expr4)
expr3, expr4 = expr1, expr2
```

5.14 Sommario

La tabella che segue riassume la precedenza degli operatori in Python, dalla precedenza più bassa (legame debole) a quella più alta (legame forte). Gli operatori nello stesso riquadro hanno la stessa precedenza. A meno che la sintassi non sia fornita esplicitamente, gli operatori sono binari. Gli operatori nello stesso riquadro sono raggruppati da sinistra a destra (ad eccezione delle comparazioni, test incluso, che hanno la stessa precedenza e concatenamenti da sinistra a destra — vedere la sezione 5.9 — ed elevazioni a potenza, raggruppati da destra a sinistra).

Operatore	**Descrizione**
`lambda`	Espressione lambda
`or`	OR booleano
`and`	AND booleano
`not` *x*	NOT booleano
`in`, `not in`	Membership tests
`is`, `is not`	Identity tests
`<`, `<=`, `>`, `>=`, `<>`, `!=`, `==`	Comparisons
`|`	OR bit per bit
`^`	XOR bit per bit
`&`	AND bit per bit
`<<`, `>>`	Scorrimento
`+`, `-`	Addizione e sottrazione
`*`, `/`, `%`	Multiplication, division, remainder
+*x*, -*x*	Positivo, negativo
~*x*	not bit per bit
`**`	Esponente
x`.`*attribute*	Riferimento ad attributo
x`[`*index*`]`	Subscription
x`[`*index*`:`*index*`]`	Affettamento
f`(`*arguments*`...)`	Chiamata a funzione
`(`*expressions*`...)`	legame o visualizzazione di tupla
`[`*expressions*`...]`	visualizzazione di lista
`{`*key*`:`*datum*`...}`	visualizzazione di dizionario
`'`*expressions*`...'`	Conversione in stringa

CAPITOLO

SEI

Istruzioni semplici

Le istruzioni semplici sono comprese all'interno di una singola riga logica. Alcune istruzioni semplici posso trovarsi su di una singola riga, separate da punti e virgola. La sintassi per istruzioni semplici è:

```
simple_stmt  ::=  expression_stmt
                | assert_stmt
                | assignment_stmt
                | augmented_assignment_stmt
                | pass_stmt
                | del_stmt
                | print_stmt
                | return_stmt
                | yield_stmt
                | raise_stmt
                | break_stmt
                | continue_stmt
                | import_stmt
                | global_stmt
                | exec_stmt
```

6.1 Espressioni di istruzioni

Le espressioni di istruzioni vengono usate (prevalentemente in modo interattivo) per calcolare e scrivere un valore, o (di solito) per chiamare una procedura (una funzione che restituisce un risultato significativo; in Python, le procedure restituiscono il valore `None`). La sintassi di un'espressione è:

```
expression_stmt  ::=  expression_list
```

Un'espressione di istruzioni valuta la lista delle espressioni (che può essere una singola espressione).

Nella modalità interattiva, se il valore non è `None`, viene convertito in una stringa usando la funzione built-in `repr()` e la stringa risultante viene scritta sullo standard output in una riga (vedere la sezione 6.6). (Le espressioni di istruzioni che producono `None` non vengono scritte, quindi queste procedure non causano alcun output).

6.2 Istruzioni assert

Le istruzioni sono un modo utile per inserire asserzioni di controllo nel programma:

```
assert_stmt  ::=  assert expression [, expression]
```

La forma semplice, '`assert espressione`', è equivalente a

```
if __debug__:
   if not expression: raise AssertionError
```

La forma estesa, '`assert espressione1, espressione2`', è equivalente a

```
if __debug__:
   if not expression1: raise AssertionError, expression2
```

Queste equivalenze assumono che `__debug__` e `AssertionError` si riferiscano alle variabili built-in con questi nomi. Nell'implementazione corrente, la variabile built-in `__debug__` è 1 in circostanze normali, 0 quando viene richiesta un'ottimizzazione (utilizzando l'opzione -O da riga di comando). Il generatore del codice corrente non emette alcun codice per un'istruzione assert quando viene richiesta un'ottimizzazione al momento della compilazione. Si noti che è inutile includere il codice sorgente per le espressioni che falliscono in quanto verranno visualizzate come parte della traccia dello stack.

Assegnamenti in `__debug__` sono illegali. Il valore per la variabile built-in viene determinato quando parte l'interprete.

6.3 Istruzioni di assegnamento

Le istruzioni di assegnamento vengono usate per (ri)legare nomi a valori e modificare attributi o elementi di oggetti mutabili:

```
assignment_stmt  ::=  (target_list =)+ expression_list
target_list      ::=  target (, target)* [,]
target           ::=  identifier
                      | ( target_list )
                      | [ target_list ]
                      | attributeref
                      | subscription
                      | slicing
```

(Vedere la sezione 5.3 per la definizione della sintassi degli ultimi tre simboli.)

Un'istruzione di assegnamento valuta la lista delle espressioni (ricordando che questa può essere una singola espressione o una lista separata da virgole, l'ultima produce una tupla) e assegna ogni singolo oggetto risultante ad ognuno dei riferimenti forniti, da sinistra a destra.

L'assegnamento viene determinato ricorsivamente a seconda della forma del riferimento (lista). Quando un riferimento fa parte di un oggetto mutabile (un riferimento ad un attributo, una subscription o a fette), l'oggetto mutabile deve infine eseguire l'assegnamento e decidere la propria validità e può sollevare un'eccezione se l'assegnamento è inaccettabile. Le regole osservate dai vari tipi ed eccezioni sollevate vengono date con la definizione del tipo di oggetto (vedere la sezione 3.2).

L'assegnamento di un oggetto ad una lista di riferimenti viene definita ricorsivamente come segue.

- Se la lista di riferimenti è un riferimento singolo: l'oggetto viene assegnato a questo riferimento.
- Se la lista di riferimenti è una lista separata da virgole: l'oggetto deve essere una sequenza con lo stesso numero di elementi dei riferimenti della lista e gli elementi vengono assegnati, da sinistra a destra, al corrispondente riferimento. (Questa regola non è rigida come in Python 1.5; nelle precedenti versioni, l'oggetto doveva essere una tupla. Dato che le stringhe sono sequenze, un assegnamento come '`a, b = xy`' è ora legale dal momento che la lista ha la giusta lunghezza.)

L'assegnamento di un oggetto ad un singolo riferimento viene ricorsivamente definito come segue.

- Se il riferimento è un identificativo (nome):

– se il nome non si trova in un'istruzione `global` nel blocco di codice corrente; il nome viene legato all'oggetto nello spazio dei nomi corrente.
– Altrimenti: il nome viene legato all'oggetto nello spazio dei nomi globale corrente.

Se il nome era già legato viene legato nuovamente. Questo può causare il raggiungimento di zero da parte del contatore di riferimenti per l'oggetto precedentemente legato al nome, causando la disallocazione dell'oggetto e la chiamata al suo distruttore (se esiste).

- Se l'identificativo è una lista di identificativi racchiusi tra parentesi tonde o parentesi quadre: l'oggetto deve essere una sequenza con lo stesso numero di elementi di quanti sono gli identificativi della lista, e gli elementi vengono assegnati, da sinistra a destra, al corrispondente identificativo.

- Se l'identificativo è un riferimento ad attributo: viene valutata l'espressione primaria dell'identificativo che dovrebbe essere un oggetto yield con attributi assegnabili; in caso contrario, viene sollevata un'eccezione `TypeError`. Questo oggetto viene quindi assegnato al dato attributo; se non si può eseguire l'assegnamento, viene sollevata un'eccezione (di solito, ma non necessariamente, un `AttributeError`).

- Se l'identificativo è una subscription: viene valutata l'espressione primaria del riferimento che dovrebbe produrre o una sequenza mutabile (per esempio una lista) o un oggetto mappabile (per esempio un dizionario). Viene quindi valutata l'espressione subscript.

  Se la primaria è una sequenza mutabile (per esempio una lista), il subscript deve produrre un intero. Se è negativo deve essere un intero non negativo, minore della lunghezza della sequenza e la sequenza assegna l'oggetto assegnato agli elementi con il dato indice. Se l'indice è fuori intervallo, viene sollevata un'eccezione `IndexError` (l'assegnamento ad una sequenza non può aggiungere nuovi elementi alla lista).

  Se la primaria è un oggetto mappabile (per esempio un dizionario), il subscript deve avere un tipo compatibile con il tipo chiave della mappa e la mappa deve creare una coppia chiave/valore che mappa il subscript all'oggetto assegnato (se non esiste una chiave con lo stesso valore).

- Se l'identificativo è una fetta: viene valutata l'espressione primaria del riferimento. Dovrebbe produrre una sequenza mutabile (per esempio una lista). L'oggetto assegnato dovrebbe essere una sequenza dello stesso tipo. Quindi vengono valutate le espressioni inferiori e superiori legate, fintanto che sono presenti; valori predefiniti sono zero e la lunghezza della sequenza. I legami dovrebbero essere valutati in (piccoli) interi. Se entrambi i legami sono negativi, viene aggiunta la lunghezza della sequenza. I legami risultanti vengono tagliati tra zero e la lunghezza della sequenza inclusi. Alla fine, la sequenza sostituisce alla fetta gli elementi assegnati della sequenza. La lunghezza della fetta può differire dalla lunghezza della sequenza assegnata, così cambia la lunghezza della sequenza identificativo, se l'oggetto lo permette.

(Nell'implementazione corrente, la sintassi per gli identificativi diviene la stessa che per le espressioni e sintassi individuali vengono rifiutate durante la fase di generazione del codice, causando messaggi di errore dettagliati.)

AVVERTENZE: benché la definizione di assegnamento implichi che sovrapposizioni tra la parte sinistra e destra siano 'innocue' (per esempio, '`a, b = b, a`' scambia le due variabili), sovrapposizioni all'interno della collezione di variabili assegnate non sono sicure! Per esempio, il seguente programma visualizza '`[0, 2]`':

```
x = [0, 1]
i = 0
i, x[i] = 1, 2
print x
```

6.3.1 Dichiarazioni di assegnamento incrementale

L'assegnamento incrementale è la combinazione, in una singola istruzione, di un'operazione binaria e un'istruzione di assegnamento:

```
augmented_assignment_stmt  ::=  target augop expression_list
augop                      ::=  += | -= | *= | /= | %= | **=
                                |»>= | «<= | &= | ^= | =
```

(Vedere la sezione 5.3 per la definizione della sintassi per gli ultimi tre assegnamenti.)

Un assegnamento incrementale valuta l'obiettivo (che, a differenza delle normali istruzioni di assegnamento, non può essere un parametro unpacking) e l'espressione, esegue l'operazione binaria specifica per il tipo di operazione dei due operandi e assegna il risultato all'obiettivo originale. L'obiettivo viene valutato una sola volta.

Un'espressione di assegnamento incrementale come `x += 1` può essere riscritto come `x = x + 1` per ottenere la stessa cosa, ma non esattamente lo stesso effetto. Nella versione incrementale, `x` viene valutata solamente una volta. Inoltre, quando possibile, l'operazione viene eseguita *sul posto*, il che significa che invece di creare un nuovo oggetto e assegnarlo all'obiettivo, viene modificato il vecchio oggetto.

Con l'eccezione dell'assegnamento di tuple ed obiettivi multipli in una singola dichiarazione, l'assegnamento fatto attraverso una dichiarazione incrementale viene gestito nello stesso modo di un normale assegnamento. In modo simile, con l'eccezione del possibile comportamento *sul posto*, l'operazione binaria viene eseguita nello stesso modo di una normale operazione binaria.

Per obiettivi che sono riferimenti ad attributi, il valore iniziale viene ottenuto con `getattr()` ed il risultato viene assegnato con `setattr()`. Da notare che i due metodi non si riferiscono necessariamente alla stessa variabile. Quando `getattr()` fa riferimento ad una variabile di classe, `setattr()` si riferisce ancora ad una variabile istanza. Per esempio:

```
class A:
    x = 3    # variabile di classe
a = A()
a.x += 1     # scrive a.x come 4 lasciando A.x come 3
```

6.4 L'istruzione `pass`

```
pass_stmt  ::=  pass
```

`pass` è un'operazione nulla — quando viene eseguita, non succede niente. Diviene utile quando viene richiesta sintatticamente un'istruzione ma non è necessario nessun codice per l'esecuzione, per esempio:

```
def f(arg): pass    # una funzione che non fa (ancora) niente

class C: pass       # una classe senza metodi (ancora)
```

6.5 L'istruzione `del`

```
del_stmt  ::=  del target_list
```

La cancellazione viene definita ricorsivamente nello stesso modo in cui viene definito l'assegnamento. Invece di descriverla in dettaglio, si daranno alcuni accenni.

La cancellazione di una lista rimuove ricorsivamente ogni elemento, da sinistra a destra.

La cancellazione di un nome rimuove i suoi legami dallo spazio dei nomi locale o globale, a seconda se il nome si trova in un'istruzione `global` nello stesso blocco di codice. Se il nome non ha legami, verrà sollevata un'eccezione `NameError`.

Non è legale cancellare un nome dallo spazio dei nomi locale se succede ad una variabile vuota in un blocco annidato.

La cancellazione di riferimenti ad attributo, subscription e affettazioni viene passata all'oggetto primario coinvolto;

la cancellazione di un'affettazione è in generale equivalente all'assegnamento di una fetta vuota del tipo giusto (ma anche questo viene determinato dall'oggetto affettato).

6.6 L'istruzione `print`

```
print_stmt  ::=  print ( [expression (, expression)* [,]]
                 |»> expression [(, expression)+ [,]] )
```

`print` valuta un'espressione alla volta e scrive l'oggetto risultante sullo standard output (vedere sotto). Se un oggetto non è una stringa, viene prima convertito in una stringa usando le regole per la conversione. La stringa (risultante o originale) viene quindi scritta. Prima di ogni oggetto viene scritto uno spazio a meno che il sistema di output non creda che sia posizionato all'inizio di una riga. Questo è il caso (1) quando nessun carattere è stato ancora scritto sullo standard output, (2) quando l'ultimo carattere scritto sullo standard output è '`\n`', o (3) quando l'ultima operazione di scrittura sullo standard output non è stata un'istruzione `print`. (Per questa ragione in alcuni casi può essere funzionale scrivere una stringa vuota sullo standard output.) **Note:** gli oggetti che si comportano come oggetti file ma che non sono oggetti file built-in, spesso non emulano appropriatamente questo aspetto del comportamento degli oggetti file, così non è buona norma fare affidamento su questo comportamento.

Alla fine viene scritto un carattere '`\n`', a meno che l'istruzione `print` non termini con una virgola. Questa è la sola azione se la l'istruzione contiene solo la parola chiave `print`.

Lo standard output viene definito come l'oggetto file chiamato stdout nel modulo built-in `sys`. Se non esiste nessun oggetto, o se non possiede un metodo `write()`, viene sollevata un'eccezione `RuntimeError`.

`print` possiede anche una forma estesa, definita dalla seconda porzione della sintassi descritta sopra. Ci si riferisce a questa forma come "`print` chevron". In questa forma, la prima espressione dopo il simbolo `>>` deve riferirsi ad un oggetto "similfile" e specificamente ad un oggetto che abbia un metodo `write()`, come descritto sopra. Se la prima espressione valutata è `None`, viene usato `sys.stdout` come file per l'output.

6.7 L'istruzione `return`

```
return_stmt  ::=  return [expression_list]
```

`return` può solo apparire sintatticamente all'interno di una definizione di funzione, non all'interno di una definizione di classe.

Se è presente una lista di espressioni questa viene valutata, altrimenti viene restituito `None`.

`return` abbandona la chiamata alla funzione corrente, con la lista di espressioni (o `None`) come valore restituito.

Quando `return` passa il controllo oltre una istruzione `try` con una clausola `finally`, la clausola `finally` viene eseguita prima di lasciare realmente la funzione.

In una funzione generatore, non è permessa un'istruzione `return` con una `expression_list`. In questo contesto, un `return` vuoto indica che il generatore è pronto e causerà il sollevamento dell'eccezione `StopIteration`.

6.8 L'istruzione `yield`

```
yield_stmt  ::=  yield expression_list
```

L'istruzione `yield` viene usata solo quando si definisce una funzione generatore e viene usata solo nel corpo della funzione generatore. L'uso dell'istruzione `yield` nella definizione di una funzione è sufficiente a creare una funzione generatore invece che una normale funzione.

Quando viene chiamata, una funzione generatore restituisce un iteratore conosciuto come un generatore iteratore, o più comunemente, un generatore. Il corpo della funzione generatore viene eseguito da una chiamata reiterata al metodo `next()` del generatore fino al sollevamento di un'eccezione.

Quando viene eseguita l'istruzione `yield`, lo stato del generatore viene congelato ed il valore dell'`expression_list` viene restituita alla chiamata `next()`. Per "congelata" si intende che tutto lo stato locale viene conservato, inclusi i legami correnti delle variabili locali, il puntatore all'istruzione e lo stack di valutazione interno: sono salvate sufficienti informazioni, cosicché la prossima volta che viene invocato `next()`, la funzione possa procedere esattamente come se l'istruzione `yield` fosse una chiamata dall'esterno.

L'istruzione `yield` non è permessa nella clausola `try` di un costrutto `try` ... `finally`. La difficoltà è che non c'è garanzia che il generatore verrà ripreso, quindi non c'è garanzia che il blocco `finally` sia eseguito.

Note: in Python 2.2, l'istruzione `yield` è permessa solo quando viene abilitata la caratteristica dei `generatori`. Sarà sempre abilitata in Python 2.3. L'istruzione `import __future__` può essere usata per abilitare questa caratteristica:

```
from __future__ import generators
```

Vedete anche:

PEP 0255, "*Simple Generators*"
Le proposte per aggiungere generatori e l'istruzione `yield` a Python.

6.9 L'istruzione `raise`

```
raise_stmt  ::=  raise [expression [, expression [, expression]]]
```

Se non sono presenti espressioni, `raise` risolleva l'ultima espressione che era stata attivata nel corrente ambito di visibilità. Se nessuna eccezione è attiva nel corrente ambito di visibilità, viene sollevata un'eccezione che indica questo errore.

Altrimenti, `raise` valuta le espressioni per prendere tre oggetti, usando `None` come valore per le espressioni omesse. I primi due oggetti vengono usati per determinare il *tipo* ed il *valore* dell'eccezione.

Se il terzo oggetto è un'istanza, il tipo dell'eccezione è la classe dell'istanza, l'istanza stessa è il valore ed il secondo oggetto deve essere `None`.

Se il primo oggetto è una classe, questa diventa il tipo di eccezione. Il secondo oggetto viene usato per determinare il valore dell'eccezione: se è un'istanza della classe, l'istanza diventa il valore dell'eccezione. Se il secondo oggetto è una tupla, viene usato come lista di argomenti per il costruttore di classe; se è `None`, viene usato un argomento lista vuoto ed ogni altro oggetto viene trattato come singolo argomento per il costruttore. L'istanza così creata dalla chiamata del costruttore viene usata come valore dell'eccezione.

Se è presente un terzo oggetto e non `None`, deve essere un oggetto traceback (vedere la sezione 3.2) e viene sostituito al posto della posizione corrente come il posto dov'è avvenuta l'eccezione. Se il terzo oggetto è presente e non è `None`, viene sollevata un'eccezione `TypeError`. Le tre forme di espressione di `raise` sono utili per risollevare nuovamente un'eccezione in modo trasparente in una clausola except, ma `raise` senza espressioni dovrebbe essere preferito se l'eccezione che deve essere risollevata era l'eccezione attiva più recente nel corrente ambito di visibilità.

Ulteriori informazioni sulle eccezioni possono essere trovate nella sezione 4.2 e informazioni sulla gestione delle eccezioni si trovano nella sezione 7.4.

6.10 L'istruzione `break`

```
break_stmt  ::=  break
```

`break` può trovarsi sintatticamente annidato in un ciclo `for` o `while`, ma non annidato in una funzione o definizione di classe senza uno di quei due cicli.

Termina il ciclo più vicino, saltando la clausola facoltativa `else` se il ciclo ne ha una.

Se un ciclo `for` finisce con un `break`, l'obiettivo del controllo del ciclo acquisisce il suo valore corrente.

Quando `break` passa il controllo fuori dall'istruzione `try` con una clausola finale, questa clausola `finally` viene eseguita prima che sia abbandonato realmente il ciclo.

6.11 L'istruzione `continue`

```
continue_stmt  ::=  continue
```

`continue` può solo trovarsi sintatticamente annidato in un ciclo `for` o `while`, ma non annidato in una funzione o in una definizione di classe o in un'istruzione `try` all'interno di quel ciclo.[1] Prosegue con il ciclo sussessivo.

6.12 L'istruzione `import`

```
import_stmt  ::=  import module [as name] ( , module [as name] )*
                  | from module import identifier [as name]
                   ( , identifier [as name] )*
                  | from module import *
module       ::=  (identifier .)* identifier
```

Le istruzioni import vengono eseguite in due passi: (1) cercare un modulo, ed inizializzarlo se necessario; (2) definire un nome o nomi nello spazio dei nomi locale (nel medesimo ambito di visibilità in cui viene eseguita l'istruzione `import`). La prima forma (senza `from`) ripete questi passi per ogni identificatore nella lista. La forma con `from`) svolge il passo (1) una volta e quindi svolge ripetutamente il passo 2.

In questo contesto, per "inizializzare" una built-in, un modulo estensione significa chiamare una funzione di inizializzazione che deve essere fornita dal modulo per l'uso (nelle implementazioni di riferimento, il nome della funzione è ottenuto antecedendo la stringa "init" al nome del modulo); per "inizializzare" un modulo codificato Python significa eseguire il corpo del modulo.

Il sistema mantiene una tavola dei moduli che sono stati o saranno inizializzati, indicizzandoli per nome. Questa tavola è accessibile come `sys.modules`. Quando il nome di un modulo viene trovato in questa tavola, il passo (1) è finito. Altrimenti, inizia la ricerca per la definizione di un modulo. Quando un modulo è stato trovato, viene caricato. Dettagli della ricerca di moduli e dei processi di caricamento sono implementazioni specifiche delle diverse piattaforme. Questo di solito comporta la ricerca per un modulo "built-in" con un dato nome e quindi la ricerca di una lista di posizione data come `sys.path`.

Se viene trovato un modulo built-in, il suo codice di inizializzazione built-in viene eseguito ed il passo (1) è terminato. Se non ci sono file che hanno corrispondenze, viene sollevata un'eccezione `ImportError`. Se viene trovato un file, viene analizzato, restituendo un blocco di codice eseguibile. Se avviene un errore di sintassi, viene sollevata un'eccezione `SyntaxError`. Altrimenti, viene creato un modulo vuoto con un dato nome ed inserito nella tavola dei moduli, e quindi viene eseguito il blocco di codice nel contesto di questo modulo. Eccezioni durante questa esecuzione terminano il passo (1).

Quando il passo (1) finisce senza avere sollevato eccezioni, il passo (2) può iniziare.

La prima forma dell'istruzione `import` collega il nome del modulo nello spazio dei nomi locale all'oggetto modulo, e quindi va ad importare il prossimo identificatore, se c'è nè qualcuno. Se il nome del modulo viene seguito da `as`, il nome successivo ad `as` viene usato come spazio dei nomi locale per il modulo.

La forma `from` non collega il nome del modulo: attraversa la lista degli identificatori, guarda in ognuno di loro con riguardo al modulo trovato al passo (1) e collega il nome allo spazio dei nomi locale all'oggetto così trovato. Come con la prima forma di `import`, un nome locale alternativo può essere fornito specificndo `as` nome locale. Se un nome non viene riscontrato, viene sollevata un'eccezione `ImportError`. Se la lista degli identificatori viene sostituita da una stella ('`*`'), tutti i nomi pubblici definiti nel modulo vengono legati allo spazio dei nomi locale dell'istruzione `import`.

I *nomi pubblici* definiti da un modulo vengono determinati dalla ricerca nello spazio dei nomi dei moduli per una variabile chiamata `__all__`; se definita, deve essere una sequenza di stringhe che rappresentano nomi definiti o

[1] Può trovarsi all'interno di una clausola `except` o `else`. La restrizione che avviene nella clausola `try` è dovuta alla pigrizia di chi l'ha implementata e potrà, in futuro, essere sistemata.

importati da quel modulo. I nomi dati in `__all__` vengono considerati pubblici e sono necessari perché esistano. Se `__all__` non viene definito, la gamma dei nomi pubblici include tutti i nomi trovati nello spazio dei nomi del modulo che non iniziano con un carattere di sottolineatura ('_'). `__all__` dovrebbe contenere l'intera API pubblica. Questo viene inteso per evitare l'esportazione accidentale di elementi che non sono parte dell'API (come moduli della libreria che erano stati importati e usati senza il modulo).

La forma `from` con '`*`' può solo trovarsi nell'ambito di visibilità del modulo. Se la forma di import col carattere jolly — '`import *`' — viene usata in una funzione e la funzione contiene o è un blocco annidato con variabili libere, il compilatore solleverà un'eccezione `SyntaxError`.

Nomi dei moduli gerarchici: quando il nome dei moduli contiene uno o più punti, il percorso di ricerca del modulo viene effettuato differentemente. La sequenza di identificatori sopra l'ultimo punto viene usata per trovare un "package"; l'identificatore finale viene quindi cercato all'interno del package. Un package è generalmente una sottodirectory di una directory su `sys.path` che ha un file '__init__.py'.. [XXX Non può si può parlare di questo ora: vedere l'URL http://www.python.org/doc/essays/packages.html per maggiori dettagli, anche su come la ricerca dei moduli lavora dall'interno di un package.]

La funzione built-in `__import__()` viene fornita per supportare applicazioni che determinano quali moduli necessitano di essere caricati dinamicamente; riferirsi a Funzioni built-in nel *La libreria di riferimento di Python* per ulteriori informazioni.

6.12.1 Istruzioni future

Un'*istruzione future* è una direttiva per il compilatore per un modulo particolare che dovrebbe essere compilato usando una sintassi o una semantica che sarà disponibile in una versione futura di Python. L'istruzione future è intesa per migrare facilmente ad una versione futura di Python che introduce cambiamenti incompatibili per il linguaggio. Permette l'uso delle nuove caratteristiche su una base di propri moduli, prima che la release in cui si trova la caratteristica diventi standard.

```
future_statement  ::=  from __future__ import feature [as name]
                       (, feature [as name])*
feature           ::=  identifier
name              ::=  identifier
```

Un'istruzione future deve apparire vicino la parte alta del modulo. Le uniche righe che possono apparire prima di una dichiarazione future sono:

- il modulo docstring (se presente),
- commenti,
- righe vuote ed
- altre istruzioni future.

Le caratteristiche riconosciute da Python 2.3 sono '`generators`', '`division`' e '`nested_scopes`'. I '`generators`' e i '`nested_scopes`' sono ridondanti in 2.3 perché sono sempre abilitati.

Un'istruzione future viene riconosciuta e trattata al momento della compilazione: i cambiamenti alla semantica del costrutto base vengono regolarmente implementati per generare codice differente. Potrebbe essere il caso che una nuova caratteristica introduca nuove incompatibilità sintattiche (come una nuova parola riservata), in quel caso il compilatore può avere bisogno di analizzare il modulo in modo differente. Queste decisioni non possono essere prese fino a che il programma è in esecuzione.

Per ogni release rilasciata, il compilatore sa quali nomi di caratteristiche sono stati definiti e solleva un'eccezione per la compilazione a runtime se una dichiarazione future contiene una caratteristica non conosciuta da esso.

Le semantiche di esecuzione diretta sono le stesse per ogni istruzione importante: c'è un modulo `__future__` standard, descritto successivamente e verrà importato nel modo usuale al momento in cui l'istruzione future verrà eseguita.

La parte interessante della semantica a runtime dipende dalla specifica caratteristica abilitata dall'istruzione future.

Notare che non c'è niente di speciale nella dichiarazione:

```
import __future__ [as name]
```

Questa non è un'istruzione future, è un'istruzione di importazione ordinaria con nessuna speciale semantica o restrizione di sintassi.

Il codice compilato da un'istruzione exec o chiamato da una funzione buit-in `compile()` ed `execfile()`, che capitano nel modulo `M`, contenente una dichiarazione future, userà, in modo predefinito, la nuova sintassi o semantica associata con la dichiarazione future. Questo può, avviando Python 2.2, essere controllato dagli argomenti facoltativi per `compile()` — per dettagli vedere la documentazione di questa funzione nella libreria di riferimento.

Una dichiarazione future scritta in un prompt interattivo resterà effettiva per il resto della sessione interpretata. Se un interprete viene avviato con l'opzione **-i**, verrà passato il nome di uno script da eseguire e se lo script include un'istruzione future, questa avrà effetto nelle sessione interattive avviate dopo l'esecuzione dello script.

6.13 L'istruzione `global`

```
global_stmt  ::=  global identifier (, identifier)*
```

L'istruzione `global` è un'istruzione che viene mantenuta per l'intero blocco di codice corrente. Questo significa che gli identificatori elencati verranno interpretati come globali. Dovrebbe essere impossible assegnare una variabile globale senza `global`, sebbene le variabili libere possano fare riferimento a globalità senza essere state dichiarate `global`.

I nomi elencati in un'istruzione `global` non devono essere usati nello stesso blocco di codice testuale che precede quell'istruzione `global`.

I nomi elencati nell'istruzione `global` non devono essere definiti come parametri formali o in un ciclo `for` per il controllo dell'obiettivo, in una definizione di classe (`class`), in una definizione di funzione, o un'istruzione `import`.

(La corrente implementazione non ha le ultime due restrizioni, ma i programmi non devono abusare di questa libertà, come future implementazioni potrebbero imporre o cambiare silenziosamente il significato del programma.)

Nota per i programmatori: `global` è una direttiva per il parser. Questo applica solo il codice analizzato nello stesso istante dell'istruzione `global`. In particolare, un'istruzione `global` contenuta in un'istruzione `exec` non influenza il blocco di codice *contenente* l'istruzione `exec`, ed il codice contenuto nell'istruzione `exec` non viene influenzato dall'istruzione `global` nel codice che contiene l'istruzione `exec`. Lo stesso principio si applica alle funzioni `eval()`, `execfile()` e `compile()`.

6.14 L'istruzione `exec`

```
exec_stmt  ::=  exec expression [in expression [, expression]]
```

Questa istruzione supporta l'esecuzione dinamica di codice Python. La prima espressione dovrebbe valutare se sia una stringa, un file oggetto aperto o un oggetto codice. Se è una stringa, viene analizzata come un insieme di istruzioni Python che quindi vanno eseguite (a meno che non avvengano errori di esecuzione). Se è un file aperto, viene analizzato prima della fine del file EOF ed eseguito. Se è un oggetto codice viene semplicemente eseguito.

In tutti i casi, se le parti facoltative vengono omesse, il codice viene eseguito nella forma corrente. Se viene specificata solo la prima espressione dopo `in`, dovrebbe essere un dizionario, che verrà usato sia per la variabili globali che per quelle locali. Se sono date due espressioni, devono essere entrambe dizionari e saranno usati rispettivamente per la variabili globali e locali.

Come effetto collaterale, un'implementazione può inserire ulteriori chiavi nei dizionari forniti oltre le loro corrispondenze ai nomi di variabile impostate dal codice eseguito. Per esempio, l'implementazione corrente può

aggiungere un riferimento al dizionario del modulo built-in `__builtin__` sotto la chiave `__builtins__` (!).

Suggerimento per i programmatori: la valutazione dinamica delle espressioni viene supportata dalla funzione built-in `eval()`. Le funzioni built-in `globals()` e `locals()` restituiscono rispettivamente il corrente dizionario globale e locale, che può essere utile per passare valori nell'uso di `exec`.

CAPITOLO

SETTE

Istruzioni composte

Le istruzioni composte contengono (gruppi di) altre istruzioni; influenzano o controllano in qualche modo l'esecuzione di queste altre istruzioni. In generale, le istruzioni composte durano per molte righe, sebbene in semplici forme un'istruzione composta completa possa essere contenuta in una sola riga.

Le istruzioni `if`, `while` e `for` implementano costrutti tradizionali di controllo di flusso. `try` specifici gestori per eccezioni e/o codice più pulito per un gruppo di istruzioni. Anche le funzioni e le definizioni di classe sono istruzioni composte.

Le dichiarazioni composte consistono in una o più 'clausole'. Una clausola consiste di un'intestazione e di una 'suite'. Le clausole d'intestazione di una particolare istruzione composta sono tutte allo stesso livello di indentazione. Ogni clausola d'intestazione inizia con una parola chiave identificabile univocamente e finisce con il carattere due punti. Una suite è un gruppo di istruzioni controllate da una clausola. Una suite può essere composta da una o più semplici istruzioni poste sulla stessa riga dell'intestazione, separate da un carattere due punti, seguendo i due punti dell'intestazione, o può essere composta da una o più istruzioni indentate sulle righe susseguenti. Solo l'ultima forma di composizione della suite può contenere istruzioni composte; quello che segue è illegale, principalmente perché non sarebbe chiaro a quale clausola `if` e seguente clausola `else` dovrebbe appartenere:

```
if test1: if test2: print x
```

Si noti anche che i due punti uniscono in modo più costrittivo che il punto in questo contesto, così come quello nel seguente esempio, una, tutte o nessuna delle istruzioni `print` stampate viene eseguita:

```
if x < y < z: print x; print y; print z
```

Riassumendo:

```
compound_stmt  ::=  if_stmt
                    | while_stmt
                    | for_stmt
                    | try_stmt
                    | funcdef
                    | classdef
suite          ::=  stmt_list NEWLINE | NEWLINE INDENT statement+ DEDENT
statement      ::=  stmt_list NEWLINE | compound_stmt
stmt_list      ::=  simple_stmt (; simple_stmt)* [;]
```

Si noti che l'istruzione finisce sempre con una `NEWLINE` seguita possibilmente da una `DEDENT` . Si noti anche che pure la clausola facoltativa continuativa, inizia con una parola chiave che non può avviare un'istruzione, così non vi sono ambiguità (il problema dell"'`else` pendente", risolto in Python dalla richiesta di annidamento dell'istruzione `if` che è indentata).

La formattazione delle regole grammaticali nella seguente sezione mette ogni clausola in una riga separata, per chiarezza.

7.1 L'istruzione `if`

L'istruzione `if` viene usata per esecuzioni condizionali:

```
if_stmt  ::=  if expression :  suite
              ( elif expression :  suite )*
              [else :  suite]
```

Seleziona esattamente una delle suite per valutare l'espressione una per una fino a che una non risulta vera (vedere la sezione 5.10 per la definizione di vero e falso); quindi quella suite viene eseguita (e nessun altra parte dell'istruzione `if` viene eseguita o valutata). Se tutte le espressioni sono false, viene eseguita la suite della clausola `else`, se presente.

7.2 L'istruzione `while`

L'istruzione `while` viene usata per esecuzioni ripetute fino a che un'espressione non risulti vera:

```
while_stmt  ::=  while expression :  suite
                 [else :  suite]
```

Questo esegue ripetutamente l'epressione e, se risulta vera, esegue la prima suite; se l'espressione risulta falsa (potrebbe essere la prima volta che viene provata) la suite della clausola `else`, se presente, viene eseguita ed il ciclo termina.

Un'istruzione `break` eseguita nella prima suite termina il ciclo senza eseguire la clausola `else` della suite. Un'istruzione `continue`, eseguita nella prima suite, salta il resto della suite e ritorna indietro per eseguire nuovamente l'espressione.

7.3 L'istruzione `for`

L'istruzione `for` viene usata per iterare attraverso gli elementi di una sequenza (come una stringa, una tupla o una lista) o altri oggetti iterabili:

```
for_stmt  ::=  for target_list in expression_list :  suite
               [else :  suite]
```

L'espressione list viene valutata una volta; dovrebbe produrre una sequenza. La suite viene quindi eseguita una volta per ogni elemento nella sequenza, in ordine ascendente rispetto all'indice. Ogni elemento viene assegnato a turno alla lista obiettivo, usando le regole standard dell'assegnazione e quindi viene eseguita la suite. Quando gli elementi sono esauriti (quando la sequenza è vuota questo accade immediatamente), la suite nella clausola `else`, se presente, viene eseguita ed il ciclo termina.

Un'istruzione `break` eseguita nella prima suite finisce il ciclo senza eseguire le clausole `else` della suite. Un'istruzione `continue` eseguita nella prima suite salta il resto della suite e continua con l'elemento successivo, o con la clausola `else` se il prossimo elemento non esiste.

La suite può assegnare alla variabile (i) la lista obiettivo; questo non influenzerà l'elemento successivo assegnato alla stessa suite.

La lista obiettivo non viene cancellata quando il ciclo è finito, ma se la sequenza è vuota, non verrà assegnata a nessuna variabile per tutto il ciclo. Suggerimento: la funzione built-in `range()` restituisce una sequenza di interi utilizzabili per emulare l'effetto del Pascal `for i := a to b do`; per esempio, `range(3)` restituisce la lista `[0, 1, 2]`.

Avvertenze: C'è una finezza quando la sequenza viene modificata dal ciclo (questo può avvenire solamente per sequenze modificabili, per esempio nel caso di liste). Un contatore interno viene usato per tenere traccia del prossimo elemento usato e viene incrementato ad ogni iterazione. Quando questo iteratore ha raggiunto la lunghezza della sequenza il ciclo finisce. Questo significa che se la suite cancella l'elemento corrente (o il precedente) dalla sequenza, il prossimo elemento verrà saltato (fino a che non acquisisce l'indice dell'elemento corrente che è stato già trattato). In modo simile, se la suite inserisce un elemento nella sequenza prima dell'elemento corrente, l'ele-

mento corrente verrà trattato ancora la prossima volta attraverso il ciclo. Questo può portare a bug fastidiosi che possono essere annullati facendo una copia temporanea dell'intera sequenza,per esempio,

```
for x in a[:]:
    if x < 0: a.remove(x)
```

7.4 L'istruzione `try`

L'istruzione `try` specifica i gestori delle eccezioni e/o le azioni di pulizia del codice per un gruppo di istruzioni:

```
try_stmt      ::=  try_exc_stmt | try_fin_stmt
try_exc_stmt  ::=  try :  suite
                   (except [expression [, target]] :  suite)+
                   [else :  suite]
try_fin_stmt  ::=  try :  suite finally :  suite
```

Ci sono due forme di istruzioni `try`: `try`...`except` e `try`...`finally`. Queste forme non possono essere mischiate (ma possono essere annidate in ogni altra).

La forma `try`...`except` specifica uno o più gestori di eccezioni (la clausola `except`). Quando non avvengono eccezioni nella clausola `try`, non viene eseguito nessun gestore di eccezioni. Quando viene sollevata un'eccezione nella suite `try`, viene avviata la ricerca per un gestore di eccezioni. Questa ricerca ispeziona le clausole except una ad una finché non ne viene trovata una che verifica l'eccezione. Una clausola per un'eccezione expressionless, se presente, deve essere l'ultima; questa verifica ogni eccezione. Per una clausola except con un'espressione, viene cercata quell'espressione e la clausola cerca l'eccezione se il risultante oggetto è "compatibile" con l'eccezione. Un oggetto è compatibile con un'eccezione se quest'ultima è anche l'oggetto che identifica l'eccezione, o (per eccezioni che sono anche classi) si intende una classe base di una eccezione, o è una tupla contenente un elemento che è compatibile con l'eccezione. Notare che l'identità dell'oggetto deve corrispondere, per esempio deve essere lo stesso oggetto, non solamente un oggetto con lo stesso valore.

Se nessuna clausola except corrisponde all'eccezione, la ricerca per un gestore di eccezione continua nel codice vicino e sulla pila invocata.

Se la valutazione di un'espressione nell'intestazione di una clausola except solleva un'eccezione, la ricerca originale per un gestore viene cancellata e parte una ricerca per una nuova eccezione nel codice vicino e sulla pila chiamata (quest'ultima viene trattata come se l'intera istruzione `try` sollevasse un'eccezione).

Quando una clausola except di ricerca viene trovata, i parametri dell'eccezione vengono assegnati alla destinazione specificata in quella clausola except, se presente, e le clausole except della suite vengono eseguite. Tutte le clusole except devono avere un blocco eseguibile. Quando la fine di questo blocco viene raggiunta, l'esecuzione continua normalmente dopo l'intera istruzione try. (Questo significa che se due gestori annidati esistono per la stessa eccezione e l'eccezione ha luogo nella clausola try del gestore nascosto, il gestore esterno non riuscirà a gestire l'eccezione.)

Prima che una clausola except della suite venga eseguita, i dettagli dell'eccezione vengono assegnati alle tre variabili del modulo `sys`: `sys.exc_type` riceve l'oggetto che identifica l'eccezione; `sys.exc_value` riceve i parametri dell'eccezione; `sys.exc_traceback` riceve un oggetto traceback (vedere la sezione 3.2) identificando nel programma il punto dove è stata sollevata l'eccezione. Questi dettagli sono anche disponibili attraverso la funzione `sys.exc_info()`, che restituisce una tupla (*exc_type*, *exc_value*, *exc_traceback*). L'uso delle variabili corrispondenti è deprecato in favore di questa funzione, da quando il loro uso è insicuro in un programma che fa uso dei thread. Come per Python 1.5, le variabili vengono riportate al loro precedente valore (prima della chiamata) quando ritornano da una funzione che ha gestito un'eccezione.

La clausola facoltativa `else` viene eseguita se e quando il controllo di flusso va oltre la fine della clausola `try`.[1] Le eccezioni della clausola `else` non vengono gestite dalle precedenti clausole `except`.

La forma `try`...`finally` specifica un gestore di pulizia. La clausola `try` viene eseguita. Quando non vengono sollevate eccezioni, la clausola `finally` viene eseguita. Quando viene sollevata un'eccezione nella clausola

[1] Allo stato attuale, il controllo di "flusso va oltre la fine", eccetto il caso in cui venga sollevata un'eccezione o l'esecuzione di un'istruzione `return`, `continue` o `break`.

`try`, l'eccezione viene temporaneamente salvata, la clausola `finally` viene eseguita e quindi l'eccezione salvata viene sollevata di nuovo. Se la clausola `finally` solleva un'altra eccezione o esegue un'istruzione `return` o `break`, l'eccezione salvata è persa. Un'istruzione `continue` è illegale in una clausola `finally`. (La ragione è un problema con la corrente implementazione — in futuro questa restrizione potrebbe essere rimossa). L'informazione sull'eccezione non è disponibile al programma durante l'esecuzione della clausola `finally`.

Quando un'istruzione `return`, `break` o `continue` viene eseguita nella suite `try` di un'istruzione `try...finally`, viene eseguita anche la clausola `finally`. (La ragione è un problema con la corrente implementazione — in futuro questa restrizione potrebbe essere rimossa).

Ulteriori informazioni sulle eccezioni possono essere trovate nella sezione 4.2 e informazioni sull'uso dell'istruzione `raise` per generare eccezioni possono essere trovate nella sezione 6.9.

7.5 Definizioni di funzione

Una definizione di funzione definisce un oggetto funzione definito dall'utente (vedere la sezione 3.2):

```
funcdef         ::=  def funcname ( [parameter_list] ) :  suite
parameter_list  ::=  (defparameter ,)*
                     (* identifier [, ** identifier]
                     | ** identifier | defparameter [,])
defparameter    ::=  parameter [= expression]
sublist         ::=  parameter (, parameter)* [,]
parameter       ::=  identifier | ( sublist )
funcname        ::=  identifier
```

Una definizione di funzione è un'istruzione eseguibile. La sua esecuzione collega il nome della funzione nello spazio dei nomi locale corrente alla funzione oggetto (un wrapper attorno al codice eseguibile per la funzione). Questa funzione oggetto contiene un riferimento al corrente spazio dei nomi globale come lo spazio dei nomi globale che deve essere usato quando viene chiamata la funzione.

La definizione della funzione non esegue il corpo della funzione; questa viene eseguita solo quando viene chiamata la funzione.

Quando uno o più parametri di alto livello hanno la forma *parameter* `=` *expression*, si dice che la funzione ha "i valori dei parametri predefiniti". Per un parametro con un valore predefinito, l'argomento corrispondente può essere omesso dalla chiamata, in quel caso il parametro predefinito viene sostituito. Se il parametro ha un valore predefinito, tutti i parametri che seguono devono avere anche loro un valore predefinito — questa è una restrizione sintattica che non viene espressa dalla grammatica.

I valori dei parametri predefiniti vengono valutati quando la definizione di funzione viene eseguita. Questo significa che l'espressione viene valutata una volta, quando la funzione viene definita e quindi il solito valore "precalcolato" viene usato per ogni chiamata. Questo è particolarmente importante per capire quando un parametro predefinito è un oggetto mutabile, come una lista o un dizionario: se la funzione modifica l'oggetto (per esempio aggiungendo un elemento ad una lista), il valore predefinito viene in effetti modificato. Questo è, generalmente, quello che intendevamo. Un modo per aggirare questo comportamento è usare `None` come valore predefinito e provarlo esplicitamente nel corpo della funzione, per esempio:

```
def whats_on_the_telly(penguin=None):
    if penguin is None:
        penguin = []
    penguin.append("property of the zoo")
    return penguin
```

Le semantiche delle funzioni di chiamata vengono descritte in modo più dettagliato nella sezione 5.3.4. Una funzione chiama sempre i valori assegnati a tutti i parametri menzionati nella lista dei parametri, insieme agli argomenti posizionali, dagli argomenti delle parole chiave. Se è presente la forma "`*identifier`", viene inizializzata come una tupla, ricevendo ogni parametro posizionale eccedente, predefinita come una tupla vuota. Se invece è presente la forma "`**identifier`", viene inizializzata come un nuovo dizionario che riceverà ogni

argomento delle parole chiave in eccesso, predefinito come un nuovo dizionario vuoto.

È anche possibile creare funzioni anonime (funzioni non legate ad un nome), da usare immediatamente nelle espressioni. Queste usano le forme lambda, descritte nella sezione 5.11. Notare che la forma lambda è solo una scorciatoia per una definizione semplificata di una funzione; una funzione definita in un'istruzione "`def`" può essere fornita o assegnata ad un altro nome proprio come una funzione definita nella forma lambda. La forma "`def`" è attualmente la più potente da quando permette l'esecuzione di istruzioni multiple.

Nota per i programmatori: Le funzioni sono oggetti di classi primarie. Una forma "`def`" eseguita all'interno di una definizione di funzione, definisce una funzione locale che può essere restituita o fornita passata. Le variabili libere, usate nelle funzioni annidate, possono accedere alle variabili locali delle funzioni che contengono il def. Vedere la sezione 4.1 per i dettagli.

7.6 Definizioni di classe

Una definizione di classe definisce un oggetto classe (vedere la sezione 3.2):

```
classdef     ::=  class classname [inheritance] :  suite
inheritance  ::=  ( [expression_list] )
classname    ::=  identifier
```

Una definizione di classe è un'istruzione eseguibile. Prima valuta la lista ereditata, se presente. Ogni elemento nella lista ereditata dovrebbe valutare un oggetto classe o una classe tipo che ne permetta la derivazione. La suite delle classi viene quindi eseguita in una nuova cornice di esecuzione (vedere la sezione 4.1), usando un nuovo spazio dei nomi creato localmente e lo spazio dei nomi globale originale. (Di solito, la suite contiene solo le definizione delle funzioni.) Quando la suite delle classi finisce l'esecuzione, la sua cornice di esecuzione viene scaricata, ma viene salvato il suo spazio dei nomi locale. Una oggetto classe viene quindi creato usando la lista ereditata per le classi base e lo spazio locale dei nomi viene salvato per il dizionario degli attributi. Il nome della classe è legato a questa classe oggetto nello spazio dei nomi originale.

Nota per i programmatori: Le variabili definite nelle definizioni di classe sono variabili di classe; vengono condivise da tutte le istanze. Per definire le variabili istanza, deve essere specificato un valore nel metodo `__init__()` o in un altro metodo. Sia la classe che la variabile istanza sono accessibili attraverso la notazione "`self.name`" ed una variabile istanza nasconde una variabile di classe con lo stesso nome quando vi si accede in questo modo. Le variabili di classe con valori immutabili possono essere usate come predefinite per variabili istanza. Per le classi di nuovo stile, i descrittori possono essere usati per creare variabili d'istanza con differenti dettagli d'implementazione.

CAPITOLO

OTTO

Componenti di alto livello

L'interprete Python può prendere il proprio input da svariate sorgenti: da uno script passatogli come standard input o come argomento di un programma, scritto interattivamente, dal sorgente di un modulo, etc.. Questo capitolo mostra la sintassi usata in questi casi.

8.1 Programmi completi in Python

Mentre le specifiche di un linguaggio non hanno la necessità di prescrivere come l'interprete del linguaggio debba essere invocato, è però utile avere una nozione di un programma Python completo. Un programma Python completo viene eseguito in un ambiente inizializzato in modo minimale: tutti i moduli built-in e standard sono disponibili, ma non sono stati inizializzati, con l'eccezione di `sys` (vari servizi di sistema), `__builtin__` (funzioni built-in, eccezioni e `None`) e `__main__`. L'ultimo viene usato per fornire lo spazio dei nomi locale e globale per l'esecuzione del programma completo.

La sintassi per un programma Python completo è come quella per i file in input, descritta nella prossima sezione.

L'interprete può essere invocato anche in modo interattivo; in questo caso, non legge ed esegue un programma completo ma legge ed esegue un'istruzione (possibilmente composta) alla volta. L'ambiente iniziale è identico a quello del programma completo; ogni istruzione viene eseguita nello spazio dei nomi di `__main__`.

Sotto UNIX, un programma completo può essere passato all'interprete in tre forme: con l'opzione **-c** *string* da riga di comando, come un file passato come primo argomento da riga di comando, o come standard input. Se il file o lo standard input sono un dispositivo tty, l'interprete entra nel modo interattivo; altrimenti esegue il file come un programma completo.

8.2 File in input

Tutti gli input letti da un file non interattivo hanno la stessa forma:

```
file_input  ::=  (NEWLINE | statement)*
```

Questa sintassi viene usata nelle seguenti situazioni:

- quando analizza un programma Python completo (da un file o da una stringa);
- quando analizza un modulo;
- quando analizza una stringa passata all'istruzione `exec`;

8.3 Input interattivo

L'input in modalità interattiva viene analizzato usando la seguente grammatica:

```
interactive_input  ::=  [stmt_list] NEWLINE | compound_stmt NEWLINE
```

Si noti che un'istruzione composta di alto livello può essere seguita da una riga vuota in modalità interattiva; questo è necessario per aiutare l'analizzatore a trovare la fine dell'input.

8.4 Espressioni in input

Ci sono due forme di espressioni in input. Ignorano entrambe il primo spazio vuoto. La stringa per l'argomento di `eval()` deve avere la seguente forma:

```
eval_input  ::=  expression_list NEWLINE*
```

La riga in input letta da `input()` deve avere la seguente forma:

```
input_input  ::=  expression_list NEWLINE
```

Nota: per leggere la riga in input 'raw' (NdT: 'grezza') senza interpretazione, si può usare la funzione built-in `raw_input()` o il metodo `readline()` degli oggetti file.

APPENDICE

A

Storia e licenza

A.1 Storia del software

Python è stato creato agli inizi degli anni 90 da Guido van Rossum al Centro di Matematica di Stichting (CWI, si veda http://www.cwi.nl/) nei Paesi Bassi, come successore di un linguaggio chiamato ABC. Guido rimane l'autore principale di Python, anche se molti altri vi hanno contribuito.

Nel 1995, Guido ha continuato il suo lavoro su Python presso il Centro Nazionale di Ricerca (CNRI, si veda http://www.cnri.reston.va.us/) in Reston, Virginia, dove ha rilasciato parecchie versioni del software.

Nel maggio 2000, Guido e la squadra di sviluppo di Python si spostano in BeOpen.com per formare la squadra PythonLabs di BeOpen. Nell'ottobre dello stesso anno, la squadra di PythonLabs diventa la Digital Creations (ora Zope Corporation; si veda http://www.zope.com/). Nel 2001, viene fondata la Python Software Foundation (PSF, si veda http://www.python.org/psf/), un'organizzazione senza scopo di lucro creata specificamente per detenere la propria proprietà intellettuale di Python. Zope Corporation è un membro sponsorizzato dalla PSF.

Tutti i rilasci di Python sono Open Source (si veda http://www.opensource.org/ per la definizione di "Open Source"). Storicamente la maggior parte, ma non tutti i rilasci di Python, sono stati GPL-compatibili; la tabella seguente ricapitola i vari rilasci.

Versione	Derivata da	Anno	Proprietario	GPL compatibile?
0.9.0 thru 1.2	n/a	1991-1995	CWI	si
1.3 thru 1.5.2	1.2	1995-1999	CNRI	si
1.6	1.5.2	2000	CNRI	no
2.0	1.6	2000	BeOpen.com	no
1.6.1	1.6	2001	CNRI	no
2.1	2.0+1.6.1	2001	PSF	no
2.0.1	2.0+1.6.1	2001	PSF	si
2.1.1	2.1+2.0.1	2001	PSF	si
2.2	2.1.1	2001	PSF	si
2.1.2	2.1.1	2002	PSF	si
2.1.3	2.1.2	2002	PSF	si
2.2.1	2.2	2002	PSF	si
2.2.2	2.2.1	2002	PSF	si
2.2.3	2.2.2	2002-2003	PSF	si
2.3	2.2.2	2002-2003	PSF	si
2.3.1	2.3	2002-2003	PSF	si
2.3.2	2.3.1	2003	PSF	si
2.3.3	2.3.2	2003	PSF	si
2.3.4	2.3.3	2004	PSF	si

Note: GPL-compatibile non significa che viene distribuito Python secondo i termini della GPL. Tutte le licenze di Python, diversamente dalla GPL, permettono di distribuire una versione modificata senza rendere i vostri cambiamenti open source. Le licenze GPL-compatibili permettono di combinare Python con altro software rilasciato sotto licenza GPL; le altre no.

Un ringraziamento ai tanti volontari che, lavorando sotto la direzione di Guido, hanno reso possibile permettere questi rilasci.

A.2 Termini e condizioni per l'accesso o altri usi di Python (licenza d'uso, volutamente non tradotta)

PSF LICENSE AGREEMENT FOR PYTHON 2.3.4

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 2.3.4 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, PSF hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 2.3.4 alone or in any derivative version, provided, however, that PSF's License Agreement and PSF's notice of copyright, i.e., "Copyright © 2001-2004 Python Software Foundation; All Rights Reserved" are retained in Python 2.3.4 alone or in any derivative version prepared by Licensee.

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 2.3.4 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 2.3.4.

4. PSF is making Python 2.3.4 available to Licensee on an "AS IS" basis. PSF MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, PSF MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 2.3.4 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.3.4 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.3.4, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between PSF and Licensee. This License Agreement does not grant permission to use PSF trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By copying, installing or otherwise using Python 2.3.4, Licensee agrees to be bound by the terms and conditions of this License Agreement.

BEOPEN.COM LICENSE AGREEMENT FOR PYTHON 2.0
BEOPEN PYTHON OPEN SOURCE LICENSE AGREEMENT VERSION 1

1. This LICENSE AGREEMENT is between BeOpen.com ("BeOpen"), having an office at 160 Saratoga Avenue, Santa Clara, CA 95051, and the Individual or Organization ("Licensee") accessing and otherwise using this software in source or binary form and its associated documentation ("the Software").

2. Subject to the terms and conditions of this BeOpen Python License Agreement, BeOpen hereby grants Licensee a non-exclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use the Software alone or in any derivative version, provided, however, that the BeOpen Python License is retained in the Software, alone or in any derivative version prepared by Licensee.

3. BeOpen is making the Software available to Licensee on an "AS IS" basis. BEOPEN MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, BEOPEN MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF THE SOFTWARE WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

4. BEOPEN SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF THE SOFTWARE FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at http://www.pythonlabs.com/logos.html may be used according to the permissions granted on that web page.

7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

CNRI LICENSE AGREEMENT FOR PYTHON 1.6.1

1. This LICENSE AGREEMENT is between the Corporation for National Research Initiatives, having an office at 1895 Preston White Drive, Reston, VA 20191 ("CNRI"), and the Individual or Organization ("Licensee") accessing and otherwise using Python 1.6.1 software in source or binary form and its associated documentation.

2. Subject to the terms and conditions of this License Agreement, CNRI hereby grants Licensee a nonexclusive, royalty-free, world-wide license to reproduce, analyze, test, perform and/or display publicly, prepare derivative works, distribute, and otherwise use Python 1.6.1 alone or in any derivative version, provided, however, that CNRI's License Agreement and CNRI's notice of copyright, i.e., "Copyright © 1995-2001 Corporation for National Research Initiatives; All Rights Reserved" are retained in Python 1.6.1 alone or in any derivative version prepared by Licensee. Alternately, in lieu of CNRI's License Agreement, Licensee may substitute the following text (omitting the quotes): "Python 1.6.1 is made available subject to the terms and conditions in CNRI's License Agreement. This Agreement together with Python 1.6.1 may be located on the Internet using the following unique, persistent identifier (known as a handle): 1895.22/1013. This Agreement may also be obtained from a proxy server on the Internet using the following URL: http://hdl.handle.net/1895.22/1013."

3. In the event Licensee prepares a derivative work that is based on or incorporates Python 1.6.1 or any part thereof, and wants to make the derivative work available to others as provided herein, then Licensee hereby agrees to include in any such work a brief summary of the changes made to Python 1.6.1.

4. CNRI is making Python 1.6.1 available to Licensee on an "AS IS" basis. CNRI MAKES NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED. BY WAY OF EXAMPLE, BUT NOT LIMITATION, CNRI MAKES NO AND DISCLAIMS ANY REPRESENTATION OR WARRANTY OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE OR THAT THE USE OF PYTHON 1.6.1 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.

5. CNRI SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 1.6.1 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.

7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to

create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.

8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

ACCEPT

CWI LICENSE AGREEMENT FOR PYTHON 0.9.0 THROUGH 1.2

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of Stichting Mathematisch Centrum or CWI not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

STICHTING MATHEMATISCH CENTRUM DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN NO EVENT SHALL STICHTING MATHEMATISCH CENTRUM BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

A.3 Licenze e riconoscimenti per i programmi incorporati

This section is an incomplete, but growing list of licenses and acknowledgements for third-party software incorporated in the Python distribution.

A.3.1 Mersenne Twister

The `_random` module includes code based on a download from http://www.math.keio.ac.jp/ matumoto/MT2002/emt19937ar.html. The following are the verbatim comments from the original code:

```
A C-program for MT19937, with initialization improved 2002/1/26.
Coded by Takuji Nishimura and Makoto Matsumoto.

Before using, initialize the state by using init_genrand(seed)
or init_by_array(init_key, key_length).

Copyright (C) 1997 - 2002, Makoto Matsumoto and Takuji Nishimura,
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:

 1. Redistributions of source code must retain the above copyright
    notice, this list of conditions and the following disclaimer.

 2. Redistributions in binary form must reproduce the above copyright
    notice, this list of conditions and the following disclaimer in the
    documentation and/or other materials provided with the distribution.

 3. The names of its contributors may not be used to endorse or promote
    products derived from this software without specific prior written
    permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR
A PARTICULAR PURPOSE ARE DISCLAIMED.  IN NO EVENT SHALL THE COPYRIGHT OWNER OR
CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,
EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO,
PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR
PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF
LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.


Any feedback is very welcome.
http://www.math.keio.ac.jp/matumoto/emt.html
email: matumoto@math.keio.ac.jp
```

A.3.2 Sockets

The `socket` module uses the functions, `getaddrinfo`, and `getnameinfo`, which are coded in separate source files from the WIDE Project, http://www.wide.ad.jp/about/index.html.

```
Copyright (C) 1995, 1996, 1997, and 1998 WIDE Project.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions
are met:
1. Redistributions of source code must retain the above copyright
   notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright
   notice, this list of conditions and the following disclaimer in the
   documentation and/or other materials provided with the distribution.
3. Neither the name of the project nor the names of its contributors
   may be used to endorse or promote products derived from this software
   without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE PROJECT AND CONTRIBUTORS ‘‘AS IS’’ AND
GAI_ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
ARE DISCLAIMED.  IN NO EVENT SHALL THE PROJECT OR CONTRIBUTORS BE LIABLE
FOR GAI_ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
HOWEVER CAUSED AND ON GAI_ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN GAI_ANY WAY
OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
SUCH DAMAGE.
```

A.3.3 Floating point exception control

The source for the `fpectl` module includes the following notice:

A.3.4 MD5 message digest algorithm

The source code for the `md5` module contains the following notice:

```
Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All
rights reserved.

License to copy and use this software is granted provided that it
is identified as the "RSA Data Security, Inc. MD5 Message-Digest
Algorithm" in all material mentioning or referencing this software
or this function.

License is also granted to make and use derivative works provided
that such works are identified as "derived from the RSA Data
Security, Inc. MD5 Message-Digest Algorithm" in all material
mentioning or referencing the derived work.

RSA Data Security, Inc. makes no representations concerning either
the merchantability of this software or the suitability of this
software for any particular purpose. It is provided "as is"
without express or implied warranty of any kind.

These notices must be retained in any copies of any part of this
documentation and/or software.
```

A.3.5 rotor – Enigma-like encryption and decryption

The source code for the `rotor` contains the following notice:

```
Copyright 1994 by Lance Ellinghouse,
Cathedral City, California Republic, United States of America.

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.

LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE BE LIABLE FOR ANY SPECIAL,
INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING
FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION
WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

A.3.6 Asynchronous socket services

The `asynchat` and `asyncore` modules contain the following notice:

```
Copyright 1996 by Sam Rushing

                        All Rights Reserved

Permission to use, copy, modify, and distribute this software and
its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of Sam
Rushing not be used in advertising or publicity pertaining to
distribution of the software without specific, written prior
permission.

SAM RUSHING DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE,
INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS, IN
NO EVENT SHALL SAM RUSHING BE LIABLE FOR ANY SPECIAL, INDIRECT OR
CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS
OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT,
NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

A.3.7 Cookie management

The `Cookie` module contains the following notice:

```
Copyright 2000 by Timothy O'Malley <timo@alum.mit.edu>

               All Rights Reserved

Permission to use, copy, modify, and distribute this software
and its documentation for any purpose and without fee is hereby
granted, provided that the above copyright notice appear in all
copies and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Timothy O'Malley  not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

Timothy O'Malley DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY
AND FITNESS, IN NO EVENT SHALL Timothy O'Malley BE LIABLE FOR
ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR
PERFORMANCE OF THIS SOFTWARE.
```

A.3.8 Profiling

The `profile` and `pstats` modules contain the following notice:

```
Copyright 1994, by InfoSeek Corporation, all rights reserved.
Written by James Roskind

Permission to use, copy, modify, and distribute this Python software
and its associated documentation for any purpose (subject to the
restriction in the following sentence) without fee is hereby granted,
provided that the above copyright notice appears in all copies, and
that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of InfoSeek not be used in
advertising or publicity pertaining to distribution of the software
without specific, written prior permission.  This permission is
explicitly restricted to the copying and modification of the software
to remain in Python, compiled Python, or other languages (such as C)
wherein the modified or derived code is exclusively imported into a
Python module.

INFOSEEK CORPORATION DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS
SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS. IN NO EVENT SHALL INFOSEEK CORPORATION BE LIABLE FOR ANY
SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER
RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF
CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN
CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
```

A.3.9 Execution tracing

The `trace` module contains the following notice:

```
portions copyright 2001, Autonomous Zones Industries, Inc., all rights...
err...  reserved and offered to the public under the terms of the
Python 2.2 license.
Author: Zooko O'Whielacronx
http://zooko.com/
mailto:zooko@zooko.com

Copyright 2000, Mojam Media, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1999, Bioreason, Inc., all rights reserved.
Author: Andrew Dalke

Copyright 1995-1997, Automatrix, Inc., all rights reserved.
Author: Skip Montanaro

Copyright 1991-1995, Stichting Mathematisch Centrum, all rights reserved.


Permission to use, copy, modify, and distribute this Python software and
its associated documentation for any purpose without fee is hereby
granted, provided that the above copyright notice appears in all copies,
and that both that copyright notice and this permission notice appear in
supporting documentation, and that the name of neither Automatrix,
Bioreason or Mojam Media be used in advertising or publicity pertaining to
distribution of the software without specific, written prior permission.
```

A.3.10 UUencode and UUdecode functions

The uu module contains the following notice:

```
Copyright 1994 by Lance Ellinghouse
Cathedral City, California Republic, United States of America.
                       All Rights Reserved
Permission to use, copy, modify, and distribute this software and its
documentation for any purpose and without fee is hereby granted,
provided that the above copyright notice appear in all copies and that
both that copyright notice and this permission notice appear in
supporting documentation, and that the name of Lance Ellinghouse
not be used in advertising or publicity pertaining to distribution
of the software without specific, written prior permission.
LANCE ELLINGHOUSE DISCLAIMS ALL WARRANTIES WITH REGARD TO
THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND
FITNESS, IN NO EVENT SHALL LANCE ELLINGHOUSE CENTRUM BE LIABLE
FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES
WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN
ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT
OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.

Modified by Jack Jansen, CWI, July 1995:
- Use binascii module to do the actual line-by-line conversion
  between ascii and binary. This results in a 1000-fold speedup. The C
  version is still 5 times faster, though.
- Arguments more compliant with python standard
```

A.3.11 XML Remote Procedure Calls

The xmlrpclib module contains the following notice:

```
    The XML-RPC client interface is

Copyright (c) 1999-2002 by Secret Labs AB
Copyright (c) 1999-2002 by Fredrik Lundh

By obtaining, using, and/or copying this software and/or its
associated documentation, you agree that you have read, understood,
and will comply with the following terms and conditions:

Permission to use, copy, modify, and distribute this software and
its associated documentation for any purpose and without fee is
hereby granted, provided that the above copyright notice appears in
all copies, and that both that copyright notice and this permission
notice appear in supporting documentation, and that the name of
Secret Labs AB or the author not be used in advertising or publicity
pertaining to distribution of the software without specific, written
prior permission.

SECRET LABS AB AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD
TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANT-
ABILITY AND FITNESS.  IN NO EVENT SHALL SECRET LABS AB OR THE AUTHOR
BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY
DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS,
WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS
ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE
OF THIS SOFTWARE.
```

INDICE ANALITICO

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z