
Python Tutorial

Relis 2.7.18

**Guido van Rossum
and the Python development team**

Mei 20, 2020

**Python Software Foundation
Email: docs@python.org**

1	Membangkitkan Selera Anda	3
2	Menggunakan Interpreter Python	5
2.1	Memanggil Interpreter	5
2.2	Interpreter dan Lingkungannya	7
3	Pengantar Informal Tentang Python	9
3.1	Menggunakan Python sebagai Kalkulator	10
3.2	Langkah Awal Menuju Pemrograman	18
4	Lebih Banyak Alat Pengatur Aliran <i>Control Flow</i>	21
4.1	<code>if</code> Statements	21
4.2	<code>for</code> Statements	22
4.3	Fungsi <code>range()</code>	22
4.4	<code>break</code> and <code>continue</code> Statements, and <code>else</code> Clauses on Loops	23
4.5	<code>pass</code> Statements	24
4.6	Mendefinisikan Fungsi	24
4.7	Lebih lanjut tentang Mendefinisikan Fungsi	26
4.8	Intermezzo: <i>Gaya Coding</i>	30
5	Struktur Data	31
5.1	Lebih Lanjut tentang Daftar <i>Lists</i>	31
5.2	The <code>del</code> statement	36
5.3	Tuples and <i>Urutan Sequences</i>	37
5.4	Himpunan <i>Set</i>	38
5.5	Kamus <i>Dictionaries</i>	39
5.6	Teknik Perulangan	40
5.7	Lebih lanjut tentang Kondisi	41
5.8	Membandingkan <i>Urutan Sequences</i> dan Jenis Lainnya	42
6	Modul-Modul	43
6.1	Lebih lanjut tentang Modul	44
6.2	Modul Standar	47
6.3	Fungsi <code>dir()</code>	47
6.4	Paket	48
7	Masukan dan Keluaran	53

7.1	Pemformatan Keluaran yang Lebih Menarik	53
7.2	Membaca dan Menulis Berkas	56
8	Kesalahan <i>errors</i> dan Pengecualian <i>exceptions</i>	61
8.1	Kesalahan Sintaksis	61
8.2	Pengecualian	61
8.3	Menangani Pengecualian	62
8.4	Memunculkan Pengecualian	64
8.5	Pengecualian yang Ditentukan Pengguna	65
8.6	Mendefinisikan Tindakan Pembersihan	66
8.7	Tindakan Pembersihan yang Sudah Ditentukan	67
9	Classes	69
9.1	Sepatah Kata Tentang Nama dan Objek	69
9.2	Lingkup Python dan <i>Namespaces</i>	70
9.3	Pandangan Pertama tentang Kelas	71
9.4	Keterangan Acak	75
9.5	Pewarisan	76
9.6	Private Variables and Class-local References	78
9.7	Barang Sisa <i>Odds and Ends</i>	79
9.8	Exceptions Are Classes Too	79
9.9	<i>Iterators</i>	80
9.10	Pembangkit <i>Generator</i>	81
9.11	Ekspresi Pembangkit <i>Generator</i>	82
10	Tur Singkat Pustaka Standar	83
10.1	Antarmuka Sistem Operasi	83
10.2	Berkas <i>Wildcard</i>	84
10.3	Baris Perintah Berargumen	84
10.4	Pengalihan Output Kesalahan dan Pengakhiran Program	84
10.5	Pencocokan Pola String	84
10.6	Matematika	85
10.7	Akses internet	85
10.8	Tanggal dan Waktu	86
10.9	Kompresi Data	86
10.10	Pengukuran Kinerja	86
10.11	Kontrol kualitas	87
10.12	Dilengkapi Baterai	87
11	Tur Singkat Pustaka Standar --- Bagian II	89
11.1	Pemformatan Output	89
11.2	<i>Templating</i>	90
11.3	Bekerja dengan Tata Letak Rekam Data Biner	91
11.4	<i>Multi-threading</i>	92
11.5	Pencatatan	92
11.6	Referensi yang Lemah	93
11.7	Alat untuk Bekerja dengan Daftar <i>Lists</i>	94
11.8	Aritmatika Pecahan <i>Floating Point</i> Desimal	95
12	Lalu apa sekarang?	97
13	Pengeditan Input Interaktif dan Penggantian Riwayat	99
13.1	Line Editing	99
13.2	History Substitution	99
13.3	Key Bindings	100

13.4	Alternatif untuk Interpreter Interaktif	101
14	Aritmatika Pecahan <i>Floating Point</i>: Masalah dan Keterbatasan	103
14.1	Kesalahan Representasi	105
15	Lampiran	107
15.1	Mode Interaktif	107
A	Ikhtisar	109
B	Tentang dokumen-dokumen ini	119
B.1	Kontributor untuk dokumentasi Python	119
C	Sejarah dan Lisensi	121
C.1	Sejarah perangkat lunak	121
C.2	Syarat dan ketentuan untuk mengakses atau menggunakan Python	122
C.3	Lisensi dan Ucapan Terima Kasih untuk Perangkat Lunak yang Tergabung	125
D	Hak Cipta	137
	Indeks	139

Python adalah bahasa pemrograman yang berdaya dan mudah dipelajari. Python memiliki struktur data tingkat tinggi yang efisien dan pendekatan yang sederhana namun efektif untuk pemrograman berorientasi objek. Sintaksis Python yang elegan dan tipe dinamis, bersama dengan sifatnya yang diinterpretasikan, menjadikannya bahasa yang ideal untuk skrip dan pengembangan aplikasi yang cepat di banyak area di sebagian besar platform.

Interpreter Python dan pustaka standar yang luas tersedia secara bebas dalam bentuk kode sumber atau biner untuk semua platform utama dari situs Web Python, <https://www.python.org/>, dan dapat didistribusikan secara bebas. Situs yang sama juga berisi distribusi dan referensi ke banyak modul Python gratis dari pihak ketiga, program dan alat, serta dokumentasi tambahan.

Interpreter Python mudah dikembangkan dengan fungsi dan tipe data baru diimplementasikan dalam C atau C ++ (atau bahasa lain yang bisa dipanggil dari C). Python juga cocok sebagai bahasa tambahan untuk aplikasi yang dapat disesuaikan.

Tutorial ini memperkenalkan pembaca secara informal ke konsep dan fitur dasar bahasa dan sistem Python. Akan membantu untuk memiliki interpreter Python yang praktis untuk pengalaman mencoba langsung, tetapi semua contoh mandiri, sehingga tutorialnya dapat dibaca secara off-line juga.

Untuk deskripsi objek dan modul standar, lihat `library-index`. `reference-index` memberikan definisi bahasa yang lebih formal. Untuk menulis ekstensi dalam C atau C ++, baca `extending-index` dan `c-api-index`. Ada juga beberapa buku yang membahas Python secara mendalam.

Tutorial ini tidak mencoba menjadi komprehensif dan mencakup semua fitur, atau bahkan setiap fitur yang umum digunakan. Alih-alih, ini memperkenalkan banyak fitur Python yang paling penting, dan akan memberi Anda ide bagus tentang rasa dan gaya bahasa itu. Setelah membacanya, Anda akan dapat membaca dan menulis modul serta program Python, dan Anda akan siap untuk mempelajari lebih lanjut tentang berbagai modul pustaka Python yang dijelaskan dalam `library-index`.

glosarium juga layak untuk dilihat.

Membangkitkan Selera Anda

Jika Anda melakukan banyak pekerjaan pada komputer, pada akhirnya Anda menemukan bahwa ada beberapa tugas yang ingin Anda lakukan secara otomatis. Misalnya, Anda mungkin ingin melakukan pencarian-dan-ganti dari sejumlah besar berkas teks, atau ganti nama dan atur ulang banyak file foto dengan cara yang rumit. Mungkin Anda ingin menulis basis data khusus kecil, atau aplikasi GUI khusus, atau permainan sederhana.

Jika Anda seorang pengembang perangkat lunak profesional, Anda mungkin harus bekerja dengan beberapa pustaka C/C++/Java tetapi menemukan siklus penulisan/kompilasi/pengujian/kompilasi ulang yang biasa terlalu lambat. Mungkin Anda sedang menulis serangkaian pengujian untuk pustaka seperti itu dan menemukan menulis kode pengujian tugas yang membosankan. Atau mungkin Anda telah menulis sebuah program yang dapat menggunakan bahasa ekstensi, dan Anda tidak ingin merancang dan mengimplementasikan bahasa yang sama sekali baru untuk aplikasi Anda.

Python adalah bahasa yang sesuai untuk Anda.

Anda bisa menulis skrip Unix *shell* atau berkas *batch* Windows untuk beberapa tugas ini, tetapi skrip *shell* paling baik untuk bergerak di sekitar berkas dan mengubah data teks, tidak cocok untuk aplikasi atau game GUI. Anda bisa menulis program C/C++/Java, tetapi mungkin butuh banyak waktu pengembangan untuk mendapatkan bahkan program draft pertama. Python lebih mudah digunakan, tersedia di sistem operasi Windows, Mac OS X, dan Unix, dan akan membantu Anda menyelesaikan pekerjaan dengan lebih cepat.

Python mudah digunakan, tetapi ini adalah bahasa pemrograman nyata, menawarkan lebih banyak struktur dan dukungan untuk program besar daripada skrip *shell* atau berkas *batch* dapat tawarkan. Di sisi lain, Python juga menawarkan pemeriksaan kesalahan jauh lebih banyak daripada C, dan, karena *bahasa tingkat sangat tinggi*, ia memiliki tipe data tingkat tinggi yang tertanam di dalamnya, seperti *arrays* dan *dictionary* yang fleksibel. Karena tipe datanya yang lebih umum, Python dapat diterapkan pada domain masalah yang jauh lebih besar daripada Awk atau bahkan Perl, namun banyak hal yang setidaknya semudah dalam Python seperti pada bahasa-bahasa tersebut.

Python memungkinkan Anda untuk membagi program Anda menjadi modul yang dapat digunakan kembali dalam program Python lainnya. Muncul dengan koleksi besar modul standar yang dapat Anda gunakan sebagai dasar program Anda --- atau sebagai contoh untuk mulai belajar memprogram dengan Python. Beberapa modul ini menyediakan hal-hal seperti berkas I/O, panggilan sistem, soket, dan bahkan antarmuka ke *toolkit* antarmuka pengguna grafis seperti Tk.

Python adalah bahasa yang difafsirkan, yang dapat menghemat waktu Anda selama pengembangan program karena tidak diperlukan kompilasi dan penautan. *interpreter* dapat digunakan secara interaktif, yang membuatnya mudah untuk bereksperimen dengan fitur-fitur bahasa, untuk menulis *throw-away programs*, atau untuk menguji fungsi selama pengembangan program *bottom-up*. Ini juga merupakan kalkulator meja yang berguna.

Python memungkinkan program ditulis secara ringkas dan mudah dibaca. Program yang ditulis dengan Python biasanya jauh lebih pendek daripada program C, C++, atau Java yang setara, karena beberapa alasan:

- tipe data tingkat tinggi memungkinkan Anda untuk mengekspresikan operasi yang kompleks dalam satu pernyataan;
- pengelompokan pernyataan dilakukan dengan indentasi alih-alih tanda kurung kurawal di awal dan akhir;
- tidak ada deklarasi variabel atau argumen yang diperlukan.

Python bersifat *extensible*: jika Anda tahu cara memprogram dalam C, mudah untuk menambahkan fungsi atau modul bawaan baru ke *interpreter*, baik untuk melakukan operasi kritis dengan kecepatan maksimum, atau untuk menautkan program Python ke perpustakaan yang mungkin hanya tersedia dalam bentuk biner (seperti pustaka grafik spesifik vendor). Setelah Anda benar-benar ketagihan, Anda dapat menautkan juru bahasa Python ke dalam aplikasi yang ditulis dalam C dan menggunakannya sebagai ekstensi atau bahasa perintah untuk aplikasi itu.

Ngomong-ngomong, bahasa tersebut dinamai menurut acara BBC "Sirkus Terbang Monty Python" dan tidak ada hubungannya dengan reptil. Membuat referensi ke sandiwara Monty Python dalam dokumentasi tidak hanya diizinkan, tetapi juga dianjurkan!

Sekarang Anda semua bersemangat tentang Python, Anda akan ingin memeriksanya lebih terinci. Karena cara terbaik untuk belajar bahasa adalah menggunakannya, tutorial mengundang Anda untuk bermain dengan *interpreter* Python saat Anda membaca.

Dalam bab selanjutnya, mekanisme penggunaan *interpreter* dijelaskan. Ini adalah informasi yang biasa saja, tetapi penting untuk mencoba contoh yang ditunjukkan nanti.

Sisa tutorial ini memperkenalkan berbagai fitur bahasa dan sistem Python melalui contoh, dimulai dengan ekspresi sederhana, pernyataan dan tipe data, melalui fungsi dan modul, dan akhirnya menyentuh konsep-konsep lanjutan seperti pengecualian dan kelas yang ditentukan pengguna.

Menggunakan Interpreter Python

2.1 Memanggil Interpreter

The Python interpreter is usually installed as `/usr/local/bin/python` on those machines where it is available; putting `/usr/local/bin` in your Unix shell's search path makes it possible to start it by typing the command

```
python
```

to the shell. Since the choice of the directory where the interpreter lives is an installation option, other places are possible; check with your local Python guru or system administrator. (E.g., `/usr/local/python` is a popular alternative location.)

On Windows machines, the Python installation is usually placed in `C:\Python27`, though you can change this when you're running the installer. To add this directory to your path, you can type the following command into the command prompt in a DOS box:

```
set path=%path%;C:\python27
```

Mengetik karakter akhir file (Control-D pada Unix, :kbd:`Control-Z` pada Windows) pada prompt utama menyebabkan interpreter keluar dengan status keluar nol. Jika itu tidak berhasil, Anda dapat keluar dari interpreter dengan mengetikkan perintah berikut: `quit()`.

The interpreter's line-editing features usually aren't very sophisticated. On Unix, whoever installed the interpreter may have enabled support for the GNU readline library, which adds more elaborate interactive editing and history features. Perhaps the quickest check to see whether command line editing is supported is typing Control-P to the first Python prompt you get. If it beeps, you have command line editing; see Appendix *Pengeditan Input Interaktif dan Penggantian Riwayat* for an introduction to the keys. If nothing appears to happen, or if ^P is echoed, command line editing isn't available; you'll only be able to use backspace to remove characters from the current line.

Interpreter beroperasi mirip seperti shell Unix: ketika dipanggil dengan masukan bawaan yang terhubung ke perangkat tty, ia membaca dan mengeksekusi perintah secara interaktif; ketika dipanggil dengan argumen nama berkas atau dengan berkas sebagai masukan bawaan, ia membaca dan mengeksekusi *script* dari berkas itu.

Cara kedua untuk memulai interpreter adalah `python -c command [arg] ...`, yang mengeksekusi pernyataan(-pernyataan) dalam *command*, dianalogikan dengan opsi shell `-c`. Karena pernyataan Python sering mengandung spasi

atau karakter lain yang khusus untuk shell, biasanya disarankan untuk mengutip *command* secara keseluruhan dengan tanda kutip tunggal.

Beberapa modul Python juga berguna sebagai skrip. Ini dapat dipanggil menggunakan `python -m module [arg] . . .`, yang mengeksekusi berkas sumber untuk *module* seolah-olah Anda telah menuliskan nama lengkapnya pada baris perintah.

Ketika berkas skrip digunakan, terkadang berguna untuk dapat menjalankan skrip dan masuk ke mode interaktif sesudahnya. Ini dapat dilakukan dengan menuliskan `-i` sebelum skrip.

All command-line options are described in *using-on-general*.

2.1.1 Melewatkan Argumen

Ketika diketahui oleh interpreter, nama skrip dan argumen tambahan sesudahnya diubah menjadi daftar string dan diberikan nilai ke variabel `argv` dalam modul `sys`. Anda dapat mengakses daftar ini dengan menjalankan `import sys`. Panjang daftar setidaknya satu; ketika tidak ada skrip dan tidak ada argumen yang diberikan, `sys.argv[0]` adalah string kosong. Ketika nama skrip diberikan sebagai `'-'` (artinya standar masukan), `sys.argv[0]` diatur ke `'-'`. Ketika *command -c* digunakan, `sys.argv[0]` diatur ke `"-c"`. Ketika **module* :option: -m* digunakan, `sys.argv[0]` diatur ke nama lengkap modul yang digunakan. Opsi ditemukan setelah *command -c* atau *module -m* tidak dikonsumsi oleh pemrosesan opsi interpreter Python tetapi ditinggalkan di `sys.argv` untuk perintah atau modul yang akan ditangani.

2.1.2 Mode Interaktif

Ketika perintah dibaca dari tty, interpreter dikatakan dalam *interactive mode*. Dalam mode ini interpreter meminta perintah berikutnya dengan *primary prompt*, biasanya tiga tanda lebih besar dari (`>>>`); untuk garis lanjutan, interpreter meminta dengan *secondary prompt*, secara bawaan tiga titik (`...`). Interpreter mencetak pesan selamat datang yang menyatakan nomor versinya dan pemberitahuan hak cipta sebelum mencetak prompt pertama:

```
python
Python 2.7 (#1, Feb 28 2010, 00:02:06)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Garis lanjutan diperlukan ketika memasuki konstruksi multi-garis. Sebagai contoh, lihat ini pernyataan `if`:

```
>>> the_world_is_flat = 1
>>> if the_world_is_flat:
...     print "Be careful not to fall off!"
...
Be careful not to fall off!
```

Untuk lebih lanjut tentang mode interaktif, lihat [Mode Interaktif](#).

2.2 Interpreter dan Lingkungannya

2.2.1 *Encoding* Penulisan Kode Sumber

By default, Python source files are treated as encoded in ASCII. To declare an encoding other than the default one, a special comment line should be added as the *first* line of the file. The syntax is as follows:

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

di mana *encoding* adalah salah satu yang valid `codec` yang didukung oleh Python.

Misalnya, untuk menyatakan bahwa pengkodean *encoding* Windows-1252 harus digunakan, baris pertama file kode sumber Anda harus:

```
# -*- coding: cp1252 -*-
```

Satu pengecualian untuk aturan *baris pertama* adalah ketika kode sumber dimulai dengan *UNIX "shebang" line*. Dalam hal ini, deklarasi penandian *encoding* harus ditambahkan sebagai baris kedua pada berkas. Sebagai contoh:

```
#!/usr/bin/env python  
# -*- coding: cp1252 -*-
```

Pengantar Informal Tentang Python

Dalam contoh berikut, masukan dan keluaran dibedakan dengan ada atau tidaknya prompt (`>` dan `...`): untuk mengulangi contoh, Anda harus mengetikkan semuanya setelah prompt, saat prompt muncul; baris yang tidak dimulai dengan prompt adalah output dari interpreter. Perhatikan bahwa baris yang hanya berisi prompt sekunder dalam contoh berarti Anda harus mengetikkan baris kosong; ini digunakan untuk mengakhiri perintah multi-baris.

Banyak contoh dalam manual ini, bahkan yang dimasukkan pada prompt interaktif, termasuk komentar. Komentar dalam Python dimulai dengan karakter hash, `#`, dan diperluas hingga akhir garis fisik. Sebuah komentar dapat muncul di awal baris atau mengikuti spasi atau kode, tetapi tidak dalam string literal. Karakter hash dalam string literal hanyalah karakter hash. Karena komentar adalah untuk mengklarifikasi kode dan tidak ditafsirkan oleh Python, mereka dapat dihilangkan saat mengetikkan contoh.

Beberapa contoh:

```
# this is the first comment
spam = 1  # and this is the second comment
          # ... and now a third!
text = "# This is not a comment because it's inside quotes."
```

3.1 Menggunakan Python sebagai Kalkulator

Mari kita coba beberapa perintah Python sederhana. Mulai interpreter dan tunggu prompt utama, `>>>`. (Seharusnya tidak butuh waktu lama.)

3.1.1 Angka

Interpreter bertindak sebagai kalkulator sederhana: Anda dapat mengetikkan ekspresi padanya dan itu akan menulis nilainya. Sintaksis ekspresi mudah: operator `+`, `-`, `*` dan `/` berfungsi seperti di sebagian besar bahasa lain (misalnya, Pascal atau C); tanda kurung `()` dapat digunakan untuk pengelompokan. Sebagai contoh:

```
>>> 2 + 2
4
>>> 50 - 5*6
20
>>> (50 - 5.0*6) / 4
5.0
>>> 8 / 5.0
1.6
```

Bilangan bulat (mis. 2, 4, 20) memiliki tipe `int`, yang memiliki bagian pecahan (mis. “5.0”, “1.6”) memiliki tipe `float`. Kita akan melihat lebih banyak tentang tipe bilangan nanti dalam tutorial.

The return type of a division (`/`) operation depends on its operands. If both operands are of type `int`, *floor division* is performed and an `int` is returned. If either operand is a `float`, classic division is performed and a `float` is returned. The `//` operator is also provided for doing floor division no matter what the operands are. The remainder can be calculated with the `%` operator:

```
>>> 17 / 3 # int / int -> int
5
>>> 17 / 3.0 # int / float -> float
5.666666666666667
>>> 17 // 3.0 # explicit floor division discards the fractional part
5.0
>>> 17 % 3 # the % operator returns the remainder of the division
2
>>> 5 * 3 + 2 # result * divisor + remainder
17
```

Dengan Python, dimungkinkan untuk menggunakan operator `**` untuk menghitung pemangkatan¹:

```
>>> 5 ** 2 # 5 squared
25
>>> 2 ** 7 # 2 to the power of 7
128
```

Tanda sama dengan (`=`) digunakan untuk memberikan nilai ke variabel. Setelah itu, tidak ada hasil yang ditampilkan sebelum prompt interaktif berikutnya:

```
>>> width = 20
>>> height = 5 * 9
>>> width * height
900
```

¹ Karena `**` memiliki prioritas lebih tinggi dari `-`, `-3**2` akan ditafsirkan sebagai `-(3**2)` dan karenanya menghasilkan `-9`. Untuk menghindari ini dan mendapatkan 9, Anda dapat menggunakan `(-3)**2`.

Jika variabel tidak "didefinisikan" (diberi nilai), mencoba menggunakannya akan menghasilkan kesalahan:

```
>>> n # try to access an undefined variable
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'n' is not defined
```

Ada dukungan penuh untuk floating point; operator dengan operan tipe campuran akan mengubah operan integer ke floating point:

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

Dalam mode interaktif, ekspresi cetak terakhir diberikan ke variabel `_`. Ini berarti bahwa ketika Anda menggunakan Python sebagai kalkulator meja, agak lebih mudah untuk melanjutkan perhitungan, misalnya:

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

Variabel ini harus diperlakukan sebagai baca-saja oleh pengguna. Jangan secara eksplisit memberikan nilai padanya - -- Anda akan membuat variabel lokal independen dengan nama yang sama menutupi variabel bawaan dengan perilaku saktinya.

Selain `int` dan `float`, Python mendukung tipe angka lainnya, seperti `Decimal` dan `Fraction`. Python juga memiliki dukungan bawaan untuk complex numbers, dan menggunakan akhiran `j` atau `J` untuk menunjukkan bagian imajiner (mis. `3+5j`).

3.1.2 String

Selain angka, Python juga dapat memanipulasi string atau teks, yang dapat diekspresikan dalam beberapa cara. Mereka dapat disertakan dalam tanda kutip tunggal (`'...'`) atau tanda kutip ganda (`"..."`) dengan hasil yang sama². `\` dapat digunakan untuk keluar dari kutipan:

```
>>> 'spam eggs' # single quotes
'spam eggs'
>>> 'doesn\'t' # use \' to escape the single quote...
"doesn't"
>>> "doesn't" # ...or use double quotes instead
"doesn't"
>>> '"Yes," they said.'
'"Yes," they said.'
>>> "\"Yes,\" they said."
'"Yes," they said.'
>>> 'Isn\'t," they said.'
'Isn\'t," they said.'
```

² Tidak seperti bahasa lain, karakter khusus seperti `\n` memiliki arti yang sama dengan kedua tanda kutip tunggal (`'...'`) dan ganda (`"..."`). Satu-satunya perbedaan antara keduanya adalah bahwa dalam tanda kutip tunggal Anda tidak perlu memisahkan `"` (tetapi Anda harus memisahkan `\`) dan sebaliknya.

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The `print` statement produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
>>> "Isn't," they said.
'Isn't," they said.'
>>> print "Isn't," they said.
Isn't," they said.
>>> s = 'First line.\nSecond line.' # \n means newline
>>> s # without print, \n is included in the output
'First line.\nSecond line.'
>>> print s # with print, \n produces a new line
First line.
Second line.
```

Jika Anda tidak ingin karakter yang diawali dengan `\` ditafsirkan sebagai karakter khusus, Anda dapat menggunakan *raw strings* dengan menambahkan `r` sebelum kutipan pertama:

```
>>> print 'C:\some\name' # here \n means newline!
C:\some
ame
>>> print r'C:\some\name' # note the r before the quote
C:\some\name
```

String literal dapat melebar hingga beberapa baris. Salah satu caranya adalah dengan menggunakan tanda kutip tiga: `"""`. `..."""` atau `'''...'''`. Akhir baris secara otomatis termasuk dalam string, tetapi dimungkinkan untuk mencegahnya dengan menambahkan `\` di akhir baris. Contoh berikut:

```
print """\
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
"""
```

menghasilkan keluaran berikut (perhatikan bahwa awal baris baru tidak termasuk):

```
Usage: thingy [OPTIONS]
    -h                Display this usage message
    -H hostname       Hostname to connect to
```

String dapat digabungkan (direkatkan) dengan operator `+`, dan diulangi dengan `*`:

```
>>> # 3 times 'un', followed by 'ium'
>>> 3 * 'un' + 'ium'
'unununium'
```

Dua atau lebih *string literals* (yaitu yang terlampir di antara tanda kutip) di sebelah satu sama lain secara otomatis digabungkan.

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

Fitur ini sangat berguna ketika Anda ingin memecah string panjang:

```
>>> text = ('Put several strings within parentheses '
...        'to have them joined together.')
>>> text
'Put several strings within parentheses to have them joined together.'
```

Ini hanya bekerja dengan dua literal, tidak dengan variabel atau ekspresi:

```
>>> prefix = 'Py'
>>> prefix 'thon' # can't concatenate a variable and a string literal
...
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
...
SyntaxError: invalid syntax
```

Jika Anda ingin menggabungkan variabel atau variabel dan literal, gunakan +:

```
>>> prefix + 'thon'
'Python'
```

String dapat diindeks atau *indexed* (disandikan), dengan karakter pertama memiliki indeks 0. Tidak ada tipe karakter yang terpisah; sebuah karakter hanyalah sebuah string berukuran satu:

```
>>> word = 'Python'
>>> word[0] # character in position 0
'P'
>>> word[5] # character in position 5
'n'
```

Indeks juga bisa berupa angka negatif, untuk mulai menghitung dari kanan:

```
>>> word[-1] # last character
'n'
>>> word[-2] # second-last character
'o'
>>> word[-6]
'P'
```

Perhatikan bahwa karena -0 sama dengan 0, indeks negatif mulai dari -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain a substring:

```
>>> word[0:2] # characters from position 0 (included) to 2 (excluded)
'Py'
>>> word[2:5] # characters from position 2 (included) to 5 (excluded)
'tho'
```

Perhatikan bagaimana awal selalu disertakan, dan akhirnya selalu dikecualikan. Ini memastikan bahwa `s[:i] + s[i:]` selalu sama dengan `s`:

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

Indeks irisan memiliki nilai bawaan yang berguna; indeks pertama yang hilang akan digantikan ke nilai nol, indeks kedua yang hilang akan digantikan ke nilai ukuran atau panjang string yang diiris.

```
>>> word[:2]    # character from the beginning to position 2 (excluded)
'Py'
>>> word[4:]    # characters from position 4 (included) to the end
'on'
>>> word[-2:]   # characters from the second-last (included) to the end
'on'
```

Salah satu cara untuk mengingat bagaimana irisan bekerja adalah dengan menganggap indeks sebagai menunjuk *between* karakter, dengan tepi kiri karakter pertama bernomor 0. Kemudian tepi kanan karakter terakhir dari string n karakter memiliki indeks n , misalnya:

```
+---+---+---+---+---+---+
| P | y | t | h | o | n |
+---+---+---+---+---+---+
 0   1   2   3   4   5   6
-6  -5  -4  -3  -2  -1
```

Baris pertama angka memberikan posisi indeks 0...6 dalam string; baris kedua memberikan indeks negatif yang sesuai. Irisan dari i ke j terdiri dari semua karakter di antara kedua sisi yang berlabel i dan j .

Untuk indeks non-negatif, panjang irisan adalah selisih indeks, jika keduanya berada dalam batas. Misalnya, panjang `word[1:3]` adalah 2.

Mencoba menggunakan indeks yang terlalu besar akan menghasilkan kesalahan:

```
>>> word[42]    # the word only has 6 characters
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Namun, indeks irisan di luar jangkauan ditangani dengan anggun ketika digunakan untuk mengiris:

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

String python tidak dapat diubah --- mereka adalah *immutable*. Oleh karena itu, menetapkan ke suatu indeks posisi dalam string menghasilkan kesalahan:

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

Jika Anda membutuhkan string yang berbeda, Anda harus membuat yang baru:

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

Fungsi bawaan `len()` mengembalikan panjang string:

```
>>> s = 'supercalifragilisticexpialidocious'
>>> len(s)
34
```

Lihat juga:

typeseq Strings, and the Unicode strings described in the next section, are examples of *sequence types*, and support the common operations supported by such types.

string-methods Both strings and Unicode strings support a large number of methods for basic transformations and searching.

formatstrings Informasi tentang pemformatan string dengan `str.format()`.

string-formatting The old formatting operations invoked when strings and Unicode strings are the left operand of the `%` operator are described in more detail here.

3.1.3 Unicode Strings

Starting with Python 2.0 a new data type for storing text data is available to the programmer: the Unicode object. It can be used to store and manipulate Unicode data (see <http://www.unicode.org/>) and integrates well with the existing string objects, providing auto-conversions where necessary.

Unicode has the advantage of providing one ordinal for every character in every script used in modern and ancient texts. Previously, there were only 256 possible ordinals for script characters. Texts were typically bound to a code page which mapped the ordinals to script characters. This lead to very much confusion especially with respect to internationalization (usually written as `i18n` --- 'i' + 18 characters + 'n') of software. Unicode solves these problems by defining one code page for all scripts.

Creating Unicode strings in Python is just as simple as creating normal strings:

```
>>> u'Hello World !'
u'Hello World !'
```

The small 'u' in front of the quote indicates that a Unicode string is supposed to be created. If you want to include special characters in the string, you can do so by using the Python *Unicode-Escape* encoding. The following example shows how:

```
>>> u'Hello\u0020World !'
u'Hello World !'
```

The escape sequence `\u0020` indicates to insert the Unicode character with the ordinal value `0x0020` (the space character) at the given position.

Other characters are interpreted by using their respective ordinal values directly as Unicode ordinals. If you have literal strings in the standard Latin-1 encoding that is used in many Western countries, you will find it convenient that the lower 256 characters of Unicode are the same as the 256 characters of Latin-1.

For experts, there is also a raw mode just like the one for normal strings. You have to prefix the opening quote with 'ur' to have Python use the *Raw-Unicode-Escape* encoding. It will only apply the above `\uXXXX` conversion if there is an uneven number of backslashes in front of the small 'u'.

```
>>> ur'Hello\u0020World !'
u'Hello World !'
>>> ur'Hello\\u0020World !'
u'Hello\\u0020World !'
```

The raw mode is most useful when you have to enter lots of backslashes, as can be necessary in regular expressions.

Apart from these standard encodings, Python provides a whole set of other ways of creating Unicode strings on the basis of a known encoding.

The built-in function `unicode()` provides access to all registered Unicode codecs (COders and DEcoders). Some of the more well known encodings which these codecs can convert are *Latin-1*, *ASCII*, *UTF-8*, and *UTF-16*. The latter two are variable-length encodings that store each Unicode character in one or more bytes. The default encoding is normally set to *ASCII*, which passes through characters in the range 0 to 127 and rejects any other characters with an error. When a Unicode string is printed, written to a file, or converted with `str()`, conversion takes place using this default encoding.

```
>>> u"abc"
u'abc'
>>> str(u"abc")
'abc'
>>> u"äöü"
u'\xe4\xfc'
>>> str(u"äöü")
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
UnicodeEncodeError: 'ascii' codec can't encode characters in position 0-2: ordinal
↪ not in range(128)
```

To convert a Unicode string into an 8-bit string using a specific encoding, Unicode objects provide an `encode()` method that takes one argument, the name of the encoding. Lowercase names for encodings are preferred.

```
>>> u"äöü".encode('utf-8')
'\xc3\xa4\xc3\xb6\xc3\xbc'
```

If you have data in a specific encoding and want to produce a corresponding Unicode string from it, you can use the `unicode()` function with the encoding name as the second argument.

```
>>> unicode('\xc3\xa4\xc3\xb6\xc3\xbc', 'utf-8')
u'\xe4\xfc'
```

3.1.4 List

Python mengetahui sejumlah tipe data *compound* atau gabungan, yang digunakan untuk mengelompokkan nilai-nilai lainnya. Yang paling serbaguna adalah *list*, yang dapat ditulis sebagai daftar nilai yang dipisahkan koma (item) antara tanda kurung siku. List atau daftar mungkin berisi item dari tipe yang berbeda, tetapi biasanya semua item memiliki tipe yang sama.

```
>>> squares = [1, 4, 9, 16, 25]
>>> squares
[1, 4, 9, 16, 25]
```

Like strings (and all other built-in *sequence* type), lists can be indexed and sliced:

```
>>> squares[0] # indexing returns the item
1
>>> squares[-1]
25
>>> squares[-3:] # slicing returns a new list
[9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

```
>>> squares[:]
[1, 4, 9, 16, 25]
```

Lists also supports operations like concatenation:

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

Tidak seperti string, yang *immutable*, list adalah *mutable*, mis. dimungkinkan untuk mengubah kontennya:

```
>>> cubes = [1, 8, 27, 65, 125] # something's wrong here
>>> 4 ** 3 # the cube of 4 is 64, not 65!
64
>>> cubes[3] = 64 # replace the wrong value
>>> cubes
[1, 8, 27, 64, 125]
```

Anda juga dapat menambahkan item baru di akhir list, dengan menggunakan *method* `append()` (kita akan melihat lebih banyak tentang metode nanti):

```
>>> cubes.append(216) # add the cube of 6
>>> cubes.append(7 ** 3) # and the cube of 7
>>> cubes
[1, 8, 27, 64, 125, 216, 343]
```

Pemberian nilai untuk irisan juga dimungkinkan, dan ini bahkan dapat mengubah ukuran list atau menghapus seluruhnya:

```
>>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # replace some values
>>> letters[2:5] = ['C', 'D', 'E']
>>> letters
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # now remove them
>>> letters[2:5] = []
>>> letters
['a', 'b', 'f', 'g']
>>> # clear the list by replacing all the elements with an empty list
>>> letters[:] = []
>>> letters
[]
```

Fungsi bawaan `len()` juga berlaku untuk list:

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

Dimungkinkan untuk membuat list atau daftar bersarang (membuat daftar yang berisi daftar lain), misalnya:

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

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
[ 'a', 'b', 'c' ]
>>> x[0][1]
'b'
```

3.2 Langkah Awal Menuju Pemrograman

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```
>>> # Fibonacci series:
... # the sum of two elements defines the next
... a, b = 0, 1
>>> while b < 10:
...     print b
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Contoh ini memperkenalkan beberapa fitur baru.

- Baris pertama berisi *multiple assignment*: variabel `a` dan `b` secara bersamaan mendapatkan nilai-nilai baru 0 dan 1. Pada baris terakhir ini digunakan lagi, menunjukkan bahwa ekspresi di sisi sebelah kanan, semua dievaluasi terlebih dahulu sebelum salah satu pemberian nilai berlangsung. Ekspresi sisi kanan dievaluasi dari kiri ke kanan.
- The `while` loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: `<` (less than), `>` (greater than), `==` (equal to), `<=` (less than or equal to), `>=` (greater than or equal to) and `!=` (not equal to).
- *body* dari pengulangan adalah *indentasi*: indentasi adalah cara Python untuk pengelompokan pernyataan. Pada prompt interaktif, Anda harus mengetikkan tab atau spasi(-spasi) untuk setiap baris yang diberikan indentasi. Dalam praktiknya Anda akan menyiapkan masukan yang lebih rumit untuk Python dengan editor teks; semua editor teks yang baik memiliki fasilitas indentasi otomatis. Ketika pernyataan majemuk dimasukkan secara interaktif, harus diikuti oleh baris kosong untuk menunjukkan penyelesaian (karena pengurai tidak dapat menebak kapan Anda mengetik baris terakhir). Perhatikan bahwa setiap baris dalam blok dasar harus diindentasi dengan jumlah yang sama.
- The `print` statement writes the value of the expression(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple expressions and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
>>> i = 256*256
>>> print 'The value of i is', i
The value of i is 65536
```

A trailing comma avoids the newline after the output:


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

Note that the interpreter inserts a newline before it prints the next prompt if the last line was not completed.

Lebih Banyak Alat Pengatur Aliran *Control Flow*

Selain pernyataan `while` baru saja diperkenalkan, Python menggunakan pernyataan kontrol aliran yang biasa dikenal dari bahasa lain, dengan beberapa *twist*.

4.1 `if` Statements

Mungkin tipe pernyataan yang paling terkenal adalah pernyataan `if`. Sebagai contoh:

```
>>> x = int(raw_input("Please enter an integer: "))
Please enter an integer: 42
>>> if x < 0:
...     x = 0
...     print 'Negative changed to zero'
... elif x == 0:
...     print 'Zero'
... elif x == 1:
...     print 'Single'
... else:
...     print 'More'
...
More
```

There can be zero or more `elif` parts, and the `else` part is optional. The keyword `'elif'` is short for `'else if'`, and is useful to avoid excessive indentation. An `if ... elif ... elif ...` sequence is a substitute for the `switch` or `case` statements found in other languages.

4.2 for Statements

The `for` statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python's `for` statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
>>> # Measure some strings:
... words = ['cat', 'window', 'defenestrate']
>>> for w in words:
...     print w, len(w)
...
cat 3
window 6
defenestrate 12
```

If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```
>>> for w in words[:]: # Loop over a slice copy of the entire list.
...     if len(w) > 6:
...         words.insert(0, w)
...
>>> words
['defenestrate', 'cat', 'window', 'defenestrate']
```

4.3 Fungsi range ()

If you do need to iterate over a sequence of numbers, the built-in function `range ()` comes in handy. It generates lists containing arithmetic progressions:

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

The given end point is never part of the generated list; `range(10)` generates a list of 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the 'step'):

```
>>> range(5, 10)
[5, 6, 7, 8, 9]
>>> range(0, 10, 3)
[0, 3, 6, 9]
>>> range(-10, -100, -30)
[-10, -40, -70]
```

Untuk beralih pada indeks urutan, Anda dapat menggabungkan `range ()` dan `len ()` sebagai berikut:

```
>>> a = ['Mary', 'had', 'a', 'little', 'lamb']
>>> for i in range(len(a)):
...     print i, a[i]
...
0 Mary
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
1 had
2 a
3 little
4 lamb
```

Dalam kebanyakan kasus seperti itu, bagaimanapun, lebih mudah untuk menggunakan fungsi `enumerate()`, lihat *Teknik Perulangan*.

4.4 break and continue Statements, and else Clauses on Loops

Pernyataan `break`, seperti dalam C, keluar dari bagian terdalam yang terlampaui perulangan `for` atau `while`.

Loop statements may have an `else` clause; it is executed when the loop terminates through exhaustion of the list (with `for`) or when the condition becomes false (with `while`), but not when the loop is terminated by a `break` statement. This is exemplified by the following loop, which searches for prime numbers:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print n, 'equals', x, '*', n/x
...             break
...         else:
...             # loop fell through without finding a factor
...             print n, 'is a prime number'
...
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

(Ya, ini adalah kode yang benar. Perhatikan baik-baik: klausul `else` milik perulangan `for`, **not** pernyataan `if`.)

When used with a loop, the `else` clause has more in common with the `else` clause of a `try` statement than it does that of `if` statements: a `try` statement's `else` clause runs when no exception occurs, and a loop's `else` clause runs when no `break` occurs. For more on the `try` statement and exceptions, see *Menangani Pengecualian*.

Pernyataan `continue`, juga dipinjam dari C, melanjutkan dengan pengulangan berikutnya dari loop:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print "Found an even number", num
...         continue
...     print "Found a number", num
Found an even number 2
Found a number 3
Found an even number 4
Found a number 5
Found an even number 6
Found a number 7
Found an even number 8
Found a number 9
```

4.5 `pass` Statements

Pernyataan `pass` tidak melakukan apa-apa. Ini dapat digunakan ketika pernyataan diperlukan secara sintaksis tetapi program tidak memerlukan tindakan. Sebagai contoh:

```
>>> while True:
...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
... 
```

Ini biasanya digunakan untuk membuat kelas minimal:

```
>>> class MyEmptyClass:
...     pass
... 
```

Another place `pass` can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The `pass` is silently ignored:

```
>>> def initlog(*args):
...     pass # Remember to implement this!
... 
```

4.6 Mendefinisikan Fungsi

Kita dapat membuat fungsi yang menulis seri Fibonacci ke batas acak *arbitrary*:

```
>>> def fib(n): # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> # Now call the function we just defined:
... fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Kata kunci `def` memperkenalkan fungsi *definition*. Itu harus diikuti oleh nama fungsi dan daftar parameter formal yang di dalam tanda kurung. Pernyataan yang membentuk tubuh fungsi mulai dari baris berikutnya, dan harus diberi indentasi.

Pernyataan pertama dari tubuh fungsi secara opsional dapat berupa string literal; string literal ini adalah string dokumentasi fungsi, atau *docstring*. (Lebih lanjut tentang *docstring* dapat ditemukan di bagian [String Dokumentasi](#).) Ada alat yang menggunakan *docstring* untuk secara otomatis menghasilkan dokumentasi online atau cetak, atau untuk membiarkan pengguna menelusuri kode secara interaktif; itu praktik yang baik untuk memasukkan dokumen dalam kode yang Anda tulis, jadi biasakan seperti itu.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a `global` statement), although they may be referenced.

Parameter aktual (*arguments*) untuk panggilan fungsi diperkenalkan dalam tabel simbol lokal dari fungsi yang dipanggil ketika dipanggil; dengan demikian, argumen dilewatkan menggunakan *call by value* (di mana *value* selalu menjadi objek

reference, bukan nilai objek).¹ Ketika suatu fungsi memanggil fungsi lain, tabel simbol lokal baru dibuat untuk panggilan itu.

Definisi fungsi memperkenalkan nama fungsi dalam tabel simbol saat ini. Nilai nama fungsi memiliki tipe yang diakui oleh *interpreter* sebagai fungsi yang ditentukan pengguna. Nilai ini dapat ditetapkan ke nama lain yang kemudian dapat juga digunakan sebagai fungsi. Ini berfungsi sebagai mekanisme penggantian nama umum:

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

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a `return` statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using `print`:

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

Sangat mudah untuk menulis fungsi yang mengembalikan daftar *list* nomor seri Fibonacci, alih-alih mencetaknya:

```
>>> def fib2(n): # return Fibonacci series up to n
...     """Return a list containing the Fibonacci series up to n."""
...     result = []
...     a, b = 0, 1
...     while a < n:
...         result.append(a) # see below
...         a, b = b, a+b
...     return result
...
>>> f100 = fib2(100) # call it
>>> f100 # write the result
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

Contoh ini, seperti biasa, menunjukkan beberapa fitur Python baru:

- The `return` statement returns with a value from a function. `return` without an expression argument returns `None`. Falling off the end of a function also returns `None`.
- Pernyataan `result.append(a)` memanggil *method* dari objek daftar *list* `result`. Sebuah metode adalah fungsi yang 'milik' sebuah objek dan dinamai `obj.methodname`, di mana `obj` adalah suatu objek (ini mungkin sebuah ekspresi), dan `methodname` adalah nama dari metode yang ditentukan oleh tipe objek. Jenis yang berbeda menentukan metode yang berbeda. Metode tipe yang berbeda mungkin memiliki nama yang sama tanpa menimbulkan ambiguitas. (Dimungkinkan untuk menentukan jenis dan metode objek Anda sendiri, menggunakan *classes*, lihat [Classes](#)) Metode `append()` yang ditunjukkan pada contoh didefinisikan untuk objek daftar; itu menambahkan elemen baru di akhir daftar. Dalam contoh ini setara dengan `result = result + [a]`, tetapi lebih efisien.

¹ Sebenarnya, *call by object reference* akan menjadi deskripsi yang lebih baik, karena jika objek yang bisa ditransmisikan dilewatkan, pemanggil akan melihat perubahan yang dibuat oleh yang dipanggil *callee* (item dimasukkan ke dalam daftar).

4.7 Lebih lanjut tentang Mendefinisikan Fungsi

Dimungkinkan juga untuk mendefinisikan fungsi dengan sejumlah variabel argumen. Ada tiga bentuk, yang bisa digabungkan.

4.7.1 Nilai Argumen Bawaan

Bentuk yang paling berguna adalah menentukan nilai bawaan untuk satu atau lebih argumen. Ini menciptakan fungsi yang bisa dipanggil dengan argumen yang lebih sedikit daripada yang didefinisikan untuk diizinkan. Sebagai contoh:

```
def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
    while True:
        ok = raw_input(prompt)
        if ok in ('y', 'ye', 'yes'):
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise IOError('refusenik user')
        print complaint
```

Fungsi ini dapat dipanggil dengan beberapa cara:

- hanya memberikan argumen wajib: `ask_ok('Do you really want to quit?')`
- memberikan salah satu argumen opsional: `ask_ok('OK to overwrite the file?', 2)`
- atau bahkan memberikan semua argumen: `ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

Contoh ini juga memperkenalkan kata kunci `in`. Ini menguji apakah suatu urutan berisi nilai tertentu atau tidak.

Nilai bawaan dievaluasi pada titik definisi fungsi dalam lingkup *defining*, sehingga:

```
i = 5

def f(arg=i):
    print arg

i = 6
f()
```

akan mencetak 5.

Peringatan penting: Nilai bawaan dievaluasi hanya sekali. Ini membuat perbedaan ketika bawaan adalah objek yang dapat diubah seperti daftar *list*, kamus *dictionary*, atau *instances* dari sebagian besar kelas. Misalnya, fungsi berikut mengakumulasi argumen yang diteruskan pada panggilan berikutnya:

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

print f(1)
print f(2)
print f(3)
```

Ini akan mencetak:


```
[1]
[1, 2]
[1, 2, 3]
```

Jika Anda tidak ingin bawaan dibagi dengan panggilan berikutnya, Anda dapat menulis fungsi seperti ini sebagai gantinya:

```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L
```

4.7.2 Argumen Kata Kunci *Keyword Arguments*

Fungsi juga dapat dipanggil menggunakan *keyword argument* dari bentuk `kwarg=value`. Misalnya, fungsi berikut:

```
def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
    print "-- This parrot wouldn't", action,
    print "if you put", voltage, "volts through it."
    print "-- Lovely plumage, the", type
    print "-- It's", state, "!"
```

menerima satu argumen yang diperlukan (`voltage`) dan tiga argumen opsional (`state`, `action`, dan `type`). Fungsi ini dapat dipanggil dengan salah satu cara berikut:

```
parrot(1000)                                # 1 positional argument
parrot(voltage=1000)                        # 1 keyword argument
parrot(voltage=1000000, action='VOOOOOM')   # 2 keyword arguments
parrot(action='VOOOOOM', voltage=1000000)   # 2 keyword arguments
parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

tetapi semua pemanggilan berikut ini tidak valid:

```
parrot()                                # required argument missing
parrot(voltage=5.0, 'dead')             # non-keyword argument after a keyword argument
parrot(110, voltage=220)                 # duplicate value for the same argument
parrot(actor='John Cleese')             # unknown keyword argument
```

Dalam pemanggilan fungsi, argumen kata kunci *keyword argument* harus mengikuti argumen posisi. Semua argumen kata kunci *keyword argument* yang diteruskan harus cocok dengan salah satu argumen yang diterima oleh fungsi (mis. `actor` bukan argumen yang valid untuk fungsi `parrot`), dan urutannya tidak penting. Ini juga termasuk argumen non-opsional (mis. `parrot(voltage=1000)` juga valid). Tidak ada argumen yang dapat menerima nilai lebih dari sekali. Berikut ini contoh yang gagal karena batasan ini:

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

When a final formal parameter of the form `**name` is present, it receives a dictionary (see `typesmapping`) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter

of the form **name* (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (**name* must occur before ***name*.) For example, if we define a function like this:

```
def cheeseshop(kind, *arguments, **keywords):
    print "-- Do you have any", kind, "?"
    print "-- I'm sorry, we're all out of", kind
    for arg in arguments:
        print arg
    print "-" * 40
    keys = sorted(keywords.keys())
    for kw in keys:
        print kw, ":", keywords[kw]
```

Ini bisa disebut seperti ini:

```
cheeseshop("Limburger", "It's very runny, sir.",
           "It's really very, VERY runny, sir.",
           shopkeeper='Michael Palin',
           client="John Cleese",
           sketch="Cheese Shop Sketch")
```

dan tentu saja itu akan mencetak:

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, VERY runny, sir.
-----
client : John Cleese
shopkeeper : Michael Palin
sketch : Cheese Shop Sketch
```

Note that the list of keyword argument names is created by sorting the result of the keywords dictionary's `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

4.7.3 Daftar Argumen Berubah-ubah *Arbitrary*

Akhirnya, opsi yang paling jarang digunakan adalah menentukan bahwa suatu fungsi dapat dipanggil dengan sejumlah argumen acak *arbitrary*. Argumen-argumen ini akan dibungkus dalam sebuah tuple (lihat `tutuples`). Sebelum jumlah variabel argumen, nol atau lebih argumen normal dapat muncul.

```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

4.7.4 Pembukaan Paket *Unpacking* Daftar Argumen

Situasi sebaliknya terjadi ketika argumen sudah ada dalam daftar *list* atau tuple tetapi perlu dibongkar untuk panggilan fungsi yang membutuhkan argumen posisi terpisah. Sebagai contoh, fungsi bawaan `range()` mengharapkan argumen terpisah *start* dan *stop*. Jika tidak tersedia secara terpisah, tulis fungsi panggilan dengan operator-*** untuk membongkar argumen dari daftar *list* atau tuple:

```
>>> range(3, 6)                # normal call with separate arguments
[3, 4, 5]
>>> args = [3, 6]
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
>>> range(*args)           # call with arguments unpacked from a list
[3, 4, 5]
```

Dengan cara yang sama, kamus dapat mengirimkan argumen kata kunci dengan operator-******:

```
>>> def parrot(voltage, state='a stiff', action='vroom'):
...     print "-- This parrot wouldn't", action,
...     print "if you put", voltage, "volts through it.",
...     print "E's", state, "!"
...
>>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
>>> parrot(**d)
-- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin'
↳ demised !
```

4.7.5 Ekspresi Lambda

Fungsi kecil anonim dapat dibuat dengan kata kunci `lambda`. Fungsi ini mengembalikan jumlah dari dua argumennya: `lambda a, b: a+b`. Fungsi Lambda dapat digunakan di mana pun objek fungsi diperlukan. Mereka secara sintaksis terbatas pada satu ekspresi. Secara semantik, mereka hanya pemanis sintaksis untuk definisi fungsi normal. Seperti definisi fungsi bersarang, fungsi lambda dapat mereferensikan variabel dari cakupan yang mengandung

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

Contoh di atas menggunakan ekspresi lambda untuk mengembalikan fungsi. Penggunaan lain adalah untuk melewati fungsi kecil sebagai argumen:

```
>>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
>>> pairs.sort(key=lambda pair: pair[1])
>>> pairs
[(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

4.7.6 String Dokumentasi

There are emerging conventions about the content and formatting of documentation strings.

Baris pertama harus selalu berupa ringkasan singkat dan ringkas dari tujuan objek. Untuk singkatnya, itu tidak boleh secara eksplisit menyatakan nama atau jenis objek, karena ini tersedia dengan cara lain (kecuali jika nama tersebut merupakan kata kerja yang menggambarkan operasi fungsi). Baris ini harus dimulai dengan huruf kapital dan diakhiri dengan titik.

Jika ada lebih banyak baris dalam string dokumentasi, baris kedua harus kosong, memisahkan ringkasan secara visual dari sisa deskripsi. Baris berikut harus satu atau lebih paragraf yang menggambarkan konvensi pemanggilan objek, efek sampingnya, dll.

Pengurai Python tidak menghapus lekukan dari string multi-baris literal di Python, jadi alat yang memproses dokumentasi harus menghapus indentasi jika diinginkan. Ini dilakukan dengan menggunakan konvensi berikut. Baris tidak-kosong pertama *setelah* baris pertama string menentukan jumlah indentasi untuk seluruh string dokumentasi. (Kami tidak dapat

menggunakan baris pertama karena umumnya berbatasan dengan tanda kutip pembukaan string sehingga indentasinya tidak terlihat dalam string literal.) Spasi "equivalent" untuk indentasi ini kemudian dihilangkan dari awal semua baris string. Baris yang indentasi lebih sedikit seharusnya tidak terjadi, tetapi jika terjadi semua spasi *whitespace* utama harus dihilangkan. Kesetaraan spasi harus diuji setelah ekspansi tab (hingga 8 spasi, biasanya).

Berikut adalah contoh dari multi-baris *docstring*:

```
>>> def my_function():
...     """Do nothing, but document it.
...
...     No, really, it doesn't do anything.
...     """
...     pass
>>> print my_function.__doc__
Do nothing, but document it.

    No, really, it doesn't do anything.
```

4.8 Intermezzo: Gaya Coding

Sekarang Anda akan menulis potongan Python yang lebih panjang dan lebih kompleks, ini adalah saat yang tepat untuk berbicara tentang *coding style*. Sebagian besar bahasa dapat ditulis (atau lebih ringkas, *formatted*) dalam gaya yang berbeda; beberapa lebih mudah dibaca daripada yang lain. Memudahkan orang lain untuk membaca kode Anda selalu merupakan ide yang baik, dan mengadopsi gaya pengkodean yang bagus sangat membantu untuk itu.

Untuk Python, **PEP 8** telah muncul sebagai panduan gaya yang dipatuhi sebagian besar proyek; itu mempromosikan gaya pengkodean yang sangat mudah dibaca dan menyenangkan. Setiap pengembang Python harus membacanya di beberapa bagian; di sini adalah poin paling penting yang ditunjukkan untuk Anda:

- Gunakan lekukan 4-spasi, dan tanpa tab.
4 spasi adalah kompromi yang baik antara indentasi kecil (memungkinkan kedalaman bersarang lebih besar) dan indentasi besar (lebih mudah dibaca). Tab menimbulkan kebingungan, dan sebaiknya ditinggalkan.
- Bungkus *wrap* garis agar tidak melebihi 79 karakter.
Ini membantu pengguna dengan tampilan kecil dan memungkinkan untuk memiliki beberapa file kode berdampingan pada tampilan yang lebih besar.
- Gunakan baris kosong untuk memisahkan fungsi dan kelas, dan blok kode yang lebih besar di dalam fungsi.
- Jika memungkinkan, berikan komentar pada baris terkait.
- Gunakan String Dokumentasi *docstrings*.
- Gunakan spasi di sekitar operator dan setelah koma, tetapi tidak secara langsung di dalam konstruksi kurung *bracketing*: `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently; the convention is to use `CamelCase` for classes and `lower_case_with_underscores` for functions and methods. Always use `self` as the name for the first method argument (see [Pandangan Pertama tentang Kelas](#) for more on classes and methods).
- Don't use fancy encodings if your code is meant to be used in international environments. Plain ASCII works best in any case.

Bab ini menjelaskan beberapa hal yang telah Anda pelajari secara lebih rinci, dan menambahkan beberapa hal baru juga.

5.1 Lebih Lanjut tentang Daftar *Lists*

Tipe data daftar *list* memiliki beberapa metode lagi. Berikut ini semua metode dari objek daftar *list*:

`list.append(x)`

Add an item to the end of the list; equivalent to `a[len(a):] = [x]`.

`list.extend(L)`

Extend the list by appending all the items in the given list; equivalent to `a[len(a):] = L`.

`list.insert(i, x)`

Masukkan item pada posisi tertentu. Argumen pertama adalah indeks elemen sebelum memasukkan, jadi `a.insert(0, x)` memasukkan di bagian depan daftar *list*, dan `a.insert(len(a), x)` sama dengan `a.append(x)`.

`list.remove(x)`

Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.pop([i])`

Hapus item pada posisi yang diberikan dalam daftar, dan kembalikan. Jika tidak ada indeks yang ditentukan, `a.pop()` menghapus dan mengembalikan item terakhir dalam daftar. (Tanda kurung siku di sekitar *i* dalam pengenalan *signature* metode menunjukkan bahwa parameternya opsional, bukan Anda harus mengetik tanda kurung siku pada posisi itu. Anda akan sering melihat notasi ini di Referensi Pustaka Python.)

`list.index(x)`

Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

`list.count(x)`

Kembalikan berapa kali *x* muncul dalam daftar.

`list.sort(cmp=None, key=None, reverse=False)`

Urutkan item daftar di tempat (argumen dapat digunakan untuk mengurutkan ubahsuaian *customization*, lihat `sorted()` untuk penjelasannya).

`list.reverse()`

Reverse the elements of the list, in place.

Contoh yang menggunakan sebagian besar metode daftar *list*:

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

You might have noticed that methods like `insert`, `remove` or `sort` that only modify the list have no return value printed -- they return the default `None`. This is a design principle for all mutable data structures in Python.

5.1.1 Menggunakan Daftar *Lists* sebagai Tumpukan *Stacks*

Metode daftar membuatnya sangat mudah untuk menggunakan daftar *list* sebagai tumpukan *stack*, di mana elemen terakhir yang ditambahkan adalah elemen pertama yang diambil ("last-in, first-out"). Untuk menambahkan item ke atas tumpukan, gunakan `append()`. Untuk mengambil item dari atas tumpukan, gunakan `pop()` tanpa indeks eksplisit. Sebagai contoh:

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

5.1.2 Menggunakan Daftar *Lists* sebagai Antrian *Queues*

Dimungkinkan juga untuk menggunakan daftar sebagai antrian, di mana elemen pertama yang ditambahkan adalah elemen pertama yang diambil ("first-in, first-out"); namun, daftar tidak efisien untuk tujuan ini. Sementara menambahkan dan muncul dari akhir daftar cepat, melakukan memasukkan atau muncul dari awal daftar lambat (karena semua elemen lain harus digeser satu).

Untuk mengimplementasikan antrian, gunakan `collections.deque` yang dirancang untuk menambahkan dan muncul dengan cepat dari kedua ujungnya. Sebagai contoh:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # Terry arrives
>>> queue.append("Graham")         # Graham arrives
>>> queue.popleft()                # The first to arrive now leaves
'Eric'
>>> queue.popleft()                # The second to arrive now leaves
'John'
>>> queue                           # Remaining queue in order of arrival
deque(['Michael', 'Terry', 'Graham'])
```

5.1.3 Functional Programming Tools

There are three built-in functions that are very useful when used with lists: `filter()`, `map()`, and `reduce()`.

`filter(function, sequence)` returns a sequence consisting of those items from the sequence for which `function(item)` is true. If *sequence* is a str, unicode or tuple, the result will be of the same type; otherwise, it is always a list. For example, to compute a sequence of numbers divisible by 3 or 5:

```
>>> def f(x): return x % 3 == 0 or x % 5 == 0
...
>>> filter(f, range(2, 25))
[3, 5, 6, 9, 10, 12, 15, 18, 20, 21, 24]
```

`map(function, sequence)` calls `function(item)` for each of the sequence's items and returns a list of the return values. For example, to compute some cubes:

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

More than one sequence may be passed; the function must then have as many arguments as there are sequences and is called with the corresponding item from each sequence (or None if some sequence is shorter than another). For example:

```
>>> seq = range(8)
>>> def add(x, y): return x+y
...
>>> map(add, seq, seq)
[0, 2, 4, 6, 8, 10, 12, 14]
```

`reduce(function, sequence)` returns a single value constructed by calling the binary function *function* on the first two items of the sequence, then on the result and the next item, and so on. For example, to compute the sum of the numbers 1 through 10:

```
>>> def add(x, y): return x+y
...
>>> reduce(add, range(1, 11))
55
```

If there's only one item in the sequence, its value is returned; if the sequence is empty, an exception is raised.

A third argument can be passed to indicate the starting value. In this case the starting value is returned for an empty sequence, and the function is first applied to the starting value and the first sequence item, then to the result and the next item, and so on. For example,

```
>>> def sum(seq):
...     def add(x, y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Don't use this example's definition of `sum()`: since summing numbers is such a common need, a built-in function `sum(sequence)` is already provided, and works exactly like this.

5.1.4 Daftar *List Comprehensions*

Pemahaman daftar *list comprehensions* menyediakan cara singkat untuk membuat daftar. Aplikasi umum adalah membuat daftar baru di mana setiap elemen adalah hasil dari beberapa operasi yang diterapkan pada setiap anggota dari urutan lain atau *iterable*, atau untuk membuat urutan elemen-elemen yang memenuhi kondisi tertentu.

Misalnya, anggap kita ingin membuat daftar kotak, seperti:

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

We can obtain the same result with:

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

This is also equivalent to `squares = map(lambda x: x**2, range(10))`, but it's more concise and readable.

A list comprehension consists of brackets containing an expression followed by a `for` clause, then zero or more `for` or `if` clauses. The result will be a new list resulting from evaluating the expression in the context of the `for` and `if` clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

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

and it's equivalent to:

```
>>> combs = []
>>> for x in [1,2,3]:
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

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

```

Perhatikan bagaimana urutan pernyataan `for` dan `if` adalah sama di kedua cuplikan ini.

Jika ekspresi adalah tuple (mis. `(x, y)` dalam contoh sebelumnya), ekspresi tersebut harus diberi kurung.

```

>>> vec = [-4, -2, 0, 2, 4]
>>> # create a new list with the values doubled
>>> [x*2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filter the list to exclude negative numbers
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # apply a function to all the elements
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # call a method on each element
>>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
>>> [weapon.strip() for weapon in freshfruit]
['banana', 'loganberry', 'passion fruit']
>>> # create a list of 2-tuples like (number, square)
>>> [(x, x**2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # the tuple must be parenthesized, otherwise an error is raised
>>> [x, x**2 for x in range(6)]
File "<stdin>", line 1, in <module>
    [x, x**2 for x in range(6)]
        ^
SyntaxError: invalid syntax
>>> # flatten a list using a listcomp with two 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]

```

Pemahaman daftar *list comprehensions* dapat berisi ekspresi kompleks dan fungsi bersarang:

```

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

```

Pemahaman Daftar *List Comprehensions* Bersarang

Ekspresi awal dalam pemahaman daftar *list comprehension* dapat berupa ekspresi acak *arbitrary*, termasuk pemahaman daftar *list comprehension* lainnya.

Perhatikan contoh matriks 3x4 berikut yang diimplementasikan sebagai daftar *list* 3 dari daftar *list* panjang 4

```

>>> matrix = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],

```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
...     [9, 10, 11, 12],
... ]
```

Pemahaman daftar *list comprehension* berikut akan mengubah baris dan kolom:

```
>>> [[row[i] for row in matrix] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Seperti yang kita lihat di bagian sebelumnya, *listcomp* bersarang dievaluasi dalam konteks *for* yang mengikutinya, jadi contoh ini setara dengan:

```
>>> transposed = []
>>> for i in range(4):
...     transposed.append([row[i] for row in matrix])
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

yang, pada gilirannya, sama dengan:

```
>>> transposed = []
>>> for i in range(4):
...     # the following 3 lines implement the nested listcomp
...     transposed_row = []
...     for row in matrix:
...         transposed_row.append(row[i])
...     transposed.append(transposed_row)
...
>>> transposed
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Di dunia nyata, Anda harus memilih fungsi bawaan untuk pernyataan aliran *flow* yang kompleks. Fungsi `zip()` akan melakukan pekerjaan yang baik untuk kasus penggunaan ini:

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

Lihat tut-unpacking-argumen untuk detail tentang tanda bintang *asterisk* di baris ini.

5.2 The `del` statement

There is a way to remove an item from a list given its index instead of its value: the `del` statement. This differs from the `pop()` method which returns a value. The `del` statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

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

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
>>> a
[]
```

`del` juga dapat digunakan untuk menghapus seluruh variabel:

```
>>> del a
```

Merujuk nama `a` selanjutnya adalah kesalahan (setidaknya sampai nilai lain ditetapkan untuknya). Kita akan menemukan kegunaan lain untuk `del` nanti.

5.3 Tuples and *Urutan* Sequences

Kita melihat bahwa daftar *list* dan string memiliki banyak properti yang sama, seperti operasi pengindeksan dan pemotongan. Mereka adalah dua contoh tipe data *sequence* (lihat `typeseq`). Karena Python adalah bahasa yang berkembang, tipe data urutan lainnya dapat ditambahkan. Ada juga tipe data urutan standar lain: *tuple*.

Sebuah *tuple* terdiri dari sejumlah nilai yang dipisahkan oleh koma, misalnya:

```
>>> t = 12345, 54321, 'hello!'
>>> t[0]
12345
>>> t
(12345, 54321, 'hello!')
>>> # Tuples may be nested:
... u = t, (1, 2, 3, 4, 5)
>>> u
((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
>>> # Tuples are immutable:
... t[0] = 88888
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> # but they can contain mutable objects:
... v = ([1, 2, 3], [3, 2, 1])
>>> v
([1, 2, 3], [3, 2, 1])
```

Seperti yang Anda lihat, pada *tuple* keluaran selalu tertutup dalam tanda kurung, sehingga *tuple* bersarang *nester* ditafsirkan dengan benar; mereka mungkin dimasukkan dengan atau tanpa tanda kurung di sekitarnya, meskipun seringkali tanda kurung diperlukan pula (jika *tuple* adalah bagian dari ekspresi yang lebih besar). Tidak mungkin untuk memberikan nilai ke masing-masing item *tuple*, namun dimungkinkan untuk membuat *tuple* yang berisi objek yang bisa berubah *mutable*, seperti daftar.

Meskipun *tuple* mungkin mirip dengan daftar, *tuple* sering digunakan dalam situasi yang berbeda dan untuk tujuan yang berbeda. *Tuples* adalah *immutable*, dan biasanya berisi urutan elemen yang heterogen yang diakses melalui *unpacking* (lihat nanti di bagian ini) atau pengindeksan (atau bahkan berdasarkan atribut dalam kasus `namedtuples` `<collections.namedtuple>`). Daftar adalah `:term:`mutable()``, dan elemen-elemennya biasanya homogen dan diakses dengan menyusuri *iterating* daftar *list*.

Masalah khusus adalah pembangunan *tuple* yang mengandung 0 atau 1 item: sintaksis memiliki beberapa kebiasaan *quirks* tambahan untuk mengakomodasi ini. *Tuple* kosong dibangun oleh sepasang kurung kosong; *tuple* dengan satu item dikonstruksi dengan mengikuti nilai dengan koma (tidak cukup untuk menyertakan nilai tunggal dalam tanda kurung). Jelek, tapi efektif. Sebagai contoh:

```
>>> empty = ()
>>> singleton = 'hello',      # <-- note trailing comma
>>> len(empty)
0
>>> len(singleton)
1
>>> singleton
('hello',)
```

Pernyataan `t = 12345, 54321, 'hello!'` adalah contoh dari *tuple packing*: nilainya 12345, 54321 dan 'hello!' dikemas bersama-sama dalam *tuple*. Operasi terbalik juga dimungkinkan

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

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires the list of variables on the left to have the same number of elements as the length of the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

5.4 Himpunan Set

Python juga menyertakan tipe data untuk *sets*. Himpunan atau *Set* adalah koleksi yang tidak terurut tanpa elemen duplikat. Penggunaan dasar termasuk pengujian keanggotaan dan menghilangkan entri duplikat. Atur objek juga mendukung operasi matematika seperti penyatuan *union*, persimpangan *intersection*, perbedaan *difference*, dan perbedaan simetris.

Kurung kurawal atau fungsi `set()` dapat digunakan untuk membuat himpunan. Catatan: untuk membuat himpunan kosong Anda harus menggunakan `set()`, bukan `{}`; yang terakhir itu membuat kamus *dictionary* kosong, struktur data yang kita bahas di bagian selanjutnya.

Berikut ini adalah demonstrasi singkat:

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> fruit = set(basket)                # create a set without duplicates
>>> fruit
set(['orange', 'pear', 'apple', 'banana'])
>>> 'orange' in fruit                  # fast membership testing
True
>>> 'crabgrass' in fruit
False

>>> # Demonstrate set operations on unique letters from two words
...
>>> a = set('abracadabra')
>>> b = set('alacazam')
>>> a                                # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> a - b                            # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                            # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                            # letters in both a and b
set(['a', 'c'])
>>> a ^ b                            # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])
```

Seperti halnya untuk *list comprehensions*, *set comprehensions* juga didukung:

```
>>> a = {x for x in 'abracadabra' if x not in 'abc'}
>>> a
set(['r', 'd'])
```

5.5 Kamus *Dictionaries*

Tipe data lain yang berguna yang dibangun ke dalam Python adalah *dictionary* (lihat *typesmapping*). Kamus *dictionary* kadang-kadang ditemukan dalam bahasa lain sebagai "associative memories" atau "associative array". Tidak seperti urutan *sequences*, yang diindeks oleh sejumlah angka, kamus *dictionary* diindeks oleh *keys*, yang dapat berupa jenis apa pun yang tidak dapat diubah *immutable type*; string dan angka selalu bisa menjadi kunci *key*. *Tuples* dapat digunakan sebagai kunci jika hanya berisi string, angka, atau *tuple*; jika sebuah *tuple* berisi objek yang bisa berubah baik secara langsung atau tidak langsung, itu tidak dapat digunakan sebagai kunci *key*. Anda tidak dapat menggunakan daftar *list* sebagai kunci, karena daftar dapat dimodifikasi di tempat menggunakan penugasan indeks, penugasan *slice*, atau metode seperti `append()` dan `extend()`.

It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of *key:value* pairs within the braces adds initial *key:value* pairs to the dictionary; this is also the way dictionaries are written on output.

Operasi utama pada kamus *dictionary* adalah menyimpan nilai dengan beberapa kunci *key* dan mengekstraksi nilai yang diberikan kunci *key*. Dimungkinkan juga untuk menghapus pasangan kunci:nilai dengan `del`. Jika Anda menyimpan menggunakan kunci yang sudah digunakan, nilai lama yang terkait dengan kunci itu dilupakan. Merupakan kesalahan untuk mengekstraksi nilai menggunakan kunci yang tidak ada.

The `keys()` method of a dictionary object returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just apply the `sorted()` function to it). To check whether a single key is in the dictionary, use the `in` keyword.

Ini adalah contoh kecil menggunakan kamus *dictionary*:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack': 4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack': 4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
```

Pembangun *constructor* `dict()` membangun kamus langsung dari urutan pasangan kunci-nilai:

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

Selain itu, pemahaman kamus *dict comprehensions* dapat digunakan untuk membuat kamus *dictionary* dari ekspresi kunci dan nilai acak *arbitrary*:

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

Ketika kunci adalah string sederhana, kadang-kadang lebih mudah untuk menentukan pasangan menggunakan argumen kata kunci *keyword arguments*:

```
>>> dict(sape=4139, guido=4127, jack=4098)
{'sape': 4139, 'jack': 4098, 'guido': 4127}
```

5.6 Teknik Perulangan

Saat mengulang melalui urutan, indeks posisi dan nilai terkait dapat diambil pada saat yang sama menggunakan fungsi `enumerate()`.

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

Untuk mengulang dua urutan atau lebih secara bersamaan, entri dapat dipasangkan dengan fungsi `zip()`.

```
>>> questions = ['name', 'quest', 'favorite color']
>>> answers = ['lancelot', 'the holy grail', 'blue']
>>> for q, a in zip(questions, answers):
...     print 'What is your {0}? It is {1}'.format(q, a)
...
What is your name? It is lancelot.
What is your quest? It is the holy grail.
What is your favorite color? It is blue.
```

Untuk mengulang urutan secara terbalik, pertama tentukan urutan dalam arah maju dan kemudian panggil fungsi `reversed()`.

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

Untuk mengulangi sebuah urutan *sequence* dalam susunan yang diurutkan, gunakan fungsi `sort()` yang mengembalikan daftar terurut baru dengan membiarkan sumber tidak diubah.

```
>>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
>>> for f in sorted(set(basket)):
...     print f
...
apple
banana
orange
pear
```

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the `iteritems()` method.

```
>>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
>>> for k, v in knights.iteritems():
...     print k, v
...
gallahad the pure
robin the brave
```

Terkadang tergoda untuk mengubah daftar *list* saat Anda mengulanginya; namun, seringkali lebih mudah dan aman untuk membuat daftar *list* baru.

```
>>> import math
>>> raw_data = [56.2, float('NaN'), 51.7, 55.3, 52.5, float('NaN'), 47.8]
>>> filtered_data = []
>>> for value in raw_data:
...     if not math.isnan(value):
...         filtered_data.append(value)
...
>>> filtered_data
[56.2, 51.7, 55.3, 52.5, 47.8]
```

5.7 Lebih lanjut tentang Kondisi

Kondisi yang digunakan dalam pernyataan `while` dan `if` dapat berisi operator apa pun, bukan hanya perbandingan.

Operator perbandingan `in` dan `not in` memeriksa apakah suatu nilai terjadi (tidak terjadi) secara berurutan. Operator `is` dan `is not` membandingkan apakah dua objek benar-benar objek yang sama; ini hanya penting untuk objek yang dapat diubah seperti daftar *list*. Semua operator pembandingan memiliki prioritas yang sama, yang lebih rendah daripada semua operator numerik.

Perbandingan bisa dibuat berantai. Sebagai contoh, `a < b == c` menguji apakah `a` kurang dari `b` dan apa `b` sama dengan `c`.

Perbandingan dapat digabungkan menggunakan operator Boolean `and` dan `or`, dan hasil perbandingan (atau ekspresi Boolean lainnya) dapat dinegasikan dengan `not`. Ini memiliki prioritas lebih rendah daripada operator pembandingan; di antara mereka, `not` memiliki prioritas tertinggi dan `or` terendah, sehingga `A and not B or C` setara dengan `(A and (not B)) or C`. Seperti biasa, tanda kurung dapat digunakan untuk mengekspresikan komposisi yang diinginkan.

Operator Boolean `and` dan `or` disebut operator *short-circuit*: argumen mereka dievaluasi dari kiri ke kanan, dan evaluasi berhenti segera setelah hasilnya ditentukan. Misalnya, jika `A` dan `C` bernilai benar tetapi `B` salah, `A and B and C` tidak mengevaluasi ekspresi `C`. Ketika digunakan sebagai nilai umum dan bukan sebagai Boolean, nilai kembalian dari operator hubung singkat *short-circuit* adalah argumen terakhir yang dievaluasi.

Dimungkinkan untuk menetapkan hasil perbandingan atau ekspresi Boolean lainnya ke variabel. Sebagai contoh,

```
>>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
>>> non_null = string1 or string2 or string3
>>> non_null
'Trondheim'
```

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: `typing = in an expression when == was intended`.

5.8 Membandingkan Urutan *Sequences* dan Jenis Lainnya

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the ASCII ordering for individual characters. Some examples of comparisons between sequences of the same type:

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

Note that comparing objects of different types is legal. The outcome is deterministic but arbitrary: the types are ordered by their name. Thus, a list is always smaller than a string, a string is always smaller than a tuple, etc.¹ Mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc.

¹ The rules for comparing objects of different types should not be relied upon; they may change in a future version of the language.

Jika Anda berhenti dari *interpreter* Python dan memasukkannya lagi, definisi yang Anda buat (fungsi dan variabel) akan hilang. Karena itu, jika Anda ingin menulis program yang agak lebih panjang, Anda lebih baik menggunakan editor teks untuk menyiapkan input bagi penerjemah dan menjalankannya dengan file itu sebagai input. Ini dikenal sebagai membuat *script*. Saat program Anda menjadi lebih panjang, Anda mungkin ingin membaginya menjadi beberapa file untuk pengelolaan yang lebih mudah. Anda mungkin juga ingin menggunakan fungsi praktis yang Anda tulis di beberapa program tanpa menyalin definisi ke setiap program.

Untuk mendukung ini, Python memiliki cara untuk meletakkan definisi dalam file dan menggunakannya dalam skrip atau dalam contoh interaktif dari *interpreter*. File seperti itu disebut *module*; definisi dari modul dapat *imported* ke modul lain atau ke modul *main* (kumpulan variabel yang Anda memiliki akses ke dalam skrip yang dieksekusi di tingkat atas dan dalam mode kalkulator).

Modul adalah file yang berisi definisi dan pernyataan Python. Nama berkas adalah nama modul dengan akhiran `.py` diakhirnya. Dalam sebuah modul, nama modul (sebagai string) tersedia sebagai nilai variabel global `__name__`. Misalnya, gunakan editor teks favorit Anda untuk membuat bernama bernama `fibonacci.py` di direktori saat ini dengan konten berikut

```
# Fibonacci numbers module

def fib(n):    # write Fibonacci series up to n
    a, b = 0, 1
    while b < n:
        print b,
        a, b = b, a+b

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

Sekarang masukkan interpreter Python dan impor modul ini dengan perintah berikut:

```
>>> import fibo
```

Ini tidak memasukkan nama fungsi yang didefinisikan dalam `fibo` secara langsung dalam tabel simbol saat ini; itu hanya memasukkan nama modul `fibo` di sana. Menggunakan nama modul Anda dapat mengakses fungsi:

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

Jika Anda sering ingin menggunakan suatu fungsi, Anda dapat menetapkan ke nama lokal:

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

6.1 Lebih lanjut tentang Modul

Modul dapat berisi pernyataan yang dapat dieksekusi serta definisi fungsi. Pernyataan ini dimaksudkan untuk menginisialisasi modul. Mereka dieksekusi hanya *first* kali nama modul ditemui dalam pernyataan impor.¹ (Mereka juga dijalankan jika file dieksekusi sebagai skrip.)

Setiap modul memiliki tabel simbol pribadi sendiri, yang digunakan sebagai tabel simbol global oleh semua fungsi yang didefinisikan dalam modul. Dengan demikian, penulis modul dapat menggunakan variabel global dalam modul tanpa khawatir tentang bentrokan tidak disengaja dengan variabel global pengguna. Di sisi lain, jika Anda tahu apa yang Anda lakukan, Anda dapat menyentuh variabel global modul dengan notasi yang sama yang digunakan untuk merujuk ke fungsinya, `modname.itemname`.

Modul dapat mengimpor modul lain. Biasanya, tetapi tidak diperlukan untuk menempatkan semua pernyataan `import` di awal modul (atau skrip, dalam hal ini). Nama-nama modul yang diimpor ditempatkan di tabel simbol global modul impor.

Ada varian dari pernyataan `import` yang mengimpor nama dari modul langsung ke tabel simbol modul impor. Sebagai contoh:

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

Ini tidak memperkenalkan nama modul dari mana impor diambil dalam tabel simbol lokal (jadi dalam contoh, `fibo` tidak didefinisikan).

Bahkan ada varian untuk mengimpor semua nama yang didefinisikan oleh modul:

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

This imports all names except those beginning with an underscore (`_`).

Perhatikan bahwa secara umum praktik mengimpor `*` dari modul atau paket tidak disukai, karena sering menyebabkan kode yang kurang dapat dibaca. Namun, boleh saja menggunakannya untuk menghemat pengetikan di sesi interaktif.

¹ Bahkan definisi fungsi juga 'statements' yang 'executed'; eksekusi dari definisi fungsi level-modul memasukkan nama fungsi di tabel simbol global modul.

If the module name is followed by `as`, then the name following `as` is bound directly to the imported module.

```
>>> import fibo as fib
>>> fib.fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ini secara efektif mengimpor modul dengan cara yang sama dengan `import fibo` akan dilakukan, dengan satu-satunya perbedaan adalah sebagai `fib`.

Itu juga dapat digunakan ketika menggunakan `from` dengan efek yang sama:

```
>>> from fibo import fib as fibonacci
>>> fibonacci(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Catatan: For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter -- or, if it's just one module you want to test interactively, use `reload()`, e.g. `reload(modulename)`.

6.1.1 Mengoperasikan modul sebagai skrip

Ketika Anda mengoperasikan modul Python dengan

```
python fibo.py <arguments>
```

kode dalam modul akan dieksekusi, sama seperti jika Anda mengimpornya, tetapi dengan `__name__` diatur ke `"__main__"`. Itu berarti bahwa dengan menambahkan kode ini di akhir modul Anda

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

Anda dapat membuat berkas dapat digunakan sebagai skrip dan juga modul yang dapat diimpor, karena kode yang mengurai *parsing* baris perintah hanya beroperasi jika modul dieksekusi sebagai berkas "main":

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

Jika modul diimpor, kode ini tidak dioperasikan

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

Ini sering digunakan baik untuk menyediakan antarmuka pengguna yang nyaman ke modul, atau untuk tujuan pengujian (menjalankan modul sebagai skrip mengeksekusi rangkaian pengujian).

6.1.2 Jalur Pencarian Modul

Ketika sebuah modul bernama `spam` diimpor, *interpreter* pertama-tama mencari modul bawaan dengan nama itu. Jika tidak ditemukan, ia kemudian mencari berkas bernama `spam.py` dalam daftar direktori yang diberikan oleh variabel `sys.path`. `sys.path` diinisialisasi dari lokasi ini:

- the directory containing the input script (or the current directory).
- `PYTHONPATH` (daftar nama direktori, dengan sintaksis yang sama dengan variabel shell `PATH`).
- the installation-dependent default.

Setelah inisialisasi, program Python dapat memodifikasi: data `sys.path`. Direktori yang berisi skrip yang dijalankan ditempatkan di awal jalur pencarian, di depan jalur pustaka standar. Ini berarti bahwa skrip dalam direktori itu akan dimuat bukan modul dengan nama yang sama di direktori pustaka. Ini adalah kesalahan kecuali penggantian memang diharapkan. Lihat bagian *Modul Standar* untuk informasi lebih lanjut.

6.1.3 Berkas Python "Compiled"

As an important speed-up of the start-up time for short programs that use a lot of standard modules, if a file called `spam.pyc` exists in the directory where `spam.py` is found, this is assumed to contain an already-"byte-compiled" version of the module `spam`. The modification time of the version of `spam.py` used to create `spam.pyc` is recorded in `spam.pyc`, and the `.pyc` file is ignored if these don't match.

Normally, you don't need to do anything to create the `spam.pyc` file. Whenever `spam.py` is successfully compiled, an attempt is made to write the compiled version to `spam.pyc`. It is not an error if this attempt fails; if for any reason the file is not written completely, the resulting `spam.pyc` file will be recognized as invalid and thus ignored later. The contents of the `spam.pyc` file are platform independent, so a Python module directory can be shared by machines of different architectures.

Beberapa tips untuk para ahli:

- When the Python interpreter is invoked with the `-O` flag, optimized code is generated and stored in `.pyo` files. The optimizer currently doesn't help much; it only removes `assert` statements. When `-O` is used, *all bytecode* is optimized; `.pyc` files are ignored and `.py` files are compiled to optimized bytecode.
- Passing two `-O` flags to the Python interpreter (`-OO`) will cause the bytecode compiler to perform optimizations that could in some rare cases result in malfunctioning programs. Currently only `__doc__` strings are removed from the bytecode, resulting in more compact `.pyo` files. Since some programs may rely on having these available, you should only use this option if you know what you're doing.
- A program doesn't run any faster when it is read from a `.pyc` or `.pyo` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` or `.pyo` files is the speed with which they are loaded.
- When a script is run by giving its name on the command line, the bytecode for the script is never written to a `.pyc` or `.pyo` file. Thus, the startup time of a script may be reduced by moving most of its code to a module and having a small bootstrap script that imports that module. It is also possible to name a `.pyc` or `.pyo` file directly on the command line.
- It is possible to have a file called `spam.pyc` (or `spam.pyo` when `-O` is used) without a file `spam.py` for the same module. This can be used to distribute a library of Python code in a form that is moderately hard to reverse engineer.
- The module `compileall` can create `.pyc` files (or `.pyo` files when `-O` is used) for all modules in a directory.

6.2 Modul Standar

Python dilengkapi dengan pustaka modul standar, yang dijelaskan dalam dokumen terpisah, Referensi Pustaka Python ("Library Reference" selanjutnya). Beberapa modul dibangun ke dalam interpreter; ini menyediakan akses ke operasi yang bukan bagian dari inti bahasa tetapi tetap dibangun, baik untuk efisiensi atau untuk menyediakan akses ke sistem operasi primitif seperti pemanggilan sistem. Himpunan modul tersebut adalah opsi konfigurasi yang juga tergantung pada platform yang mendasarinya. Sebagai contoh, modul `winreg` hanya disediakan pada sistem Windows. Satu modul tertentu patut mendapat perhatian: `sys`, yang dibangun ke dalam setiap interpreter Python. Variabel `sys.ps1` dan `sys.ps2` menentukan string yang digunakan sebagai prompt primer dan sekunder

```
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C> print 'Yuck!'
Yuck!
C>
```

Kedua variabel ini hanya ditentukan jika interpreter dalam mode interaktif.

Variabel `sys.path` adalah daftar string yang menentukan jalur pencarian *interpreter* untuk modul. Ini diinisialisasi ke jalur default yang diambil dari variabel lingkungan `PYTHONPATH`, atau dari bawaan bawaan jika `PYTHONPATH` tidak disetel. Anda dapat memodifikasinya menggunakan operasi standar untuk *list*:

```
>>> import sys
>>> sys.path.append('/ufs/guido/lib/python')
```

6.3 Fungsi `dir()`

Fungsi bawaan `dir()` digunakan untuk mencari tahu nama-nama yang ditentukan oleh modul. Ia mengembalikan *list* string yang diurutkan:

```
>>> import fibo, sys
>>> dir(fibo)
['__name__', 'fib', 'fib2']
>>> dir(sys)
['__displayhook__', '__doc__', '__excepthook__', '__name__', '__package__',
'__stderr__', '__stdin__', '__stdout__', '__clear_type_cache__',
'_current_frames', '_getframe', '_mercurial', 'api_version', 'argv',
'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
'copyright', 'displayhook', 'dont_write_bytecode', 'exc_clear', 'exc_info',
'exc_traceback', 'exc_type', 'exc_value', 'excepthook', 'exec_prefix',
'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
'getrefcount', 'getsizeof', 'gettotalrefcount', 'gettrace', 'hexversion',
'long_info', 'maxint', 'maxsize', 'maxunicode', 'meta_path', 'modules',
'path', 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
'py3kwarning', 'setcheckinterval', 'setdlopenflags', 'setprofile',
'setrecursionlimit', 'settrace', 'stderr', 'stdin', 'stdout', 'subversion',
'version', 'version_info', 'warnoptions']
```

Tanpa argumen, `dir()` mencantumkan nama yang telah Anda tentukan saat ini:

```
>>> a = [1, 2, 3, 4, 5]
>>> import fibo
>>> fib = fibo.fib
>>> dir()
['__builtins__', '__name__', '__package__', 'a', 'fib', 'fibo', 'sys']
```

Perhatikan bahwa ini mencantumkan semua jenis nama: variabel, modul, fungsi, dll.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module `__builtin__`:

```
>>> import __builtin__
>>> dir(__builtin__)
['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
'BufferError', 'BytesWarning', 'DeprecationWarning', 'EOFError',
'Ellipsis', 'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError', 'ImportWarning',
'IndentationError', 'IndexError', 'KeyError', 'KeyboardInterrupt',
'LookupError', 'MemoryError', 'NameError', 'None', 'NotImplemented',
'NotImplementedError', 'OSError', 'OverflowError',
'PendingDeprecationWarning', 'ReferenceError', 'RuntimeError',
'RuntimeWarning', 'StandardError', 'StopIteration', 'SyntaxError',
'SyntaxWarning', 'SystemError', 'SystemExit', 'TabError', 'True',
'TypeError', 'UnboundLocalError', 'UnicodeDecodeError',
'UnicodeEncodeError', 'UnicodeError', 'UnicodeTranslateError',
'UnicodeWarning', 'UserWarning', 'ValueError', 'Warning',
'ZeroDivisionError', '_', '__debug__', '__doc__', '__import__',
'__name__', '__package__', 'abs', 'all', 'any', 'apply', 'basestring',
'bin', 'bool', 'buffer', 'bytearray', 'bytes', 'callable', 'chr',
'classmethod', 'cmp', 'coerce', 'compile', 'complex', 'copyright',
'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input',
'int', 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license',
'list', 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next',
'object', 'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit',
'range', 'raw_input', 'reduce', 'reload', 'repr', 'reversed', 'round',
'set', 'setattr', 'slice', 'sorted', 'staticmethod', 'str', 'sum', 'super',
'tuple', 'type', 'unichr', 'unicode', 'vars', 'xrange', 'zip']
```

6.4 Paket

Paket adalah cara penataan *namespace* modul Python dengan menggunakan "dotted module names". Sebagai contoh, nama modul `A.B` menetapkan submodule bernama `B` dalam sebuah paket bernama `A`. Sama seperti penggunaan modul menyelamatkan penulis modul yang berbeda dari harus khawatir tentang nama variabel global masing-masing, penggunaan nama modul bertitik menyelamatkan penulis paket multi-modul seperti NumPy atau Pillow dari harus khawatir tentang nama modul masing-masing.

Misalkan Anda ingin merancang koleksi modul ("paket") untuk penanganan berkas suara dan data suara yang seragam. Ada banyak format berkas suara yang berbeda (biasanya dikenali oleh ekstensi mereka, misalnya: `.wav`, `.aiff`, `.au`), jadi Anda mungkin perlu membuat dan memelihara koleksi modul yang terus bertambah untuk konversi antara berbagai format file. Ada juga banyak operasi berbeda yang mungkin ingin Anda lakukan pada data suara (seperti mencampur, menambahkan gema, menerapkan fungsi equalizer, menciptakan efek stereo buatan), jadi selain itu Anda akan menulis aliran modul tanpa henti untuk melakukan operasi ini. Berikut adalah struktur yang mungkin untuk paket Anda (dinyatakan dalam hierarki sistem file):

```

sound/                                Top-level package
__init__.py                           Initialize the sound package
formats/                              Subpackage for file format conversions
    __init__.py
    wavread.py
    wavwrite.py
    aiffread.py
    aiffwrite.py
    auread.py
    auwrite.py
    ...
effects/                              Subpackage for sound effects
    __init__.py
    echo.py
    surround.py
    reverse.py
    ...
filters/                              Subpackage for filters
    __init__.py
    equalizer.py
    vocoder.py
    karaoke.py
    ...

```

Saat mengimpor paket, Python mencari melalui direktori pada `sys.path` mencari subdirektori paket.

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Pengguna paket dapat mengimpor modul individual dari paket, misalnya:

```
import sound.effects.echo
```

Ini memuat submodule `sound.effects.echo`. Itu harus dirujuk dengan nama lengkapnya.

```
sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

Cara alternatif mengimpor submodule adalah:

```
from sound.effects import echo
```

Ini juga memuat submodul `:mod: echo`, dan membuatnya tersedia tanpa awalan paketnya, sehingga dapat digunakan sebagai berikut:

```
echo.echofilter(input, output, delay=0.7, atten=4)
```

Namun variasi lain adalah mengimpor fungsi atau variabel yang diinginkan secara langsung:

```
from sound.effects.echo import echofilter
```

Sekali lagi, ini memuat submodul `echo`, tetapi ini membuat fungsinya `echofilter()` langsung tersedia:

```
echofilter(input, output, delay=0.7, atten=4)
```

Perhatikan bahwa ketika menggunakan `from package import item`, item tersebut dapat berupa submodul (atau subpaket) dari paket, atau beberapa nama lain yang ditentukan dalam paket, seperti fungsi, kelas atau variabel. Pernyataan

`import` pertama menguji apakah item tersebut didefinisikan dalam paket; jika tidak, ini dianggap sebagai modul dan mencoba memuatnya. Jika gagal menemukannya, pengecualian `ImportError` dimunculkan.

Sebaliknya, ketika menggunakan sintaksis seperti `import item.subitem.subsubitem`, setiap item kecuali yang terakhir harus berupa paket; item terakhir dapat berupa modul atau paket tetapi tidak bisa berupa kelas atau fungsi atau variabel yang didefinisikan dalam item sebelumnya.

6.4.1 Mengimpor * Dari Paket

Sekarang apa yang terjadi ketika pengguna menulis `from sound.effects import *`? Idealnya, orang akan berharap bahwa ini entah bagaimana keluar ke sistem file, menemukan submodul mana yang ada dalam paket, dan mengimpor semuanya. Ini bisa memakan waktu lama dan mengimpor submodul mungkin memiliki efek samping yang tidak diinginkan yang seharusnya hanya terjadi ketika submodul diimpor secara eksplisit.

Satu-satunya solusi adalah bagi pembuat paket untuk memberikan indeks paket secara eksplisit. Pernyataan `import` menggunakan konvensi berikut: jika suatu paket punya kode `__init__.py` yang mendefinisikan daftar bernama `__all__`, itu diambil sebagai daftar nama modul yang harus diimpor ketika `from package import *` ditemukan. Terserah pembuat paket untuk tetap memperbarui daftar ini ketika versi baru dari paket dirilis. Pembuat paket juga dapat memutuskan untuk tidak mendukungnya, jika mereka tidak melihat penggunaan untuk mengimpor `*` dari paket mereka. Sebagai contoh, berkas `sound/effects/__init__.py` dapat berisi kode berikut:

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

Ini berarti bahwa `from sound.effects import *` akan mengimpor tiga submodul bernama dari paket `sound`.

Jika `__all__` tidak didefinisikan, pernyataan `from sound.effects import *` *tidak* impor semua submodul dari paket `sound.effects` ke *namespace* saat ini; itu hanya memastikan bahwa paket `sound.effects` telah diimpor (mungkin menjalankan kode inisialisasi apa pun di `__init__.py`) dan kemudian mengimpor nama apa pun yang didefinisikan dalam paket. Ini termasuk semua nama yang didefinisikan (dan submodul yang dimuat secara eksplisit) oleh `__init__.py`. Itu juga termasuk semua submodul dari paket yang secara eksplisit dimuat oleh sebelumnya pernyataan `import`. Pertimbangkan kode ini

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

Dalam contoh ini, modul `echo` dan `surround` diimpor dalam *namespace* saat ini karena mereka didefinisikan dalam paket `sound.effects` ketika paket `from...import` Pernyataan dieksekusi. (Ini juga berfungsi ketika `__all__` didefinisikan.)

Meskipun modul-modul tertentu dirancang hanya untuk mengekspos nama-nama yang mengikuti pola tertentu ketika Anda menggunakan `import *`, itu masih dianggap praktik buruk dalam lingkungan kode produksi *production*.

Ingat, tidak ada yang salah dengan menggunakan `from package import specific_submodule`! Sebenarnya, ini adalah notasi yang disarankan kecuali modul impor perlu menggunakan submodul dengan nama yang sama dari paket yang berbeda.

6.4.2 Referensi Intra-paket

The submodules often need to refer to each other. For example, the `surround` module might use the `echo` module. In fact, such references are so common that the `import` statement first looks in the containing package before looking in the standard module search path. Thus, the `surround` module can simply use `import echo` or `from echo import echofilter`. If the imported module is not found in the current package (the package of which the current module is a submodule), the `import` statement looks for a top-level module with the given name.

Ketika paket disusun menjadi subpaket (seperti pada paket `sound` pada contoh), Anda dapat menggunakan impor absolut untuk merujuk pada submodul paket saudara kandung. Misalnya, jika modul `sound.filters.vocoder` perlu menggunakan modul `echo` dalam paket `sound.effects`, ia dapat menggunakan `from sound.effects import echo`.

Starting with Python 2.5, in addition to the implicit relative imports described above, you can write explicit relative imports with the `from module import name` form of `import` statement. These explicit relative imports use leading dots to indicate the current and parent packages involved in the relative import. From the `surround` module for example, you might use:

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

Note that both explicit and implicit relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application should always use absolute imports.

6.4.3 Paket di Beberapa Direktori

Paket mendukung satu atribut khusus lagi, `__path__`. Ini diinisialisasi menjadi daftar yang berisi nama direktori yang menyimpan file paket: `__init__.py` sebelum kode dalam file tersebut dieksekusi. Variabel ini dapat dimodifikasi; hal itu memengaruhi pencarian modul dan subpackage di masa depan yang terkandung dalam paket.

Meskipun fitur ini tidak sering dibutuhkan, fitur ini dapat digunakan untuk memperluas rangkaian modul yang ditemukan dalam suatu paket.

Masukan dan Keluaran

Ada beberapa cara untuk mempresentasikan keluaran suatu program; data dapat dicetak dalam bentuk yang dapat dibaca manusia, atau ditulis ke berkas untuk digunakan di masa mendatang. Bab ini akan membahas beberapa kemungkinan.

7.1 Pemformatan Keluaran yang Lebih Menarik

So far we've encountered two ways of writing values: *expression statements* and the `print` statement. (A third way is using the `write()` method of file objects; the standard output file can be referenced as `sys.stdout`. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The string types have some methods that perform useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the `str.format()` method.

The `string` module contains a `Template` class which offers yet another way to substitute values into strings.

One question remains, of course: how do you convert values to strings? Luckily, Python has ways to convert any value to a string: pass it to the `repr()` or `str()` functions.

The `str()` function is meant to return representations of values which are fairly human-readable, while `repr()` is meant to generate representations which can be read by the interpreter (or will force a `SyntaxError` if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, `str()` will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings and floating point numbers, in particular, have two distinct representations.

Beberapa contoh:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

>>> str(1.0/7.0)
'0.142857142857'
>>> repr(1.0/7.0)
'0.14285714285714285'
>>> x = 10 * 3.25
>>> y = 200 * 200
>>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
>>> print s
The value of x is 32.5, and y is 40000...
>>> # The repr() of a string adds string quotes and backslashes:
... hello = 'hello, world\n'
>>> hellos = repr(hello)
>>> print hellos
'hello, world\n'
>>> # The argument to repr() may be any Python object:
... repr((x, y, ('spam', 'eggs'))))
"(32.5, 40000, ('spam', 'eggs'))"

```

Here are two ways to write a table of squares and cubes:

```

>>> for x in range(1, 11):
...     print repr(x).rjust(2), repr(x*x).rjust(3),
...           # Note trailing comma on previous line
...           print repr(x*x*x).rjust(4)
...
1   1   1
2   4   8
3   9  27
4  16  64
5  25 125
6  36 216
7  49 343
8  64 512
9  81 729
10 100 1000

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

```

(Note that in the first example, one space between each column was added by the way `print` works: by default it adds spaces between its arguments.)

This example demonstrates the `str.rjust()` method of string objects, which right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would

be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

Ada metode lain, `str.zfill()`, yang melapisi string numerik di sebelah kiri dengan nol. Itu mengerti tentang tanda plus dan minus:

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

Penggunaan dasar metode `str.format()` terlihat seperti ini:

```
>>> print 'We are the {} who say "{}!".format('knights', 'Ni')
We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets refers to the position of the object passed into the `str.format()` method.

```
>>> print '{0} and {1}'.format('spam', 'eggs')
spam and eggs
>>> print '{1} and {0}'.format('spam', 'eggs')
eggs and spam
```

Jika argumen kata kunci *keyword argument* digunakan dalam metode `str.format()`, nilainya dirujuk dengan menggunakan nama argumen.

```
>>> print 'This {food} is {adjective}'.format(
...     food='spam', adjective='absolutely horrible')
This spam is absolutely horrible.
```

Argumen posisi dan kata kunci dapat dikombinasikan secara bergantian:

```
>>> print 'The story of {0}, {1}, and {other}'.format('Bill', 'Manfred',
...                                                other='Georg')
The story of Bill, Manfred, and Georg.
```

'!s' (apply `str()`) and '!r' (apply `repr()`) can be used to convert the value before it is formatted.

```
>>> import math
>>> print 'The value of PI is approximately {}.'.format(math.pi)
The value of PI is approximately 3.14159265359.
>>> print 'The value of PI is approximately {!r}'.format(math.pi)
The value of PI is approximately 3.141592653589793.
```

An optional ':' and format specifier can follow the field name. This allows greater control over how the value is formatted. The following example rounds Pi to three places after the decimal.

```
>>> import math
>>> print 'The value of PI is approximately {:.3f}'.format(math.pi)
The value of PI is approximately 3.142.
```

Passing an integer after the ':' will cause that field to be a minimum number of characters wide. This is useful for making tables pretty.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
>>> for name, phone in table.items():
...     print '{0:10} ==> {1:10d}'.format(name, phone)
...
Jack          ==>      4098
Dcab          ==>      7678
Sjoerd        ==>      4127
```

Jika Anda memiliki string format yang sangat panjang yang tidak ingin Anda pisahkan, alangkah baiknya jika Anda bisa mereferensikan variabel yang akan diformat berdasarkan nama alih-alih berdasarkan posisi. Ini dapat dilakukan hanya dengan melewati *dict* dan menggunakan tanda kurung siku ' [] ' untuk mengakses kunci dari *dict*

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print ('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
...       'Dcab: {0[Dcab]:d}'.format(table))
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Ini juga bisa dilakukan dengan memberikan tabel sebagai argumen kata kunci *keyword argument* dengan notasi '**'.

```
>>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
>>> print 'Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table)
Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

Ini sangat berguna dalam kombinasi dengan fungsi bawaan `vars()`, yang mengembalikan *dictionary* yang berisi semua variabel lokal.

Untuk ikhtisar lengkap pemformatan string dengan `str.format()`, lihat `formatstrings`.

7.1.1 Pemformatan string lama

Operator `%` juga dapat digunakan untuk pemformatan string. Ini menafsirkan argumen kiri seperti gaya `sprintf()` format string untuk diterapkan pada argumen yang benar, dan mengembalikan string yang dihasilkan dari operasi pemformatan ini. Sebagai contoh:

```
>>> import math
>>> print 'The value of PI is approximately %5.3f.' % math.pi
The value of PI is approximately 3.142.
```

More information can be found in the string-formatting section.

7.2 Membaca dan Menulis Berkas

`open()` returns a file object, and is most commonly used with two arguments: `open(filename, mode)`.

```
>>> f = open('workfile', 'w')
>>> print f
<open file 'workfile', mode 'w' at 80a0960>
```

Argumen pertama adalah string yang berisi nama file. Argumen kedua adalah string lain yang berisi beberapa karakter yang menggambarkan cara berkas akan digunakan. *mode* dapat `'r'` ketika file hanya akan dibaca, `'w'` untuk hanya menulis (berkas yang ada dengan nama yang sama akan dihapus), dan `'a'` membuka berkas untuk ditambahkan; setiap data yang ditulis ke file secara otomatis ditambahkan ke bagian akhir. `'r+'` membuka berkas untuk membaca dan menulis. Argumen *mode* adalah opsional; `'r'` akan diasumsikan jika dihilangkan.

On Windows, 'b' appended to the mode opens the file in binary mode, so there are also modes like 'rb', 'wb', and 'r+b'. Python on Windows makes a distinction between text and binary files; the end-of-line characters in text files are automatically altered slightly when data is read or written. This behind-the-scenes modification to file data is fine for ASCII text files, but it'll corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files. On Unix, it doesn't hurt to append a 'b' to the mode, so you can use it platform-independently for all binary files.

7.2.1 Metode Objek Berkas

Sisa contoh di bagian ini akan menganggap bahwa objek berkas bernama `f` telah dibuat.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string (`""`).

```
>>> f.read()
'This is the entire file.\n'
>>> f.read()
''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `'\n'`, a string containing only a single newline.

```
>>> f.readline()
'This is the first line of the file.\n'
>>> f.readline()
'Second line of the file\n'
>>> f.readline()
''
```

Untuk membaca baris dari file, Anda dapat mengulangi objek berkas. Ini hemat memori, cepat, dan mengarah ke kode sederhana

```
>>> for line in f:
    print line,

This is the first line of the file.
Second line of the file
```

Jika Anda ingin membaca semua baris file dalam daftar *list*, Anda juga dapat menggunakan `list(f)` atau `f.readlines()`.

`f.write(string)` writes the contents of *string* to the file, returning `None`.

```
>>> f.write('This is a test\n')
```

To write something other than a string, it needs to be converted to a string first:

```
>>> value = ('the answer', 42)
>>> s = str(value)
>>> f.write(s)
```

`f.tell()` returns an integer giving the file object's current position in the file, measured in bytes from the beginning of the file. To change the file object's position, use `f.seek(offset, from_what)`. The position is computed from

adding *offset* to a reference point; the reference point is selected by the *from_what* argument. A *from_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
>>> f = open('workfile', 'r+')
>>> f.write('0123456789abcdef')
>>> f.seek(5)          # Go to the 6th byte in the file
>>> f.read(1)
'5'
>>> f.seek(-3, 2)     # Go to the 3rd byte before the end
>>> f.read(1)
'd'
```

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.

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

It is good practice to use the `with` keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent `try-finally` blocks:

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

Objek file memiliki beberapa metode tambahan, seperti `isatty()` dan `truncate()` yang lebih jarang digunakan; bacalah Referensi Pustaka untuk panduan lengkap untuk objek berkas.

7.2.2 Menyimpan data terstruktur dengan json

String dapat dengan mudah ditulis dan dibaca dari file. Angka membutuhkan sedikit usaha, karena metode `read()` hanya mengembalikan string, yang harus diteruskan ke fungsi seperti `int()`, yang mengambil string seperti `'123'` dan mengembalikan nilai numerik 123. Ketika Anda ingin menyimpan tipe data yang lebih kompleks seperti daftar *list* dan *dictionary* bersarang, penguraian dan pembuatan serialisasi dengan tangan menjadi rumit.

Alih-alih membuat pengguna terus-menerus menulis dan men-debug kode untuk menyimpan tipe data yang rumit ke berkas, Python memungkinkan Anda untuk menggunakan format pertukaran data populer yang disebut **JSON (JavaScript Object Notation)**. Modul standar bernama `json` dapat mengambil hierarki data Python, dan mengubahnya menjadi representasi string; proses ini disebut *serializing*. Merekonstruksi data dari representasi string disebut *deserializing*. Antara *serializing* dan *deserializing*, string yang mewakili objek mungkin telah disimpan dalam berkas atau data, atau dikirim melalui koneksi jaringan ke beberapa mesin yang jauh.

Catatan: Format JSON umumnya digunakan oleh aplikasi modern untuk memungkinkan pertukaran data. Banyak programmer sudah terbiasa dengannya, yang membuatnya menjadi pilihan yang baik untuk interoperabilitas.

Jika Anda memiliki objek `x`, Anda dapat melihat representasi string JSON dengan baris kode sederhana:


```
>>> import json
>>> json.dumps([1, 'simple', 'list'])
'[1, "simple", "list"]'
```

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a file. So if `f` is a *file object* opened for writing, we can do this:

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

To decode the object again, if `f` is a *file object* which has been opened for reading:

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

Teknik serialisasi sederhana ini dapat menangani daftar *list* dan *dictionary*, tetapi membuat serialisasi *instance* kelas yang berubah-ubah *arbitrary* di JSON membutuhkan sedikit usaha ekstra. Referensi untuk modul `json` berisi penjelasan tentang ini.

Lihat juga:

Pickle - modul *pickle*

Berlawanan dengan *JSON*, *pickle* adalah protokol yang memungkinkan serialisasi objek Python yang semena-mena *arbitrarily* kompleks. Dengan demikian, ini khusus untuk Python dan tidak dapat digunakan untuk berkomunikasi dengan aplikasi yang ditulis dalam bahasa lain. Ini juga tidak aman secara bawaan: *deserializing pickle* data yang berasal dari sumber yang tidak dipercaya dapat mengeksekusi kode semena-mena *arbitrary*, jika data dibuat oleh penyerang yang terampil.

Kesalahan *errors* dan Pengecualian *exceptions*

Sampai sekarang pesan kesalahan belum lebih dari yang disebutkan, tetapi jika Anda telah mencoba contohnya, Anda mungkin telah melihat beberapa. Ada (setidaknya) dua jenis kesalahan yang dapat dibedakan: *syntax errors* dan *exceptions*.

8.1 Kesalahan Sintaksis

Kesalahan sintaksis, juga dikenal sebagai kesalahan penguraian *parsing*, mungkin merupakan jenis keluhan paling umum yang Anda dapatkan saat Anda masih belajar Python:

```
>>> while True print 'Hello world'
File "<stdin>", line 1
    while True print 'Hello world'
                ^
SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little 'arrow' pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the keyword `print`, since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

8.2 Pengecualian

Bahkan jika suatu pernyataan atau ungkapan secara sintaksis benar, itu dapat menyebabkan kesalahan ketika suatu usaha dilakukan untuk mengeksekusinya. Kesalahan yang terdeteksi selama eksekusi disebut *exceptions* dan tidak fatal tanpa syarat: Anda akan segera belajar cara menanganinya dalam program Python. Namun, sebagian besar pengecualian tidak ditangani oleh program, dan menghasilkan pesan kesalahan seperti yang ditunjukkan di sini:

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: cannot concatenate 'str' and 'int' objects
```

Baris terakhir dari pesan kesalahan menunjukkan apa yang terjadi. Pengecualian ada berbagai jenis yang berbeda, dan tipe dicetak sebagai bagian dari pesan: tipe dalam contoh adalah `ZeroDivisionError`, `NameError` dan `TypeError`. String yang dicetak sebagai jenis pengecualian adalah nama pengecualian bawaan yang terjadi. Ini berlaku untuk semua pengecualian bawaan, tetapi tidak harus sama untuk pengecualian yang dibuat pengguna (meskipun ini adalah konvensi yang bermanfaat). Nama pengecualian standar adalah pengidentifikasi bawaan (bukan kata kunci yang dipesan *reserved keyword*).

Sisa baris menyediakan detail berdasarkan jenis pengecualian dan apa yang menyebabkannya.

Bagian sebelumnya dari pesan kesalahan menunjukkan konteks di mana pengecualian terjadi, dalam bentuk *stack traceback*. Secara umum berisi daftar baris sumber *stack traceback*; namun, ini tidak akan menampilkan baris yang dibaca dari masukan standar.

`bltin-exceptions` memberikan daftar pengecualian bawaan dan artinya.

8.3 Menangani Pengecualian

Dimungkinkan untuk menulis program yang menangani pengecualian yang dipilih. Lihatlah contoh berikut, yang meminta masukan dari pengguna sampai integer yang valid telah dimasukkan, tetapi memungkinkan pengguna untuk menghentikan program (menggunakan `Control-C` atau apa pun yang didukung sistem operasi); perhatikan bahwa gangguan yang dibuat pengguna ditandai dengan munculnya pengecualian `KeyboardInterrupt`.

```
>>> while True:
...     try:
...         x = int(raw_input("Please enter a number: "))
...         break
...     except ValueError:
...         print "Oops! That was no valid number. Try again..."
... 
```

Pernyataan `try` berfungsi sebagai berikut.

- Pertama, *try clause* (pernyataan(-pernyataan) di antara kata kunci `try` dan `except`) dieksekusi.
- Jika tidak ada pengecualian terjadi, *except clause* dilewati dan eksekusi pernyataan `:keyword: try` selesai.
- Jika pengecualian terjadi selama eksekusi klausa *try*, sisa klausa dilewati. Kemudian jika jenisnya cocok dengan pengecualian yang dinamai dengan kata kunci `exception`, klausa *except* dioperasikan, dan kemudian eksekusi berlanjut setelah pernyataan `try`.
- Jika terjadi pengecualian yang tidak cocok dengan pengecualian yang disebutkan dalam klausa kecuali, itu diteruskan ke luar pernyataan `try`; jika tidak ada penanganan yang ditemukan, ini adalah *unhandled exception* dan eksekusi berhenti dengan pesan seperti yang ditunjukkan di atas.

A `try` statement may have more than one `except` clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding `try` clause, not in other handlers of the same `try` statement. An `except` clause may name multiple exceptions as a parenthesized tuple, for example:

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

Note that the parentheses around this tuple are required, because `except ValueError, e:` was the syntax used for what is normally written as `except ValueError as e:` in modern Python (described below). The old syntax is still supported for backwards compatibility. This means `except RuntimeError, TypeError:` is not equivalent to `except (RuntimeError, TypeError):` but to `except RuntimeError as TypeError:` which is not what you want.

Klausula *except* terakhir dapat menghilangkan nama-(nama) pengecualian, untuk berfungsi sebagai *wildcard*. Gunakan ini dengan sangat hati-hati, karena mudah untuk menutupi kesalahan nyata pemrograman dengan cara ini! Ini juga dapat digunakan untuk mencetak pesan kesalahan dan kemudian menimbulkan kembali pengecualian (memungkinkan pemanggil untuk menangani pengecualian juga)

```
import sys

try:
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e:
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError:
    print "Could not convert data to an integer."
except:
    print "Unexpected error:", sys.exc_info()[0]
    raise
```

Pernyataan `try ... :keyword:'except'` memiliki opsi *else clause*, yang, jika ada, harus mengikuti semua klausa *except*. Ini berguna untuk kode yang harus dijalankan jika klausa *try* tidak menimbulkan pengecualian. Sebagai contoh:

```
for arg in sys.argv[1:]:
    try:
        f = open(arg, 'r')
    except IOError:
        print 'cannot open', arg
    else:
        print arg, 'has', len(f.readlines()), 'lines'
        f.close()
```

The use of the `else` clause is better than adding additional code to the `try` clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the `try ... except` statement.

Ketika pengecualian terjadi, itu mungkin memiliki nilai terkait, juga dikenal sebagai *argument* pengecualian. Kehadiran dan jenis argumen tergantung pada jenis pengecualian.

The `except` clause may specify a variable after the exception name (or tuple). The variable is bound to an exception instance with the arguments stored in `instance.args`. For convenience, the exception instance defines `__str__()` so the arguments can be printed directly without having to reference `.args`.

One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
>>> try:
...     raise Exception('spam', 'eggs')
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

... except Exception as inst:
...     print type(inst)      # the exception instance
...     print inst.args       # arguments stored in .args
...     print inst           # __str__ allows args to be printed directly
...     x, y = inst.args
...     print 'x =', x
...     print 'y =', y
...
<type 'exceptions.Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs

```

If an exception has an argument, it is printed as the last part ('detail') of the message for unhandled exceptions.

Penangan pengecualian tidak hanya menangani pengecualian jika mereka muncul segera di klausa *try*, tetapi juga jika mereka terjadi di dalam fungsi yang disebut (bahkan secara tidak langsung) di klausa *try*. Sebagai contoh:

```

>>> def this_fails():
...     x = 1/0
...
>>> try:
...     this_fails()
... except ZeroDivisionError as detail:
...     print 'Handling run-time error:', detail
...
Handling run-time error: integer division or modulo by zero

```

8.4 Memunculkan Pengecualian

Pernyataan *raise* memungkinkan programmer untuk memaksa pengecualian yang ditentukan terjadi. Sebagai contoh:

```

>>> raise NameError('HiThere')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: HiThere

```

The sole argument to *raise* indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from *Exception*).

Jika Anda perlu menentukan apakah pengecualian muncul tetapi tidak bermaksud menanganinya, bentuk yang lebih sederhana dari pernyataan *raise* memungkinkan Anda untuk memunculkan kembali pengecualian:

```

>>> try:
...     raise NameError('HiThere')
... except NameError:
...     print 'An exception flew by!'
...     raise
...
An exception flew by!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
NameError: HiThere

```

8.5 Pengecualian yang Ditentukan Pengguna

Programs may name their own exceptions by creating a new exception class (see [Classes](#) for more about Python classes). Exceptions should typically be derived from the `Exception` class, either directly or indirectly. For example:

```
>>> class MyError(Exception):
...     def __init__(self, value):
...         self.value = value
...     def __str__(self):
...         return repr(self.value)
...
>>> try:
...     raise MyError(2*2)
... except MyError as e:
...     print 'My exception occurred, value:', e.value
...
My exception occurred, value: 4
>>> raise MyError('oops!')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.MyError: 'oops!'
```

In this example, the default `__init__()` of `Exception` has been overridden. The new behavior simply creates the *value* attribute. This replaces the default behavior of creating the *args* attribute.

Kelas pengecualian dapat didefinisikan yang melakukan apa saja yang dapat dilakukan oleh kelas lain, tetapi biasanya tetap sederhana, seringkali hanya menawarkan sejumlah atribut yang memungkinkan informasi tentang kesalahan diekstraksi oleh penanganan sebagai pengecualian. Saat membuat modul yang dapat menimbulkan beberapa kesalahan berbeda, praktik yang umum adalah membuat kelas dasar untuk pengecualian yang ditentukan oleh modul itu, dan mensubkelaskan kelas itu untuk membuat kelas pengecualian khusus untuk kondisi kesalahan yang berbeda:

```
class Error(Exception):
    """Base class for exceptions in this module."""
    pass

class InputError(Error):
    """Exception raised for errors in the input.

    Attributes:
        expr -- input expression in which the error occurred
        msg  -- explanation of the error
    """

    def __init__(self, expr, msg):
        self.expr = expr
        self.msg = msg

class TransitionError(Error):
    """Raised when an operation attempts a state transition that's not
    allowed.

    Attributes:
        prev -- state at beginning of transition
        next -- attempted new state
        msg  -- explanation of why the specific transition is not allowed
    """
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
def __init__(self, prev, next, msg):
    self.prev = prev
    self.next = next
    self.msg = msg
```

Sebagian besar pengecualian didefinisikan dengan nama yang diakhiri dengan "Error", mirip dengan penamaan pengecualian standar.

Banyak modul standar menentukan pengecualian mereka sendiri untuk melaporkan kesalahan yang mungkin terjadi pada fungsi yang mereka tetapkan. Informasi lebih lanjut tentang kelas disajikan dalam bab tut-class.

8.6 Mendefinisikan Tindakan Pembersihan

Pernyataan `try` memiliki klausa opsional lain yang dimaksudkan untuk menentukan tindakan pembersihan yang harus dijalankan dalam semua keadaan. Sebagai contoh:

```
>>> try:
...     raise KeyboardInterrupt
... finally:
...     print 'Goodbye, world!'
...
Goodbye, world!
KeyboardInterrupt
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
```

A *finally clause* is always executed before leaving the `try` statement, whether an exception has occurred or not. When an exception has occurred in the `try` clause and has not been handled by an `except` clause (or it has occurred in an `except` or `else` clause), it is re-raised after the `finally` clause has been executed. The `finally` clause is also executed "on the way out" when any other clause of the `try` statement is left via a `break`, `continue` or `return` statement. A more complicated example (having `except` and `finally` clauses in the same `try` statement works as of Python 2.5):

```
>>> def divide(x, y):
...     try:
...         result = x / y
...     except ZeroDivisionError:
...         print "division by zero!"
...     else:
...         print "result is", result
...     finally:
...         print "executing finally clause"
...
>>> divide(2, 1)
result is 2
executing finally clause
>>> divide(2, 0)
division by zero!
executing finally clause
>>> divide("2", "1")
executing finally clause
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the `finally` clause is executed in any event. The `TypeError` raised by dividing two strings is not handled by the `except` clause and therefore re-raised after the `finally` clause has been executed.

Dalam aplikasi dunia nyata, klausa `finally` berguna untuk melepaskan sumber daya eksternal (seperti berkas atau koneksi jaringan), terlepas dari apakah penggunaan sumber daya tersebut berhasil.

8.7 Tindakan Pembersihan yang Sudah Ditentukan

Beberapa objek mendefinisikan tindakan pembersihan standar yang harus dilakukan ketika objek tidak lagi diperlukan, terlepas dari apakah operasi menggunakan objek berhasil atau gagal. Lihatlah contoh berikut, yang mencoba membuka berkas dan mencetak isinya ke layar.

```
for line in open("myfile.txt"):
    print line,
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The `with` statement allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

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

After the statement is executed, the file `f` is always closed, even if a problem was encountered while processing the lines. Other objects which provide predefined clean-up actions will indicate this in their documentation.

Dibandingkan dengan bahasa pemrograman lain, mekanisme kelas Python menambah kelas dengan minimum sintaksis dan semantik baru. Ini adalah campuran dari mekanisme kelas yang ditemukan dalam C++ dan Modula-3. Kelas Python menyediakan semua fitur standar Pemrograman Berorientasi Objek: mekanisme pewarisan kelas memungkinkan beberapa kelas dasar, kelas turunan dapat menimpa metode apa pun dari kelas dasar atau kelasnya, dan metode dapat memanggil metode kelas dasar dengan nama yang sama. Objek dapat berisi jumlah dan jenis data yang berubah-ubah. Seperti halnya untuk modul, kelas mengambil bagian dari sifat dinamis Python: mereka dibuat pada saat runtime, dan dapat dimodifikasi lebih lanjut setelah pembuatan.

Dalam terminologi C++, biasanya anggota kelas (termasuk anggota data) adalah *public* (kecuali lihat di bawah *Private Variables and Class-local References*), dan semua fungsi anggota adalah *virtual*. Seperti dalam Modula-3, tidak ada singkatan untuk merujuk anggota objek dari metodenya: fungsi metode dideklarasikan dengan argumen pertama eksplisit yang mewakili objek, yang diberikan secara implisit oleh panggilan. Seperti dalam Smalltalk, kelas itu sendiri adalah objek. Ini memberikan semantik untuk mengimpor dan mengganti nama. Tidak seperti C++ dan Modula-3, tipe bawaan dapat digunakan sebagai kelas dasar untuk ekstensi oleh pengguna. Juga, seperti di C++, sebagian besar operator bawaan dengan sintaks khusus (operator aritmatika, *subscripting* dll) dapat didefinisikan ulang untuk *instance* kelas.

(Kurangnya terminologi yang diterima secara universal untuk berbicara tentang kelas, saya akan sesekali menggunakan istilah Smalltalk dan C++. Saya akan menggunakan istilah Modula-3, karena semantik berorientasi objeknya lebih dekat dengan Python daripada C++, tapi saya berharap bahwa beberapa pembaca pernah mendengarnya.)

9.1 Sepatah Kata Tentang Nama dan Objek

Objek memiliki individualitas, dan banyak nama (dalam berbagai lingkup) dapat terikat ke objek yang sama. Ini dikenal sebagai *aliasing* dalam bahasa lain. Ini biasanya tidak dihargai pada pandangan pertama pada Python, dan dapat diabaikan dengan aman ketika berhadapan dengan tipe dasar yang tidak dapat diubah (angka, string, *tuple*). Namun, *aliasing* memiliki efek yang mungkin mengejutkan pada semantik kode Python yang melibatkan objek yang bisa berubah seperti daftar *list*, kamus *dictionary*, dan sebagian besar jenis lainnya. Ini biasanya digunakan untuk kepentingan program, karena alias berperilaku seperti *pointers* dalam beberapa hal. Sebagai contoh, melewatkan objek adalah murah karena hanya sebuah *pointer* dilewatkan oleh implementasi; dan jika suatu fungsi memodifikasi objek yang dilewatkan sebagai argumen, pemanggil akan melihat perubahan --- ini menghilangkan kebutuhan untuk dua mekanisme yang berbeda melewatkan argumen *argument passing* seperti dalam Pascal.

9.2 Lingkup Python dan *Namespaces*

Sebelum memperkenalkan kelas, pertama-tama saya harus memberi tahu Anda tentang aturan ruang lingkup *scope* Python. Definisi kelas memainkan beberapa trik rapi dengan ruang nama *namespaces*, dan Anda perlu tahu bagaimana ruang lingkup dan ruang nama *namespaces* bekerja untuk sepenuhnya memahami apa yang terjadi. Kebetulan, pengetahuan tentang subjek ini berguna untuk programmer Python tingkat lanjut.

Mari kita mulai dengan beberapa definisi.

Sebuah *namespace* adalah pemetaan dari nama ke objek. Sebagian besar ruang nama *namespace* saat ini diimplementasikan sebagai kamus *dictionary* Python, tetapi itu biasanya tidak terlihat dengan cara apa pun (kecuali untuk kinerja), dan itu mungkin berubah di masa depan. Contoh ruang nama *namespace* adalah: himpunan nama bawaan (berisi fungsi seperti `abs()`, dan nama pengecualian bawaan); nama-nama global dalam sebuah modul; dan nama-nama lokal dalam pemanggilan fungsi. Dalam arti himpunan atribut suatu objek juga membentuk *namespace*. Hal penting yang perlu diketahui tentang ruang nama *namespace* adalah sama sekali tidak ada hubungan antara nama dalam ruang nama *namespace* yang berbeda; misalnya, dua modul yang berbeda dapat mendefinisikan fungsi `maximize` tanpa kebingungan --- pengguna modul harus memberikan awalan dengan nama modul.

Ngomong-ngomong, saya menggunakan kata *attribute* untuk nama apa pun yang mengikuti titik --- misalnya, dalam ekspresi `z.real`, `real` adalah atribut dari objek `z`. Sebenarnya, referensi ke nama dalam modul adalah referensi atribut: dalam ekspresi `modname.funcname`, `modname` adalah objek modul dan `funcname` adalah atributnya. Dalam kasus ini akan terjadi pemetaan langsung antara atribut modul dan nama global yang didefinisikan dalam modul: mereka berbagi *namespace* yang sama!¹

Atribut dapat baca-saja *read-only* atau dapat ditulis. Dalam kasus terakhir, pemberian nilai ke atribut dimungkinkan. Atribut modul dapat ditulis: Anda dapat menulis `modname.the_answer = 42`. Atribut yang dapat ditulis juga dapat dihapus dengan pernyataan `del`. Sebagai contoh, `del modname.the_answer` akan menghapus atribut `the_answer` dari objek yang dinamai oleh `modname`.

Namespaces are created at different moments and have different lifetimes. The namespace containing the built-in names is created when the Python interpreter starts up, and is never deleted. The global namespace for a module is created when the module definition is read in; normally, module namespaces also last until the interpreter quits. The statements executed by the top-level invocation of the interpreter, either read from a script file or interactively, are considered part of a module called `__main__`, so they have their own global namespace. (The built-in names actually also live in a module; this is called `__builtin__`.)

Namespace lokal untuk suatu fungsi dibuat ketika fungsi dipanggil, dan dihapus ketika fungsi kembali *returns* atau memunculkan pengecualian yang tidak ditangani dalam fungsi tersebut. (Sebenarnya, melupakan akan menjadi cara yang lebih baik untuk menggambarkan apa yang sebenarnya terjadi.) Tentu saja, pemanggilan rekursif masing-masing memiliki ruang-nama *namespace* lokal mereka sendiri.

Suatu *scope* adalah wilayah tekstual dari program Python di mana *namespace* dapat diakses secara langsung. "Directly accessible" di sini berarti bahwa referensi yang tidak memenuhi syarat untuk suatu nama berusaha menemukan nama tersebut di *namespace*.

Meskipun cakupan *scopes* ditentukan secara statis, mereka digunakan secara dinamis. Setiap saat selama eksekusi, setidaknya ada tiga cakupan bersarang yang ruang nama-nya *namespaces* dapat diakses secara langsung:

- ruang lingkup *scope* terdalam, yang dicari pertama kali, berisi nama-nama lokal
- lingkup *scope* dari setiap fungsi penutup, yang dicari dimulai dengan lingkup penutup terdekat, berisi nama-nama non-lokal, tetapi juga non-global
- lingkup berikutnya *next-to-last* berisi nama global modul saat ini
- ruang lingkup *scope* terluar (dicari terakhir) adalah *namespace* yang mengandung nama bawaan

¹ Kecuali satu hal. Objek modul memiliki atribut baca-saja *read-only* rahasia bernama `__dict__` yang mengembalikan kamus *dictionary* yang digunakan untuk mengimplementasikan *namespace* modul; nama `__dict__` adalah atribut tetapi bukan nama global. Jelas, menggunakan ini melanggar abstraksi implementasi *namespace*, dan harus dibatasi untuk hal-hal seperti *debuggers post-mortem*.

If a name is declared global, then all references and assignments go directly to the middle scope containing the module's global names. Otherwise, all variables found outside of the innermost scope are read-only (an attempt to write to such a variable will simply create a *new* local variable in the innermost scope, leaving the identically named outer variable unchanged).

Biasanya, cakupan lokal merujuk nama lokal dari fungsi (secara tekstual) saat ini. Fungsi luar, lingkup lokal merujuk *namespace* yang sama dengan lingkup global: *namespace* modul. Definisi kelas menempatkan *namespace* lain dalam lingkup lokal.

Penting untuk menyadari bahwa cakupan *scope* ditentukan secara tekstual: ruang lingkup global dari suatu fungsi yang didefinisikan dalam modul adalah ruang nama *namespace* modul itu, tidak peduli dari mana atau oleh apa alias fungsi itu dipanggil. Di sisi lain, pencarian nama sebenarnya dilakukan secara dinamis, pada saat *run time* --- namun, definisi bahasa berkembang menuju resolusi nama statis, pada waktu "compile", jadi jangan mengandalkan resolusi nama dinamis! (Faktanya, variabel lokal sudah ditentukan secara statis.)

A special quirk of Python is that -- if no `global` statement is in effect -- assignments to names always go into the innermost scope. Assignments do not copy data --- they just bind names to objects. The same is true for deletions: the statement `del x` removes the binding of `x` from the namespace referenced by the local scope. In fact, all operations that introduce new names use the local scope: in particular, `import` statements and function definitions bind the module or function name in the local scope. (The `global` statement can be used to indicate that particular variables live in the global scope.)

9.3 Pandangan Pertama tentang Kelas

Kelas memperkenalkan sedikit sintaks baru, tiga tipe objek baru, dan beberapa semantik baru.

9.3.1 Sintaks Definisi Kelas

Bentuk definisi kelas paling sederhana terlihat seperti ini:

```
class ClassName:
    <statement-1>
    .
    .
    .
    <statement-N>
```

Definisi kelas, seperti definisi fungsi (pernyataan `def`) harus dieksekusi sebelum mereka memiliki efek. (Anda dapat menempatkan definisi kelas di cabang dari pernyataan `if`, atau di dalam suatu fungsi.)

Dalam praktiknya, pernyataan di dalam definisi kelas biasanya akan menjadi definisi fungsi, tetapi pernyataan lain diizinkan, dan terkadang berguna --- kami akan kembali ke sini nanti. Definisi fungsi di dalam kelas biasanya memiliki bentuk khusus daftar argumen, didikte oleh konvensi pemanggilan untuk metode --- sekali lagi, ini dijelaskan nanti.

Ketika definisi kelas dimasukkan, *namespace* baru dibuat, dan digunakan sebagai lingkup *scope* lokal --- dengan demikian, semua tugas untuk variabel lokal masuk ke *namespace* baru ini. Secara khusus, definisi fungsi mengikat nama fungsi baru di sini.

Ketika definisi kelas dibiarkan normal (melalui akhir), *class object* dibuat. Ini pada dasarnya adalah pembungkus di sekitar isi *namespace* yang dibuat oleh definisi kelas; kita akan belajar lebih banyak tentang objek kelas di bagian selanjutnya. Lingkup *scope* lokal asli (yang berlaku tepat sebelum definisi kelas dimasukkan) diaktifkan kembali, dan objek kelas terikat di sini dengan nama kelas yang diberikan dalam header definisi kelas (`ClassName` dalam contoh).

9.3.2 Objek Kelas *Class Objects*

Objek kelas mendukung dua jenis operasi: referensi atribut dan instansiasi.

Attribute references menggunakan sintaks standar yang digunakan untuk semua referensi atribut dalam Python: `obj.name`. Nama atribut yang valid adalah semua nama yang ada di *namespace* kelas saat objek kelas dibuat. Jadi, jika definisi kelas tampak seperti ini:

```
class MyClass:
    """A simple example class"""
    i = 12345

    def f(self):
        return 'hello world'
```

kemudian `MyClass.i` dan `MyClass.f` adalah referensi atribut yang valid, masing-masing mengembalikan integer dan objek fungsi. Atribut kelas juga dapat ditetapkan, sehingga Anda dapat mengubah nilai `MyClass.i` oleh penugasan. `__doc__` juga merupakan atribut yang valid, mengembalikan *docstring* milik kelas: "A simple example class".

instantiation kelas menggunakan notasi fungsi. Hanya berpura-pura bahwa objek kelas adalah fungsi tanpa parameter yang mengembalikan instance baru dari kelas. Misalnya (dengan asumsi kelas di atas):

```
x = MyClass()
```

membuat *instance* baru dari kelas dan menetapkan objek ini ke variabel lokal `x`.

Operasi instansiasi ("calling" objek kelas) membuat objek kosong. Banyak kelas suka membuat objek dengan *instance* yang disesuaikan dengan kondisi awal tertentu. Oleh karena itu sebuah kelas dapat mendefinisikan metode khusus bernama `__init__()`, seperti ini:

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

Ketika sebuah kelas mendefinisikan metode `__init__()`, instansiasi kelas secara otomatis memanggil `__init__()` untuk instance kelas yang baru dibuat. Jadi dalam contoh ini, contoh baru yang diinisialisasi dapat diperoleh oleh:

```
x = MyClass()
```

Tentu saja, metode `__init__()` mungkin memiliki argumen untuk fleksibilitas yang lebih besar. Dalam hal itu, argumen yang diberikan kepada operator instansiasi kelas diteruskan ke `__init__()`. Sebagai contoh,

```
>>> class Complex:
...     def __init__(self, realpart, imagpart):
...         self.r = realpart
...         self.i = imagpart
...
>>> x = Complex(3.0, -4.5)
>>> x.r, x.i
(3.0, -4.5)
```

9.3.3 Objek *Instance*

Sekarang apa yang bisa kita lakukan dengan objek instan? Satu-satunya operasi yang dipahami oleh objek instan adalah referensi atribut. Ada dua jenis nama atribut yang valid, atribut data, dan metode.

data attributes sesuai dengan "variabel instan" di Smalltalk, dan "data members" di C++. Atribut data tidak perlu dinyatakan; seperti variabel lokal, mereka muncul ketika mereka pertama kali ditugaskan. Misalnya, jika `x` adalah turunan dari `MyClass` yang dibuat di atas, bagian kode berikut akan mencetak nilai 16, tanpa meninggalkan jejak:

```
x.counter = 1
while x.counter < 10:
    x.counter = x.counter * 2
print x.counter
del x.counter
```

Jenis lain dari referensi atribut instance adalah *method*. Metode adalah fungsi yang "milik" suatu objek. (Dalam Python, istilah metode tidak unik untuk instance kelas: tipe objek lain dapat memiliki metode juga. Misalnya, objek daftar memiliki metode yang disebut *append*, *insert*, *remove*, *sort*, dan sebagainya. Namun, dalam diskusi berikut, kita akan menggunakan istilah metode secara eksklusif untuk mengartikan metode objek *instance* kelas, kecuali dinyatakan secara eksplisit.)

Nama metode yang valid dari objek *instance* bergantung pada kelasnya. Menurut definisi, semua atribut dari kelas yang merupakan objek fungsi menentukan metode yang sesuai dari *instance*-nya. Jadi dalam contoh kita, `x.f` adalah referensi metode yang valid, karena `MyClass.f` adalah fungsi, tetapi `x.i` tidak, karena `MyClass.i` tidak. Tetapi `x.f` bukan hal yang sama dengan `MyClass.f` --- itu adalah *method object*, bukan objek fungsi.

9.3.4 Metode Objek

Biasanya, metode dipanggil tepat setelah itu terikat:

```
x.f()
```

Dalam contoh `MyClass`, ini akan mengembalikan string 'hello world'. Namun, tidak perlu memanggil metode segera: `x.f` adalah metode objek, dan dapat disimpan dan dipanggil di lain waktu. Sebagai contoh:

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

akan terus mencetak hello world hingga akhir waktu.

Apa yang sebenarnya terjadi ketika suatu metode dipanggil? Anda mungkin telah memperhatikan bahwa `x.f()` dipanggil tanpa argumen di atas, meskipun definisi fungsi untuk `f()` menentukan argumen. Apa yang terjadi dengan argumen itu? Tentunya Python memunculkan pengecualian ketika fungsi yang membutuhkan argumen dipanggil tanpa --- bahkan jika argumen tersebut tidak benar-benar digunakan...

Actually, you may have guessed the answer: the special thing about methods is that the object is passed as the first argument of the function. In our example, the call `x.f()` is exactly equivalent to `MyClass.f(x)`. In general, calling a method with a list of *n* arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.

Jika Anda masih tidak mengerti bagaimana metode bekerja, melihat implementasi mungkin dapat mengklarifikasi masalah. Ketika atribut non-data dari sebuah *instance* direferensikan, kelas *instance* tersebut dicari. Jika nama menunjukkan atribut kelas yang valid yang merupakan objek fungsi, objek metode dibuat dengan mengemas (menunjuk *pointers* ke) objek *instance* dan objek fungsi yang baru saja ditemukan bersama dalam objek abstrak: ini adalah objek metode. Ketika objek metode dipanggil dengan daftar argumen, daftar argumen baru dibangun dari objek instance dan daftar argumen, dan objek fungsi dipanggil dengan daftar argumen baru ini.

9.3.5 Variabel Kelas dan *Instance*

Secara umum, variabel *instance* adalah untuk data unik untuk setiap *instance* dan variabel kelas adalah untuk atribut dan metode yang dibagikan oleh semua *instance* kelas:

```
class Dog:

    kind = 'canine'          # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                # shared by all dogs
'canine'
>>> e.kind                # shared by all dogs
'canine'
>>> d.name                # unique to d
'Fido'
>>> e.name                # unique to e
'Buddy'
```

Seperti yang dibahas dalam *Sepatah Kata Tentang Nama dan Objek*, data bersama dapat memiliki efek yang mengejutkan dengan melibatkan objek *mutable* seperti daftar *lists* dan kamus *dictionaries*. Sebagai contoh, daftar *tricks* dalam kode berikut tidak boleh digunakan sebagai variabel kelas karena hanya satu daftar yang akan dibagikan oleh semua *Dog* instance:

```
class Dog:

    tricks = []             # mistaken use of a class variable

    def __init__(self, name):
        self.name = name

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks                # unexpectedly shared by all dogs
['roll over', 'play dead']
```

Desain kelas yang benar harus menggunakan variabel *instance* sebagai gantinya:

```
class Dog:

    def __init__(self, name):
        self.name = name
        self.tricks = []    # creates a new empty list for each dog

    def add_trick(self, trick):
        self.tricks.append(trick)

>>> d = Dog('Fido')
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
>>> e = Dog('Buddy')
>>> d.add_trick('roll over')
>>> e.add_trick('play dead')
>>> d.tricks
['roll over']
>>> e.tricks
['play dead']
```

9.4 Keterangan Acak

Data attributes override method attributes with the same name; to avoid accidental name conflicts, which may cause hard-to-find bugs in large programs, it is wise to use some kind of convention that minimizes the chance of conflicts. Possible conventions include capitalizing method names, prefixing data attribute names with a small unique string (perhaps just an underscore), or using verbs for methods and nouns for data attributes.

Atribut data dapat dirujuk oleh metode dan juga oleh pengguna biasa ("clients") dari suatu objek. Dengan kata lain, kelas tidak dapat digunakan untuk mengimplementasikan tipe data abstrak murni. Faktanya, tidak ada dalam Python yang memungkinkan untuk menegakkan *enforce* data yang disembunyikan --- semuanya didasarkan pada konvensi. (Di sisi lain, implementasi Python, ditulis dalam C, dapat sepenuhnya menyembunyikan detail implementasi dan mengontrol akses ke objek jika perlu; ini dapat digunakan oleh ekstensi ke Python yang ditulis dalam C.)

Klien harus menggunakan atribut data dengan hati-hati --- klien dapat mengacaukan invarian yang dikelola oleh metode dengan menginjak *stamping* atribut data mereka. Perhatikan bahwa klien dapat menambahkan atribut data mereka sendiri ke objek *instance* tanpa memengaruhi validitas metode, asalkan konflik nama dihindari --- sekali lagi, konvensi penamaan dapat menghindarkan dari banyak sakit kepala di sini.

Tidak ada istilah untuk referensi atribut data (atau metode lain!) dari dalam metode. Saya menemukan bahwa ini sebenarnya meningkatkan keterbacaan metode: tidak ada kemungkinan membingungkan variabel lokal dan variabel *instance* ketika melirik *glancing* melalui metode.

Seringkali, argumen pertama dari suatu metode disebut *self*. Ini tidak lebih dari sebuah konvensi: nama *self* sama sekali tidak memiliki arti khusus untuk Python. Perhatikan, bagaimanapun, bahwa dengan tidak mengikuti konvensi kode Anda mungkin kurang dapat dibaca oleh programmer Python lain, dan juga dapat dibayangkan bahwa program *class browser* dapat ditulis yang bergantung pada konvensi semacam itu.

Objek fungsi apa pun yang merupakan atribut kelas menentukan metode untuk *instance* dari kelas itu. Tidak perlu bahwa definisi fungsi tertutup secara teks dalam definisi kelas: menetapkan objek fungsi ke variabel lokal di kelas juga ok. Sebagai contoh:

```
# Function defined outside the class
def f1(self, x, y):
    return min(x, x+y)

class C:
    f = f1

    def g(self):
        return 'hello world'

    h = g
```

Sekarang *f*, *g* dan *h* adalah semua atribut class *C* yang merujuk ke objek-objek fungsi, dan akibatnya semuanya adalah metode instance dari *C* --- *h* sama persis dengan *g*. Perhatikan bahwa praktik ini biasanya hanya membingungkan pembaca program.

Metode dapat memanggil metode lain dengan menggunakan atribut metode dari argumen `self`:

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

    def add(self, x):
        self.data.append(x)

    def addtwice(self, x):
        self.add(x)
        self.add(x)
```

Metode dapat merujuk nama global dengan cara yang sama seperti fungsi biasa. Ruang lingkup *scope* global yang terkait dengan suatu metode adalah modul yang berisi definisinya. (Kelas tidak pernah digunakan sebagai ruang lingkup *scope* global.) Sementara seseorang jarang menemukan alasan yang baik untuk menggunakan data global dalam suatu metode, ada banyak penggunaan sah lingkup global: untuk satu hal, fungsi dan modul yang diimpor ke dalam lingkup global dapat digunakan oleh metode, serta fungsi dan kelas yang didefinisikan di dalamnya. Biasanya, kelas yang berisi metode itu sendiri didefinisikan dalam lingkup global ini, dan di bagian selanjutnya kita akan menemukan beberapa alasan bagus mengapa suatu metode ingin merujuk kelasnya sendiri.

Setiap nilai adalah objek, dan karenanya memiliki *kelas* (juga disebut sebagai *type*). Ini disimpan sebagai `object.__class__`.

9.5 Pewarisan

Tentu saja, fitur bahasa tidak akan layak untuk nama "class" tanpa mendukung pewarisan. Sintaks untuk definisi kelas turunan terlihat seperti ini:

```
class DerivedClassName(BaseClassName):
    <statement-1>
    .
    .
    .
    <statement-N>
```

Nama `BaseClassName` harus didefinisikan dalam lingkup yang berisi definisi kelas turunan. Di tempat nama kelas dasar, ekspresi berubah-ubah *arbitrary* lainnya juga diperbolehkan. Ini bisa berguna, misalnya, ketika kelas dasar didefinisikan dalam modul lain:

```
class DerivedClassName(modname.BaseClassName):
```

Eksekusi definisi kelas turunan menghasilkan sama seperti untuk kelas dasar. Ketika objek kelas dibangun, kelas dasar diingat. Ini digunakan untuk menyelesaikan referensi atribut: jika atribut yang diminta tidak ditemukan di kelas, pencarian dilanjutkan untuk mencari di kelas dasar. Aturan ini diterapkan secara rekursif jika kelas dasar itu sendiri berasal dari beberapa kelas lain.

Tidak ada yang istimewa tentang instance kelas turunan: `DerivedClassName()` membuat instance baru dari kelas. Referensi metode diselesaikan sebagai berikut: atribut kelas yang sesuai dicari, turun rantai kelas dasar jika perlu, dan referensi metode ini valid jika ini menghasilkan objek fungsi.

Kelas turunan dapat menimpa metode kelas dasar mereka. Karena metode tidak memiliki hak khusus ketika memanggil metode lain dari objek yang sama, metode kelas dasar yang memanggil metode lain yang didefinisikan dalam kelas dasar yang sama mungkin akhirnya memanggil metode kelas turunan yang menyimpannya. (Untuk programmer C++: semua metode dalam Python secara efektif *virtual*.)

Menimpa metode dalam kelas turunan mungkin sebenarnya ingin memperluas daripada hanya mengganti metode kelas dasar dengan nama yang sama. Ada cara sederhana untuk memanggil metode kelas dasar secara langsung: cukup panggil `BaseClassName.methodname(self, arguments)`. Ini kadang-kadang berguna untuk klien juga. (Perhatikan bahwa ini hanya berfungsi jika kelas dasar dapat diakses sebagai `BaseClassName` dalam lingkup global.)

Python memiliki dua fungsi bawaan yang bekerja dengan warisan:

- Gunakan `isinstance()` untuk memeriksa jenis instance: `isinstance(obj, int)` akan menjadi `True` hanya jika `obj.__class__` adalah `int` atau beberapa kelas yang diturunkan dari `int`.
- Use `issubclass()` to check class inheritance: `issubclass(bool, int)` is `True` since `bool` is a subclass of `int`. However, `issubclass(unicode, str)` is `False` since `unicode` is not a subclass of `str` (they only share a common ancestor, `basestring`).

9.5.1 Pewarisan Berganda

Python supports a limited form of multiple inheritance as well. A class definition with multiple base classes looks like this:

```
class DerivedClassName(Base1, Base2, Base3):
    <statement-1>
    .
    .
    .
    <statement-N>
```

For old-style classes, the only rule is depth-first, left-to-right. Thus, if an attribute is not found in `DerivedClassName`, it is searched in `Base1`, then (recursively) in the base classes of `Base1`, and only if it is not found there, it is searched in `Base2`, and so on.

(To some people breadth first --- searching `Base2` and `Base3` before the base classes of `Base1` --- looks more natural. However, this would require you to know whether a particular attribute of `Base1` is actually defined in `Base1` or in one of its base classes before you can figure out the consequences of a name conflict with an attribute of `Base2`. The depth-first rule makes no differences between direct and inherited attributes of `Base1`.)

For *new-style classes*, the method resolution order changes dynamically to support cooperative calls to `super()`. This approach is known in some other multiple-inheritance languages as call-next-method and is more powerful than the super call found in single-inheritance languages.

With new-style classes, dynamic ordering is necessary because all cases of multiple inheritance exhibit one or more diamond relationships (where at least one of the parent classes can be accessed through multiple paths from the bottommost class). For example, all new-style classes inherit from `object`, so any case of multiple inheritance provides more than one path to reach `object`. To keep the base classes from being accessed more than once, the dynamic algorithm linearizes the search order in a way that preserves the left-to-right ordering specified in each class, that calls each parent only once, and that is monotonic (meaning that a class can be subclassed without affecting the precedence order of its parents). Taken together, these properties make it possible to design reliable and extensible classes with multiple inheritance. For more detail, see <https://www.python.org/download/releases/2.3/mro/>.

9.6 Private Variables and Class-local References

Variabel instance "Private" yang tidak dapat diakses kecuali dari dalam suatu objek tidak ada dalam Python. Namun, ada konvensi yang diikuti oleh sebagian besar kode Python: nama diawali dengan garis bawah (mis. `__spam`) harus diperlakukan sebagai bagian non-publik dari API (apakah itu fungsi, metode atau anggota data). Ini harus dianggap sebagai detail implementasi dan dapat berubah tanpa pemberitahuan.

Karena ada kasus penggunaan yang valid untuk anggota kelas-pribadi (yaitu untuk menghindari bentrokan nama dengan nama yang ditentukan oleh subkelas), ada dukungan terbatas untuk mekanisme semacam itu, yang disebut *name mangling*. Setiap pengidentifikasi dari bentuk `__spam` (setidaknya dua garis bawah utama, paling banyak satu garis bawah garis bawah) secara teks diganti dengan `__classname__spam`, di mana `classname` adalah nama kelas saat ini dengan garis(-garis) bawah utama dilucuti. *Mangling* ini dilakukan tanpa memperhatikan posisi sintaksis pengidentifikasi, asalkan terjadi dalam definisi kelas.

Name mangling sangat membantu untuk membiarkan subclass menimpa metode tanpa memutus panggilan metode *intra-class*. Sebagai contoh:

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

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

    __update = update    # private copy of original update() method

class MappingSubclass(Mapping):

    def update(self, keys, values):
        # provides new signature for update()
        # but does not break __init__()
        for item in zip(keys, values):
            self.items_list.append(item)
```

Contoh di atas akan berfungsi bahkan jika `MappingSubclass` akan memperkenalkan sebuah pengidentifikasi `__update` karena diganti dengan `__Mapping__update` di kelas `Mapping` dan `__MappingSubclass__update` di kelas `MappingSubclass` masing-masing.

Perhatikan bahwa aturan *mangling* sebagian besar dirancang untuk menghindari kecelakaan; masih dimungkinkan untuk mengakses atau memodifikasi variabel yang dianggap pribadi. Ini bahkan dapat berguna dalam keadaan khusus, seperti di *debugger*.

Notice that code passed to `exec`, `eval()` or `execfile()` does not consider the classname of the invoking class to be the current class; this is similar to the effect of the `global` statement, the effect of which is likewise restricted to code that is byte-compiled together. The same restriction applies to `getattr()`, `setattr()` and `delattr()`, as well as when referencing `__dict__` directly.

9.7 Barang Sisa *Odds and Ends*

Kadang-kadang berguna untuk memiliki tipe data yang mirip dengan "record" Pascal atau "struct" C, menyatukan beberapa item data bernama. Definisi kelas kosong akan menghasilkan hal tersebut dengan baik:

```
class Employee:
    pass

john = Employee() # Create an empty employee record

# Fill the fields of the record
john.name = 'John Doe'
john.dept = 'computer lab'
john.salary = 1000
```

Sepotong kode Python yang mengharapkan tipe data abstrak tertentu sering dapat dilewatkan kelas yang mengemulasi metode tipe data itu sebagai gantinya. Misalnya, jika Anda memiliki fungsi yang memformat beberapa data dari objek file, Anda dapat mendefinisikan kelas dengan metode `read()` dan `readline()` yang mendapatkan data dari buffer string sebagai gantinya, dan meneruskan itu sebagai argumen.

Instance method objects have attributes, too: `m.im_self` is the instance object with the method `m()`, and `m.im_func` is the function object corresponding to the method.

9.8 Exceptions Are Classes Too

User-defined exceptions are identified by classes as well. Using this mechanism it is possible to create extensible hierarchies of exceptions.

There are two new valid (semantic) forms for the `raise` statement:

```
raise Class, instance

raise instance
```

In the first form, `instance` must be an instance of `Class` or of a class derived from it. The second form is a shorthand for:

```
raise instance.__class__, instance
```

A class in an `except` clause is compatible with an exception if it is the same class or a base class thereof (but not the other way around --- an `except` clause listing a derived class is not compatible with a base class). For example, the following code will print B, C, D in that order:

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

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

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

except C:
    print "C"
except B:
    print "B"

```

Note that if the `except` clauses were reversed (with `except B` first), it would have printed B, B, B --- the first matching `except` clause is triggered.

When an error message is printed for an unhandled exception, the exception's class name is printed, then a colon and a space, and finally the instance converted to a string using the built-in function `str()`.

9.9 Iterators

Sekarang Anda mungkin telah memperhatikan bahwa sebagian besar objek penampung *container* dapat dibuat perulangan menggunakan pernyataan `for`:

```

for element in [1, 2, 3]:
    print element
for element in (1, 2, 3):
    print element
for key in {'one':1, 'two':2}:
    print key
for char in "123":
    print char
for line in open("myfile.txt"):
    print line,

```

This style of access is clear, concise, and convenient. The use of iterators pervades and unifies Python. Behind the scenes, the `for` statement calls `iter()` on the container object. The function returns an iterator object that defines the method `next()` which accesses elements in the container one at a time. When there are no more elements, `next()` raises a `StopIteration` exception which tells the `for` loop to terminate. This example shows how it all works:

```

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

```

Having seen the mechanics behind the iterator protocol, it is easy to add iterator behavior to your classes. Define an `__iter__()` method which returns an object with a `next()` method. If the class defines `next()`, then `__iter__()` can just return `self`:

```

class Reverse:
    """Iterator for looping over a sequence backwards."""

```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

def __init__(self, data):
    self.data = data
    self.index = len(data)

def __iter__(self):
    return self

def next(self):
    if self.index == 0:
        raise StopIteration
    self.index = self.index - 1
    return self.data[self.index]

```

```

>>> rev = Reverse('spam')
>>> iter(rev)
<__main__.Reverse object at 0x00A1DB50>
>>> for char in rev:
...     print char
...
m
a
p
s

```

9.10 Pembangkit Generator

: term: *Generators* adalah alat yang sederhana dan berdaya untuk membuat *iterator*. Mereka ditulis seperti fungsi biasa tetapi menggunakan pernyataan `yield` setiap kali mereka ingin mengembalikan data. Setiap kali `next()` dipanggil, *generator* melanjutkan di tempat yang ditinggalkannya (ia mengingat semua nilai data dan pernyataan mana yang terakhir dieksekusi). Contoh menunjukkan bahwa pembangkit *generator* dapat dengan mudah dibuat:

```

def reverse(data):
    for index in range(len(data)-1, -1, -1):
        yield data[index]

```

```

>>> for char in reverse('golf'):
...     print char
...
f
l
o
g

```

Anything that can be done with generators can also be done with class-based iterators as described in the previous section. What makes generators so compact is that the `__iter__()` and `next()` methods are created automatically.

Fitur utama lainnya adalah variabel lokal dan status eksekusi secara otomatis disimpan di antara pemanggilan. Ini membuat fungsi lebih mudah untuk ditulis dan jauh lebih jelas daripada pendekatan menggunakan variabel instan seperti `self.index` dan `self.data`.

Selain pembuatan metode otomatis dan menyimpan status program, ketika pembangkit *generator* berhenti, mereka secara otomatis menimbulkan `StopIteration`. Secara kombinasi, fitur-fitur ini membuatnya mudah untuk membuat *iterator* tanpa lebih dari sekadar menulis fungsi biasa.

9.11 Ekspresi Pembangkit *Generator*

Beberapa pembangkit *generators* sederhana dapat dikodekan secara ringkas sebagai ekspresi menggunakan sintaksis yang mirip dengan pemahaman daftar *list comprehensions* tetapi dengan tanda kurung bukan dengan tanda kurung siku. Ungkapan-ungkapan ini dirancang untuk situasi di mana *generator* digunakan segera oleh fungsi penutup. Ekspresi *generator* lebih kompak tetapi kurang fleksibel daripada definisi *generator* penuh dan cenderung lebih ramah memori daripada pemahaman daftar *list comprehensions* setara.

Contoh:

```
>>> sum(i*i for i in range(10))           # sum of squares
285

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

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

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

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

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


10.1 Antarmuka Sistem Operasi

Modul `os` menyediakan puluhan fungsi untuk berinteraksi dengan sistem operasi:

```
>>> import os
>>> os.getcwd()           # Return the current working directory
'C:\\Python26'
>>> os.chdir('/server/accesslogs') # Change current working directory
>>> os.system('mkdir today')      # Run the command mkdir in the system shell
0
```

Pastikan untuk menggunakan gaya `import os` alih-alih `from os import *`. Ini akan menjaga `os.open()` dari membayangi *shadowing* fungsi bawaan `open()` yang beroperasi jauh berbeda.

Fungsi bawaan `dir()` dan `help()` berguna sebagai alat bantu interaktif untuk bekerja dengan modul besar seperti `os`:

```
>>> import os
>>> dir(os)
<returns a list of all module functions>
>>> help(os)
<returns an extensive manual page created from the module's docstrings>
```

Untuk tugas manajemen berkas dan direktori sehari-hari, modul `shutil` menyediakan antarmuka level yang lebih tinggi yang lebih mudah digunakan:

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

10.2 Berkas *Wildcard*

Modul `glob` menyediakan fungsi untuk membuat daftar berkas dari pencarian *wildcard* di direktori:

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

10.3 Baris Perintah Berargumen

Skrip utilitas umum seringkali perlu memproses argumen baris perintah. Argumen-argumen ini disimpan dalam atribut `argv` dari modul `sys` sebagai daftar. Sebagai contoh, hasil keluaran berikut dari menjalankan `python demo.py one two three` di baris perintah

```
>>> import sys
>>> print sys.argv
['demo.py', 'one', 'two', 'three']
```

The `getopt` module processes `sys.argv` using the conventions of the Unix `getopt()` function. More powerful and flexible command line processing is provided by the `argparse` module.

10.4 Pengalihan Output Kesalahan dan Pengakhiran Program

Modul `sys` juga memiliki atribut untuk `stdin`, `stdout`, dan `stderr`. Yang terakhir berguna untuk mengirimkan peringatan dan pesan kesalahan untuk membuatnya terlihat bahkan ketika `stdout` telah dialihkan:

```
>>> sys.stderr.write('Warning, log file not found starting a new one\n')
Warning, log file not found starting a new one
```

Cara paling langsung untuk mengakhiri skrip adalah dengan menggunakan `sys.exit()`.

10.5 Pencocokan Pola String

Modul `re` menyediakan alat ekspresi reguler untuk pemrosesan string lanjutan. Untuk pencocokan dan manipulasi yang kompleks, ekspresi reguler menawarkan solusi yang ringkas dan dioptimalkan:

```
>>> import re
>>> re.findall(r'\b[a-z]*', 'which foot or hand fell fastest')
['foot', 'fell', 'fastest']
>>> re.sub(r'(\b[a-z]+) \1', r'\1', 'cat in the the hat')
'cat in the hat'
```

Ketika hanya kemampuan sederhana yang diperlukan, metode string lebih disukai karena lebih mudah dibaca dan dilakukan *debug*:

```
>>> 'tea for too'.replace('too', 'two')
'tea for two'
```

10.6 Matematika

Modul `math` memberikan akses ke fungsi-fungsi pustaka C yang mendasari untuk matematika angka pecahan *floating point*:

```
>>> import math
>>> math.cos(math.pi / 4.0)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Modul `random` menyediakan alat untuk membuat pilihan acak:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(xrange(100), 10)    # sampling without replacement
[30, 83, 16, 4, 8, 81, 41, 50, 18, 33]
>>> random.random()                  # random float
0.17970987693706186
>>> random.randrange(6)              # random integer chosen from range(6)
4
```

10.7 Akses internet

There are a number of modules for accessing the internet and processing internet protocols. Two of the simplest are `urllib2` for retrieving data from URLs and `smtplib` for sending mail:

```
>>> import urllib2
>>> for line in urllib2.urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl'):
...     if 'EST' in line or 'EDT' in line: # look for Eastern Time
...         print line

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

>>> import smtplib
>>> server = smtplib.SMTP('localhost')
>>> server.sendmail('soothsayer@example.org', 'jcaesar@example.org',
... """To: jcaesar@example.org
... From: soothsayer@example.org
...
... Beware the Ides of March.
... """)
>>> server.quit()
```

(Perhatikan bahwa contoh kedua membutuhkan server mail yang beroperasi di `localhost`.)

10.8 Tanggal dan Waktu

Modul `datetime` menyediakan kelas untuk memanipulasi tanggal dan waktu dengan cara yang sederhana dan kompleks. Sementara aritmatika tanggal dan waktu didukung, fokus implementasi adalah pada ekstraksi anggota yang efisien untuk pemformatan dan manipulasi keluaran. Modul ini juga mendukung objek yang sadar zona waktu.

```
>>> # dates are easily constructed and formatted
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2003, 12, 2)
>>> now.strftime("%m-%d-%y. %d %b %Y is a %A on the %d day of %B.")
'12-02-03. 02 Dec 2003 is a Tuesday on the 02 day of December.'

>>> # dates support calendar arithmetic
>>> birthday = date(1964, 7, 31)
>>> age = now - birthday
>>> age.days
14368
```

10.9 Kompresi Data

Common data archiving and compression formats are directly supported by modules including: `zlib`, `gzip`, `bz2`, `zipfile` and `tarfile`.

```
>>> import zlib
>>> s = 'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
'witch which has which witches wrist watch'
>>> zlib.crc32(s)
226805979
```

10.10 Pengukuran Kinerja

Beberapa pengguna Python mengembangkan minat yang mendalam untuk mengetahui kinerja relatif dari berbagai pendekatan untuk masalah yang sama. Python menyediakan alat pengukuran yang segera menjawab pertanyaan-pertanyaan itu.

Misalnya, mungkin tergoda untuk menggunakan fitur *tuple packing* dan *unpacking* daripada pendekatan tradisional untuk bertukar argumen. Modul `timeit` dengan cepat menunjukkan keunggulan kinerja secara sederhana:

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

Berbeda dengan granularity tingkat halus `timeit`, modul `profile` dan `pstats` menyediakan alat untuk mengidentifikasi bagian kritis waktu dalam blok kode yang lebih besar.

10.11 Kontrol kualitas

Salah satu pendekatan untuk mengembangkan perangkat lunak berkualitas tinggi adalah dengan menulis tes untuk setiap fungsi yang dikembangkan dan untuk sering menjalankan tes tersebut selama proses pengembangan.

Modul: `mod:doctest` menyediakan alat untuk memindai modul dan memvalidasi tes yang tertanam dalam dokumen program. Konstruksi pengujian sederhana memotong dan menempel panggilan khas beserta hasilnya ke dalam docstring. Ini meningkatkan dokumentasi dengan memberikan contoh kepada pengguna dan memungkinkan modul `doctest` untuk memastikan kode tetap benar untuk dokumentasi:

```
def average(values):
    """Computes the arithmetic mean of a list of numbers.

    >>> print average([20, 30, 70])
    40.0
    """
    return sum(values, 0.0) / len(values)

import doctest
doctest.testmod()    # automatically validate the embedded tests
```

Modul `unittest` tidak semudah modul `doctest`, tetapi memungkinkan serangkaian tes yang lebih komprehensif untuk dipertahankan dalam file terpisah:

```
import unittest

class TestStatisticalFunctions(unittest.TestCase):

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

unittest.main()    # Calling from the command line invokes all tests
```

10.12 Dilengkapi Baterai

Python memiliki filosofi "dilengkapi baterai". Ini paling baik dilihat melalui kemampuan yang canggih dan kuat *robust* dengan dukungan paket-paket yang lebih besar. Sebagai contoh:

- The `xmlrpclib` and `SimpleXMLRPCServer` modules make implementing remote procedure calls into an almost trivial task. Despite the modules names, no direct knowledge or handling of XML is needed.
- The `email` package is a library for managing email messages, including MIME and other RFC 2822-based message documents. Unlike `smtplib` and `poplib` which actually send and receive messages, the `email` package has a complete toolset for building or decoding complex message structures (including attachments) and for implementing internet encoding and header protocols.

- The `xml.dom` and `xml.sax` packages provide robust support for parsing this popular data interchange format. Likewise, the `csv` module supports direct reads and writes in a common database format. Together, these modules and packages greatly simplify data interchange between Python applications and other tools.
- Internasionalisasi didukung oleh sejumlah modul termasuk paket `gettext`, `locale`, dan `codecs`.

Tur Singkat Pustaka Standar --- Bagian II

Tur kedua ini mencakup modul lanjutan yang mendukung kebutuhan pemrograman profesional. Modul-modul ini jarang terjadi dalam skrip kecil.

11.1 Pemformatan Output

The `repr` module provides a version of `repr()` customized for abbreviated displays of large or deeply nested containers:

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

Modul `pprint` menawarkan kontrol yang lebih canggih atas pencetakan objek bawaan dan yang ditentukan pengguna dengan cara yang dapat dibaca oleh *interpreter*. Ketika hasilnya lebih dari satu baris, "pretty printer" menambahkan jeda baris dan indentasi untuk lebih jelas mengungkapkan struktur data:

```
>>> import pprint
>>> t = [[['black', 'cyan'], 'white', ['green', 'red']], [['magenta',
...     'yellow'], 'blue']]
...
>>> pprint.pprint(t, width=30)
[[['black', 'cyan'],
   'white',
   ['green', 'red']],
 [['magenta', 'yellow'],
  'blue']]
```

Modul `textwrap` memformat paragraf teks agar sesuai dengan lebar layar yang diberikan:

```
>>> import textwrap
>>> doc = """The wrap() method is just like fill() except that it returns
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
...
>>> print textwrap.fill(doc, width=40)
The wrap() method is just like fill()
except that it returns a list of strings
instead of one big string with newlines
to separate the wrapped lines.
```

Modul `locale` mengakses basis data format data kultur khusus. Atribut pengelompokan fungsi format lokal *locale* menyediakan cara langsung memformat angka dengan pemisah grup:

```
>>> import locale
>>> locale.setlocale(locale.LC_ALL, 'English_United States.1252')
'English_United States.1252'
>>> conv = locale.localeconv()           # get a mapping of conventions
>>> x = 1234567.8
>>> locale.format("%d", x, grouping=True)
'1,234,567'
>>> locale.format_string("%s%.*f", (conv['currency_symbol'],
...                               conv['frac_digits'], x), grouping=True)
'$1,234,567.80'
```

11.2 Templating

Modul `string` menyertakan kelas serbaguna `Template` dengan sintaks yang disederhanakan yang cocok untuk diedit oleh pengguna. Ini memungkinkan pengguna untuk menyesuaikan aplikasi mereka tanpa harus mengubah aplikasi.

Format ini menggunakan nama penampung yang dibentuk oleh `$` dengan pengidentifikasi Python yang valid (karakter alfanumerik dan garis bawah). Mengitari *placeholder* dengan kurung kurawal memungkinkan diikuti oleh lebih banyak huruf alfanumerik tanpa spasi. Menulis `$$` menciptakan satu yang terpisah `$`

```
>>> from string import Template
>>> t = Template('${village}folk send $$10 to $cause.')
>>> t.substitute(village='Nottingham', cause='the ditch fund')
'Nottinghamfolk send $10 to the ditch fund.'
```

Metode `substitute()` memunculkan `KeyError` saat *placeholder* tidak disertakan dalam *dictionary* atau argumen kata kunci *keyword argument*. Untuk aplikasi gaya gabungan-surat *mail-merge*, data yang diberikan pengguna mungkin tidak lengkap dan metode `safe_substitute()` mungkin lebih tepat --- itu akan membuat *placeholder* tidak berubah jika data hilang

```
>>> t = Template('Return the $item to $owner.')
>>> d = dict(item='unladen swallow')
>>> t.substitute(d)
Traceback (most recent call last):
...
KeyError: 'owner'
>>> t.safe_substitute(d)
'Return the unladen swallow to $owner.'
```

Subkelas templat dapat menentukan pembatas khusus. Misalnya, utilitas penggantian nama setumpuk *batch* untuk *browser* foto dapat memilih untuk menggunakan tanda persen untuk penampung seperti tanggal saat ini, nomor urut gambar, atau format berkas:


```
>>> import time, os.path
>>> photofiles = ['img_1074.jpg', 'img_1076.jpg', 'img_1077.jpg']
>>> class BatchRename(Template):
...     delimiter = '%'
>>> fmt = raw_input('Enter rename style (%d-date %n-seqnum %f-format): ')
Enter rename style (%d-date %n-seqnum %f-format): Ashley_%n%f

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

img_1074.jpg --> Ashley_0.jpg
img_1076.jpg --> Ashley_1.jpg
img_1077.jpg --> Ashley_2.jpg
```

Aplikasi lain untuk *templating* adalah memisahkan logika program dari detail berbagai format output. Ini memungkinkan untuk mengganti templat khusus untuk file XML, laporan teks biasa, dan laporan web HTML.

11.3 Bekerja dengan Tata Letak Rekam Data Biner

Modul `struct` menyediakan `pack()` dan `unpack()` berfungsi untuk bekerja dengan format rekaman biner yang memiliki panjang variabel. Contoh berikut menunjukkan bagaimana cara loop tajuk *header* informasi dalam berkas ZIP tanpa menggunakan modul `zipfile`. Kode paket "H" dan "I" masing-masing mewakili dua dan empat byte angka yang tidak bertanda *unsigned*. "<" Menunjukkan bahwa mereka adalah ukuran standar dan dalam urutan byte *little-endian*:

```
import struct

data = open('myfile.zip', 'rb').read()
start = 0
for i in range(3):                                # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIHH', data[start:start+16])
    crc32, comp_size, uncomp_size, filenamesize, extra_size = fields

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

    start += extra_size + comp_size                # skip to the next header
```

11.4 Multi-threading

Threading adalah teknik untuk memisahkan tugas yang tidak tergantung secara berurutan. Utas *threads* dapat digunakan untuk meningkatkan responsif aplikasi yang menerima masukan pengguna saat tugas lain beroperasi di latar belakang. Kasus penggunaan terkait menjalankan I/O secara paralel dengan perhitungan di utas *thread* lainnya.

Kode berikut menunjukkan bagaimana tingkat tinggi modul:mod:threading dapat menjalankan tugas di latar belakang sementara program utama terus beroperasi:

```
import threading, zipfile

class AsyncZip(threading.Thread):
    def __init__(self, infile, outfile):
        threading.Thread.__init__(self)
        self.infile = infile
        self.outfile = outfile

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

background = AsyncZip('mydata.txt', 'myarchive.zip')
background.start()
print 'The main program continues to run in foreground.'

background.join()    # Wait for the background task to finish
print 'Main program waited until background was done.'
```

Tantangan utama aplikasi multi-utas *multi-threaded* adalah mengoordinasikan utas *thread* yang berbagi data atau sumber daya lainnya. Untuk itu, modul *threading* menyediakan sejumlah primitif sinkronisasi termasuk kunci *locks*, peristiwa *events*, variabel kondisi, dan semafor.

While those tools are powerful, minor design errors can result in problems that are difficult to reproduce. So, the preferred approach to task coordination is to concentrate all access to a resource in a single thread and then use the `Queue` module to feed that thread with requests from other threads. Applications using `Queue.Queue` objects for inter-thread communication and coordination are easier to design, more readable, and more reliable.

11.5 Pencatatan

Modul `logging` menawarkan sistem pencatatan *logging* yang lengkap dan fleksibel. Paling sederhana, catatan *log* pesan dikirim ke berkas atau ke `sys.stderr`:

```
import logging
logging.debug('Debugging information')
logging.info('Informational message')
logging.warning('Warning:config file %s not found', 'server.conf')
logging.error('Error occurred')
logging.critical('Critical error -- shutting down')
```

Ini menghasilkan keluaran berikut:

```
WARNING:root:Warning:config file server.conf not found
ERROR:root:Error occurred
CRITICAL:root:Critical error -- shutting down
```

Secara bawaan, pesan informasi dan *debugging* ditutupi *suppressed* dan keluaran dikirim ke standar kesalahan. Opsi keluaran lainnya termasuk merutekan pesan melalui email, datagram, soket, atau ke Server HTTP. Filter baru dapat memilih rute berbeda berdasarkan prioritas pesan: DEBUG, INFO, WARNING, ERROR, dan CRITICAL.

Sistem pencatatan dapat dikonfigurasi secara langsung dari Python atau dapat dimuat dari berkas konfigurasi yang dapat diedit pengguna untuk pencatatan yang disesuaikan tanpa mengubah aplikasi.

11.6 Referensi yang Lemah

Python melakukan manajemen memori otomatis (penghitungan referensi untuk sebagian besar objek dan garbage collection untuk menghilangkan siklus). Memori dibebaskan tidak lama setelah referensi terakhir untuk itu telah dihilangkan.

Pendekatan ini berfungsi dengan baik untuk sebagian besar aplikasi tetapi kadang-kadang ada kebutuhan untuk melacak objek hanya selama mereka digunakan oleh sesuatu yang lain. Sayangnya, hanya melacak mereka membuat referensi yang membuatnya permanen. Modul `weakref` menyediakan alat untuk melacak objek tanpa membuat referensi. Ketika objek tidak lagi diperlukan, itu secara otomatis dihapus dari tabel *weakref* dan panggilan balik *callback* dipicu untuk *weakref*. Aplikasi yang umum termasuk *caching* objek yang mahal untuk dibuat:

```
>>> import weakref, gc
>>> class A:
...     def __init__(self, value):
...         self.value = value
...     def __repr__(self):
...         return str(self.value)
...
>>> a = A(10)                                # create a reference
>>> d = weakref.WeakValueDictionary()
>>> d['primary'] = a                           # does not create a reference
>>> d['primary']                               # fetch the object if it is still alive
10
>>> del a                                     # remove the one reference
>>> gc.collect()                             # run garbage collection right away
0
>>> d['primary']                             # entry was automatically removed
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    d['primary']                               # entry was automatically removed
  File "C:/python26/lib/weakref.py", line 46, in __getitem__
    o = self.data[key]()
KeyError: 'primary'
```

11.7 Alat untuk Bekerja dengan Daftar *Lists*

Banyak kebutuhan struktur data dapat dipenuhi dengan tipe daftar *list* bawaan. Namun, kadang-kadang ada kebutuhan untuk implementasi alternatif dengan mengorbankan kinerja yang menurun.

Modul `array` menyediakan objek `array()` yang seperti daftar *list* dimana hanya menyimpan data homogen dan menyimpannya dengan lebih kompak. Contoh berikut menunjukkan array angka yang disimpan sebagai dua byte angka biner yang tidak ditandai (kode tipe "H") daripada 16 byte per entri biasa untuk daftar *list* reguler objek `int` Python:

```
>>> from array import array
>>> a = array('H', [4000, 10, 700, 22222])
>>> sum(a)
26932
>>> a[1:3]
array('H', [10, 700])
```

Modul `collections` menyediakan objek `deque()` yang seperti daftar *list* dengan tambahan yang lebih cepat dan muncul dari sisi kiri tetapi pencarian yang lebih lambat di tengah. Objek-objek ini sangat cocok untuk mengimplementasikan antrian dan pencarian pohon pertama yang luas *breadth first tree searches*:

```
>>> from collections import deque
>>> d = deque(["task1", "task2", "task3"])
>>> d.append("task4")
>>> print "Handling", d.popleft()
Handling task1
```

```
unsearched = deque([starting_node])
def breadth_first_search(unsearched):
    node = unsearched.popleft()
    for m in gen_moves(node):
        if is_goal(m):
            return m
    unsearched.append(m)
```

Selain implementasi daftar *list* alternatif, di pustaka juga menawarkan alat-alat lain seperti modul `bisect` dengan fungsi untuk memanipulasi daftar *list* yang diurutkan:

```
>>> import bisect
>>> scores = [(100, 'perl'), (200, 'tcl'), (400, 'lua'), (500, 'python')]
>>> bisect.insort(scores, (300, 'ruby'))
>>> scores
[(100, 'perl'), (200, 'tcl'), (300, 'ruby'), (400, 'lua'), (500, 'python')]
```

Modul `heapq` menyediakan fungsi untuk mengimplementasikan *heaps* berdasarkan daftar *list* reguler. Entri dengan nilai terendah selalu disimpan di posisi nol. Ini berguna untuk aplikasi yang berulang kali mengakses elemen terkecil tetapi tidak ingin mengoperasikan daftar pengurutan *list* secara penuh:

```
>>> from heapq import heapify, heappop, heappush
>>> data = [1, 3, 5, 7, 9, 2, 4, 6, 8, 0]
>>> heapify(data) # rearrange the list into heap order
>>> heappush(data, -5) # add a new entry
>>> [heappop(data) for i in range(3)] # fetch the three smallest entries
[-5, 0, 1]
```

11.8 Aritmatika Pecahan *Floating Point* Desimal

Modul `decimal` menawarkan `Decimal` tipe data untuk aritmatika pecahan desimal. Dibandingkan dengan implementasi bawaan `float` dari pecahan *floating point* biner, kelas ini sangat membantu

- aplikasi keuangan dan penggunaan lainnya yang membutuhkan representasi desimal yang tepat,
- kontrol atas presisi,
- kontrol atas pembulatan untuk memenuhi persyaratan sah *legal* atau peraturan,
- pelacakan tempat desimal yang signifikan, atau
- aplikasi tempat pengguna mengharapkan hasil agar sesuai dengan perhitungan yang dilakukan dengan tangan.

Misalnya, menghitung pajak 5% pada biaya telepon 70 sen memberikan hasil berbeda dalam pecahan *floating point* desimal dan pecahan *floating point* biner. Perbedaan ini menjadi signifikan jika hasilnya dibulatkan ke sen terdekat:

```
>>> from decimal import *
>>> x = Decimal('0.70') * Decimal('1.05')
>>> x
Decimal('0.7350')
>>> x.quantize(Decimal('0.01')) # round to nearest cent
Decimal('0.74')
>>> round(.70 * 1.05, 2)         # same calculation with floats
0.73
```

Hasil `Decimal` menjaga akhiran *trailing nol*, secara otomatis menyimpulkan empat tempat signifikansi dari *multiplicands* dengan dua tempat signifikansi. Desimal mereproduksi matematika seperti yang dilakukan dengan tangan dan menghindari masalah yang dapat muncul ketika pecahan *floating point* biner tidak dapat secara tepat mewakili jumlah desimal.

Representasi yang tepat memungkinkan kelas `Decimal` untuk melakukan perhitungan modulo dan tes persamaan yang tidak cocok untuk angka pecahan *floating point* biner:

```
>>> Decimal('1.00') % Decimal('.10')
Decimal('0.00')
>>> 1.00 % 0.10
0.09999999999999995

>>> sum([Decimal('0.1')]*10) == Decimal('1.0')
True
>>> sum([0.1]*10) == 1.0
False
```

Modul `decimal` menyediakan aritmatika dengan ketelitian sebanyak yang dibutuhkan:

```
>>> getcontext().prec = 36
>>> Decimal(1) / Decimal(7)
Decimal('0.142857142857142857142857142857142857')
```

Lalu apa sekarang?

Membaca tutorial ini mungkin telah menambah minat Anda dalam menggunakan Python --- Anda harus bersemangat untuk menerapkan Python untuk menyelesaikan masalah dunia nyata Anda. Ke mana Anda harus pergi untuk belajar lebih banyak?

Tutorial ini adalah bagian dari kumpulan dokumentasi Python. Beberapa dokumen lain di dalam kumpulan adalah:

- `library-index`:

Anda harus menelusuri manual ini, yang memberikan bahan referensi lengkap (meskipun singkat) tentang tipe, fungsi, dan modul di pustaka standar. Distribusi Python standar termasuk *banyak* kode tambahan. Ada modul untuk membaca kotak surat Unix, mengambil dokumen melalui HTTP, menghasilkan angka acak, mengurai opsi baris perintah, menulis program CGI, mengompres data, dan banyak tugas lainnya. Membaca sekilas Referensi Pustaka akan memberi Anda gagasan tentang apa yang tersedia.

- `install-index` explains how to install external modules written by other Python users.
- `reference-index`: Penjelasan terperinci tentang sintaksis dan semantik Python. Ini bacaan yang berat, tetapi berguna sebagai panduan lengkap untuk bahasa itu sendiri.

Lebih banyak sumber tentang Python:

- <https://www.python.org>: Situs web Python utama. Ini berisi kode, dokumentasi, dan petunjuk ke halaman terkait Python di seluruh Web. Situs web ini diduplikasi di berbagai tempat di seluruh dunia, seperti Eropa, Jepang, dan Australia; situs duplikat mungkin lebih cepat dari situs utama, tergantung pada lokasi geografis Anda.
- <https://docs.python.org>: Akses cepat ke dokumentasi Python.
- <https://pypi.org>: The Python Package Index, previously also nicknamed the Cheese Shop, is an index of user-created Python modules that are available for download. Once you begin releasing code, you can register it here so that others can find it.
- <https://code.activestate.com/recipes/langs/python/>: Python Cookbook adalah kumpulan contoh kode yang cukup besar, modul yang lebih besar, dan skrip yang bermanfaat. Kontribusi yang sangat penting dikumpulkan dalam sebuah buku yang juga berjudul Python Cookbook (O'Reilly & Associates, ISBN 0-596-00797-3.)

For Python-related questions and problem reports, you can post to the newsgroup `comp.lang.python`, or send them to the mailing list at python-list@python.org. The newsgroup and mailing list are gatewayed, so messages posted to one will automatically be forwarded to the other. There are around 120 postings a day (with peaks up to several hundred),

asking (and answering) questions, suggesting new features, and announcing new modules. Before posting, be sure to check the list of Frequently Asked Questions (also called the FAQ). Mailing list archives are available at <https://mail.python.org/pipermail/>. The FAQ answers many of the questions that come up again and again, and may already contain the solution for your problem.

Pengeditan Input Interaktif dan Penggantian Riwayat

Some versions of the Python interpreter support editing of the current input line and history substitution, similar to facilities found in the Korn shell and the GNU Bash shell. This is implemented using the [GNU Readline](#) library, which supports Emacs-style and vi-style editing. This library has its own documentation which I won't duplicate here; however, the basics are easily explained. The interactive editing and history described here are optionally available in the Unix and Cygwin versions of the interpreter.

This chapter does *not* document the editing facilities of Mark Hammond's PythonWin package or the Tk-based environment, IDLE, distributed with Python. The command line history recall which operates within DOS boxes on NT and some other DOS and Windows flavors is yet another beast.

13.1 Line Editing

If supported, input line editing is active whenever the interpreter prints a primary or secondary prompt. The current line can be edited using the conventional Emacs control characters. The most important of these are: C-A (Control-A) moves the cursor to the beginning of the line, C-E to the end, C-B moves it one position to the left, C-F to the right. Backspace erases the character to the left of the cursor, C-D the character to its right. C-K kills (erases) the rest of the line to the right of the cursor, C-Y yanks back the last killed string. C-underscore undoes the last change you made; it can be repeated for cumulative effect.

13.2 History Substitution

History substitution works as follows. All non-empty input lines issued are saved in a history buffer, and when a new prompt is given you are positioned on a new line at the bottom of this buffer. C-P moves one line up (back) in the history buffer, C-N moves one down. Any line in the history buffer can be edited; an asterisk appears in front of the prompt to mark a line as modified. Pressing the Return key passes the current line to the interpreter. C-R starts an incremental reverse search; C-S starts a forward search.

13.3 Key Bindings

The key bindings and some other parameters of the Readline library can be customized by placing commands in an initialization file called `~/.inputrc`. Key bindings have the form

```
key-name: function-name
```

or

```
"string": function-name
```

and options can be set with

```
set option-name value
```

For example:

```
# I prefer vi-style editing:
set editing-mode vi

# Edit using a single line:
set horizontal-scroll-mode On

# Rebind some keys:
Meta-h: backward-kill-word
"\C-u": universal-argument
"\C-x\C-r": re-read-init-file
```

Note that the default binding for Tab in Python is to insert a Tab character instead of Readline's default filename completion function. If you insist, you can override this by putting

```
Tab: complete
```

in your `~/.inputrc`. (Of course, this makes it harder to type indented continuation lines if you're accustomed to using Tab for that purpose.)

Automatic completion of variable and module names is optionally available. To enable it in the interpreter's interactive mode, add the following to your startup file:¹

```
import rlcompleter, readline
readline.parse_and_bind('tab: complete')
```

This binds the Tab key to the completion function, so hitting the Tab key twice suggests completions; it looks at Python statement names, the current local variables, and the available module names. For dotted expressions such as `string.a`, it will evaluate the expression up to the final `'.'` and then suggest completions from the attributes of the resulting object. Note that this may execute application-defined code if an object with a `__getattr__()` method is part of the expression.

A more capable startup file might look like this example. Note that this deletes the names it creates once they are no longer needed; this is done since the startup file is executed in the same namespace as the interactive commands, and removing the names avoids creating side effects in the interactive environment. You may find it convenient to keep some of the imported modules, such as `os`, which turn out to be needed in most sessions with the interpreter.

¹ Python will execute the contents of a file identified by the `PYTHONSTARTUP` environment variable when you start an interactive interpreter. To customize Python even for non-interactive mode, see [Modul Ubahsuaian](#).

```
# Add auto-completion and a stored history file of commands to your Python
# interactive interpreter. Requires Python 2.0+, readline. Autocomplete is
# bound to the Esc key by default (you can change it - see readline docs).
#
# Store the file in ~/.pystartup, and set an environment variable to point
# to it: "export PYTHONSTARTUP=~/.pystartup" in bash.

import atexit
import os
import readline
import rlcompleter

historyPath = os.path.expanduser("~/pyhistory")

def save_history(historyPath=historyPath):
    import readline
    readline.write_history_file(historyPath)

if os.path.exists(historyPath):
    readline.read_history_file(historyPath)

atexit.register(save_history)
del os, atexit, readline, rlcompleter, save_history, historyPath
```

13.4 Alternatif untuk Interpreter Interaktif

Fasilitas ini merupakan kemajuan yang sangat besar dibandingkan dengan interpreter versi sebelumnya; namun, ada beberapa keinginan yang tersisa: Akan lebih baik jika indentasi yang tepat disarankan pada baris lanjutan (parser tahu jika token indentasi diperlukan berikutnya). Mekanisme pelengkapan mungkin menggunakan tabel simbol interpreter. Perintah untuk memeriksa (atau bahkan menyarankan) tanda kurung, tanda kutip, dll., juga berguna.

Salah satu alternatif interpreter interaktif canggih yang telah ada selama beberapa waktu adalah [IPython](#), yang menampikan pelengkapan tab, eksplorasi objek, dan manajemen riwayat lanjut. Itu juga dapat sepenuhnya disesuaikan dan tertanam ke dalam aplikasi lain. Lingkungan interaktif ditingkatkan serupa lainnya adalah [bpython](#).

Aritmatika Pecahan *Floating Point*: Masalah dan Keterbatasan

Angka pecahan *floating point* diwakili dalam perangkat keras komputer sebagai pecahan basis 2 (biner). Misalnya, pecahan desimal:

0.125

memiliki nilai $1/10 + 2/100 + 5/1000$, dan dengan cara yang sama pecahan biner

0.001

memiliki nilai $0/2 + 0/4 + 1/8$. Dua pecahan ini memiliki nilai yang identik, satu-satunya perbedaan nyata adalah bahwa yang pertama ditulis dalam notasi fraksi basis 10, dan yang kedua dalam basis 2.

Sayangnya, sebagian besar pecahan desimal tidak dapat direpresentasikan persis dengan pecahan biner. Konsekuensinya adalah bahwa, secara umum, angka pecahan *floating-point* desimal yang Anda masukkan hanya didekati oleh angka-angka pecahan *floating-point* biner yang sebenarnya disimpan dalam mesin.

Masalahnya lebih mudah dipahami pada awalnya di basis 10. Pertimbangkan fraksi $1/3$. Anda dapat memperkirakannya sebagai pecahan basis 10:

0.3

atau, lebih baik,

0.33

atau, lebih baik,

0.333

dan seterusnya. Tidak peduli berapa banyak digit yang Anda ingin tulis, hasilnya tidak akan pernah benar-benar $1/3$, tetapi akan menjadi perkiraan yang semakin baik dari $1/3$.

Dengan cara yang sama, tidak peduli berapa banyak digit basis 2 yang ingin Anda gunakan, nilai desimal 0.1 tidak dapat direpresentasikan persis sebagai fraksi basis 2. Dalam basis 2, $1/10$ adalah pecahan berulang yang tak terhingga

```
0.000110011001100110011001100110011001100110011...
```

Stop at any finite number of bits, and you get an approximation.

On a typical machine running Python, there are 53 bits of precision available for a Python float, so the value stored internally when you enter the decimal number `0.1` is the binary fraction

```
0.000110011001100110011001100110011001100110011010
```

which is close to, but not exactly equal to, $1/10$.

It's easy to forget that the stored value is an approximation to the original decimal fraction, because of the way that floats are displayed at the interpreter prompt. Python only prints a decimal approximation to the true decimal value of the binary approximation stored by the machine. If Python were to print the true decimal value of the binary approximation stored for 0.1, it would have to display

```
>>> 0.1
0.1000000000000000055511151231257827021181583404541015625
```

Itu lebih banyak angka daripada yang dianggap berguna oleh kebanyakan orang, jadi Python menjaga jumlah angka tetap dapat dikelola dengan menampilkan nilai bulat sebagai gantinya

```
>>> 0.1
0.1
```

It's important to realize that this is, in a real sense, an illusion: the value in the machine is not exactly $1/10$, you're simply rounding the *display* of the true machine value. This fact becomes apparent as soon as you try to do arithmetic with these values

```
>>> 0.1 + 0.2
0.30000000000000004
```

Perhatikan bahwa ini adalah sifat dasar dari pecahan *floating-point* biner: ini bukan bug di Python, dan ini juga bukan bug dalam kode Anda. Anda akan melihat hal yang sama dalam semua bahasa yang mendukung aritmatika pecahan *floating-point* perangkat keras Anda (meskipun beberapa bahasa mungkin tidak *display* perbedaan secara default, atau dalam semua mode keluaran).

Other surprises follow from this one. For example, if you try to round the value 2.675 to two decimal places, you get this

```
>>> round(2.675, 2)
2.67
```

The documentation for the built-in `round()` function says that it rounds to the nearest value, rounding ties away from zero. Since the decimal fraction 2.675 is exactly halfway between 2.67 and 2.68, you might expect the result here to be (a binary approximation to) 2.68. It's not, because when the decimal string `2.675` is converted to a binary floating-point number, it's again replaced with a binary approximation, whose exact value is

2.6749999999999982236431605997495353221893310546875

Since this approximation is slightly closer to 2.67 than to 2.68, it's rounded down.

If you're in a situation where you care which way your decimal halfway-cases are rounded, you should consider using the `decimal` module. Incidentally, the `decimal` module also provides a nice way to "see" the exact value that's stored in any particular Python float

```
>>> from decimal import Decimal
>>> Decimal(2.675)
Decimal('2.67499999999999982236431605997495353221893310546875')
```

Another consequence is that since 0.1 is not exactly 1/10, summing ten values of 0.1 may not yield exactly 1.0, either:

```
>>> sum = 0.0
>>> for i in range(10):
...     sum += 0.1
...
>>> sum
0.9999999999999999
```

Aritmatika pecahan *floating-point* biner memiliki banyak kejutan seperti ini. Masalah dengan "0.1" dijelaskan secara rinci di bawah ini, di bagian "Representation Error". Lihat [Perils of Floating Point](#) untuk penjelasan lebih lengkap tentang kejutan umum lainnya.

As that says near the end, "there are no easy answers." Still, don't be unduly wary of floating-point! The errors in Python float operations are inherited from the floating-point hardware, and on most machines are on the order of no more than 1 part in 2^{53} per operation. That's more than adequate for most tasks, but you do need to keep in mind that it's not decimal arithmetic, and that every float operation can suffer a new rounding error.

While pathological cases do exist, for most casual use of floating-point arithmetic you'll see the result you expect in the end if you simply round the display of your final results to the number of decimal digits you expect. For fine control over how a float is displayed see the `str.format()` method's format specifiers in formatstrings.

14.1 Kesalahan Representasi

Bagian ini menjelaskan contoh "0.1" secara terperinci, dan menunjukkan bagaimana Anda dapat melakukan analisis yang tepat atas kasus-kasus seperti ini sendiri. Diasumsikan terbiasa secara mendasar dengan representasi pecahan *floating point* biner.

Representation error refers to the fact that some (most, actually) decimal fractions cannot be represented exactly as binary (base 2) fractions. This is the chief reason why Python (or Perl, C, C++, Java, Fortran, and many others) often won't display the exact decimal number you expect:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Why is that? 1/10 and 2/10 are not exactly representable as a binary fraction. Almost all machines today (July 2010) use IEEE-754 floating point arithmetic, and almost all platforms map Python floats to IEEE-754 "double precision". 754 doubles contain 53 bits of precision, so on input the computer strives to convert 0.1 to the closest fraction it can of the form $J/2^N$ where J is an integer containing exactly 53 bits. Rewriting

```
1 / 10 ~= J / (2**N)
```

sebagai

```
J ~= 2**N / 10
```

dan mengingat bahwa J memiliki tepat 53 bit (adalah $\geq 2^{52}$ tetapi $< 2^{53}$), nilai terbaik untuk N adalah 56:

```
>>> 2**52
4503599627370496
>>> 2**53
9007199254740992
>>> 2**56/10
7205759403792793
```

That is, 56 is the only value for N that leaves J with exactly 53 bits. The best possible value for J is then that quotient rounded:

```
>>> q, r = divmod(2**56, 10)
>>> r
6
```

Karena sisanya lebih dari setengah dari 10, perkiraan terbaik diperoleh dengan membulatkan ke atas:

```
>>> q+1
7205759403792794
```

Therefore the best possible approximation to $1/10$ in 754 double precision is that over 2^{56} , or

```
7205759403792794 / 72057594037927936
```

Perhatikan bahwa sejak kami mengumpulkan, ini sebenarnya sedikit lebih besar dari $1/10$; jika kita belum mengumpulkan, hasil bagi akan sedikit lebih kecil dari $1/10$. Tetapi tidak dapatkah hal itu *exactly* $1/10$!

Jadi komputer tidak pernah "sees" $1/10$: apa yang dilihatnya adalah pecahan tepat yang diberikan di atas, perkiraan 754 *double* terbaik yang bisa didapatnya:

```
>>> .1 * 2**56
7205759403792794.0
```

If we multiply that fraction by 10^{30} , we can see the (truncated) value of its 30 most significant decimal digits:

```
>>> 7205759403792794 * 10**30 // 2**56
10000000000000000005551115123125L
```

meaning that the exact number stored in the computer is approximately equal to the decimal value 0.10000000000000000005551115123125. In versions prior to Python 2.7 and Python 3.1, Python rounded this value to 17 significant digits, giving '0.10000000000000001'. In current versions, Python displays a value based on the shortest decimal fraction that rounds correctly back to the true binary value, resulting simply in '0.1'.

15.1 Mode Interaktif

15.1.1 Penanganan Kesalahan

Ketika terjadi kesalahan, interpreter mencetak pesan kesalahan dan tumpukan jejak. Dalam mode interaktif, kemudian kembali ke prompt utama; ketika masukan datang dari berkas, akan keluar dengan status keluar yang tidak nol setelah mencetak tumpukan jejak. (Pengecualian yang ditangani oleh klausa `exception` dalam pernyataan :keyword:'try' bukan kesalahan dalam konteks ini.) Beberapa kesalahan berakibat fatal dan menyebabkan keluar dengan kode keluar selain-nol; ini berlaku untuk inkonsistensi internal dan beberapa kasus kehabisan memori. Semua pesan kesalahan ditulis ke aliran standar kesalahan; keluaran normal dari perintah yang dieksekusi ditulis ke standar keluaran.

Mengetik karakter interupsi (biasanya `Control-C` atau `Delete`) ke prompt utama atau sekunder membatalkan masukan dan kembali ke prompt utama.¹ Mengetik interupsi saat sebuah perintah dieksekusi memunculkan pengecualian `KeyboardInterrupt`, yang dapat ditangani oleh pernyataan :keyword:'try'.

15.1.2 Skrip Python Yang Dapat Dieksekusi

Pada sistem Unix BSD-ish, skrip Python dapat dibuat langsung dapat dieksekusi, seperti skrip shell, dengan meletakkan baris

```
#!/usr/bin/env python
```

(dengan asumsi bahwa interpreter ada di `PATH` pengguna) di awal skrip dan memberikan mode pada berkas untuk dapat dieksekusi. `#!` Harus merupakan dua karakter pertama dari file tersebut. Pada beberapa platform, baris pertama ini harus diakhiri dengan akhiran bergaya Unix (`'\n'`), bukan akhiran Windows (`'\r\n'`). Perhatikan bahwa hash, atau pon, karakter, `'#'`, digunakan untuk memulai komentar dengan Python.

Skrip bisa diberikan mode yang dapat dieksekusi, atau izin, menggunakan perintah `chmod`.

¹ Masalah dengan paket GNU Readline dapat mencegah hal ini.

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

Pada sistem Windows, tidak ada gagasan tentang "mode yang dapat dieksekusi". Pemasang Python secara otomatis mengaitkan file `.py` dengan `python.exe` sehingga klik dua kali pada file Python akan menjalankannya sebagai skrip. Ekstensi juga bisa `.pyw`, dalam hal ini, jendela konsol yang biasanya muncul dihilangkan.

15.1.3 Berkas Permulaan Interaktif

Ketika Anda menggunakan Python secara interaktif, seringkali berguna untuk menjalankan beberapa perintah standar setiap kali interpreter dimulai. Anda dapat melakukan ini dengan mengatur variabel lingkungan bernama `PYTHONSTARTUP` ke nama berkas yang berisi perintah permulaan Anda. Ini mirip dengan fitur `.profile` dari shell Unix.

File ini hanya dibaca dalam sesi interaktif, bukan ketika Python membaca perintah dari skrip, dan bukan ketika `/dev/tty` diberikan sebagai sumber perintah eksplisit (yang jika tidak berperilaku seperti sesi interaktif). Itu dieksekusi di namespace yang sama di mana perintah interaktif dieksekusi, sehingga objek yang didefinisikan atau impor dapat digunakan tanpa kualifikasi dalam sesi interaktif. Anda juga dapat mengubah prompt `sys.ps1` dan `sys.ps2` dalam file ini.

Jika Anda ingin membaca berkas permulaan tambahan dari direktori saat ini, Anda dapat memrogram ini dalam berkas permulaan global menggunakan kode seperti `if os.path.isfile('.pythonrc.py'): exec(open('.pythonrc.py').read())`. Jika Anda ingin menggunakan berkas permulaan dalam skrip, Anda harus melakukan ini secara eksplisit dalam skrip:

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

15.1.4 Modul Ubahsuaian

Python menyediakan dua kait untuk memungkinkan Anda menyesuaikannya: `sitecustomize` dan `usercustomize`. Untuk melihat cara kerjanya, Anda harus terlebih dahulu menemukan lokasi direktori `site-packages` pengguna Anda. Mulai Python dan operasikan kode ini:

```
>>> import site
>>> site.getusersitepackages()
'/home/user/.local/lib/python2.7/site-packages'
```

Sekarang Anda dapat membuat file bernama `usercustomize.py` di direktori itu dan memasukkan apa pun yang Anda inginkan di dalamnya. Ini akan memengaruhi setiap seruan dari Python, kecuali dimulai dengan opsi `-s` untuk menonaktifkan impor otomatis.

`sitecustomize` bekerja dengan cara yang sama, tetapi biasanya dibuat oleh administrator komputer di direktori `site-packages` global, dan diimpor sebelum `usercustomize`. Lihat dokumentasi modul `site` untuk lebih jelasnya.

>>> Prompt Python bawaan dari *shell* interaktif. Sering terlihat untuk contoh kode yang dapat dieksekusi secara interaktif dalam *interpreter*.

. . . The default Python prompt of the interactive shell when entering code for an indented code block, when within a pair of matching left and right delimiters (parentheses, square brackets, curly braces or triple quotes), or after specifying a decorator.

2to3 A tool that tries to convert Python 2.x code to Python 3.x code by handling most of the incompatibilities which can be detected by parsing the source and traversing the parse tree.

2to3 is available in the standard library as `lib2to3`; a standalone entry point is provided as `Tools/scripts/2to3`. See 2to3-reference.

kelas basis abstrak Abstract base classes complement *duck-typing* by providing a way to define interfaces when other techniques like `hasattr()` would be clumsy or subtly wrong (for example with magic methods). ABCs introduce virtual subclasses, which are classes that don't inherit from a class but are still recognized by `isinstance()` and `issubclass()`; see the `abc` module documentation. Python comes with many built-in ABCs for data structures (in the `collections` module), numbers (in the `numbers` module), and streams (in the `io` module). You can create your own ABCs with the `abc` module.

argumen A value passed to a *function* (or *method*) when calling the function. There are two types of arguments:

- *keyword argument*: an argument preceded by an identifier (e.g. `name=`) in a function call or passed as a value in a dictionary preceded by `**`. For example, 3 and 5 are both keyword arguments in the following calls to `complex()`:

```
complex(real=3, imag=5)
complex(**{'real': 3, 'imag': 5})
```

- *positional argument*: an argument that is not a keyword argument. Positional arguments can appear at the beginning of an argument list and/or be passed as elements of an *iterable* preceded by `*`. For example, 3 and 5 are both positional arguments in the following calls:

```
complex(3, 5)
complex(*(3, 5))
```

Arguments are assigned to the named local variables in a function body. See the calls section for the rules governing this assignment. Syntactically, any expression can be used to represent an argument; the evaluated value is assigned to the local variable.

See also the [parameter](#) glossary entry and the FAQ question on the difference between arguments and parameters.

atribut A value associated with an object which is referenced by name using dotted expressions. For example, if an object *o* has an attribute *a* it would be referenced as *o.a*.

BDFL Benevolent Dictator For Life, a.k.a. [Guido van Rossum](#), Python's creator.

bytes-like object An object that supports the buffer protocol, like `str`, `bytearray` or `memoryview`. Bytes-like objects can be used for various operations that expect binary data, such as compression, saving to a binary file or sending over a socket. Some operations need the binary data to be mutable, in which case not all bytes-like objects can apply.

bytecode Python source code is compiled into bytecode, the internal representation of a Python program in the CPython interpreter. The bytecode is also cached in `.pyc` and `.pyo` files so that executing the same file is faster the second time (recompilation from source to bytecode can be avoided). This "intermediate language" is said to run on a [virtual machine](#) that executes the machine code corresponding to each bytecode. Do note that bytecodes are not expected to work between different Python virtual machines, nor to be stable between Python releases.

Daftar instruksi-instruksi bytecode dapat ditemukan di dokumentasi pada the `dis` module.

kelas A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

classic class Any class which does not inherit from `object`. See [new-style class](#). Classic classes have been removed in Python 3.

paksa The implicit conversion of an instance of one type to another during an operation which involves two arguments of the same type. For example, `int(3.15)` converts the floating point number to the integer 3, but in `3+4.5`, each argument is of a different type (one `int`, one `float`), and both must be converted to the same type before they can be added or it will raise a `TypeError`. Coercion between two operands can be performed with the `coerce` built-in function; thus, `3+4.5` is equivalent to calling `operator.add(*coerce(3, 4.5))` and results in `operator.add(3.0, 4.5)`. Without coercion, all arguments of even compatible types would have to be normalized to the same value by the programmer, e.g., `float(3)+4.5` rather than just `3+4.5`.

bilangan kompleks An extension of the familiar real number system in which all numbers are expressed as a sum of a real part and an imaginary part. Imaginary numbers are real multiples of the imaginary unit (the square root of -1), often written *i* in mathematics or *j* in engineering. Python has built-in support for complex numbers, which are written with this latter notation; the imaginary part is written with a *j* suffix, e.g., `3+1j`. To get access to complex equivalents of the `math` module, use `cmath`. Use of complex numbers is a fairly advanced mathematical feature. If you're not aware of a need for them, it's almost certain you can safely ignore them.

manajer konteks An object which controls the environment seen in a `with` statement by defining `__enter__()` and `__exit__()` methods. See [PEP 343](#).

CPython The canonical implementation of the Python programming language, as distributed on [python.org](#). The term "CPython" is used when necessary to distinguish this implementation from others such as Jython or IronPython.

penghias A function returning another function, usually applied as a function transformation using the `@wrapper` syntax. Common examples for decorators are `classmethod()` and `staticmethod()`.

The decorator syntax is merely syntactic sugar, the following two function definitions are semantically equivalent:

```
def f(...):
    ...
f = staticmethod(f)

@staticmethod
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
def f(...):
    ...
```

The same concept exists for classes, but is less commonly used there. See the documentation for function definitions and class definitions for more about decorators.

descriptor Any *new-style* object which defines the methods `__get__()`, `__set__()`, or `__delete__()`. When a class attribute is a descriptor, its special binding behavior is triggered upon attribute lookup. Normally, using `a.b` to get, set or delete an attribute looks up the object named `b` in the class dictionary for `a`, but if `b` is a descriptor, the respective descriptor method gets called. Understanding descriptors is a key to a deep understanding of Python because they are the basis for many features including functions, methods, properties, class methods, static methods, and reference to super classes.

For more information about descriptors' methods, see descriptors.

kamus An associative array, where arbitrary keys are mapped to values. The keys can be any object with `__hash__()` and `__eq__()` methods. Called a hash in Perl.

dictionary view The objects returned from `dict.viewkeys()`, `dict.viewvalues()`, and `dict.viewitems()` are called dictionary views. They provide a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. To force the dictionary view to become a full list use `list(dictview)`. See dict-views.

docstring A string literal which appears as the first expression in a class, function or module. While ignored when the suite is executed, it is recognized by the compiler and put into the `__doc__` attribute of the enclosing class, function or module. Since it is available via introspection, it is the canonical place for documentation of the object.

duck-typing A programming style which does not look at an object's type to determine if it has the right interface; instead, the method or attribute is simply called or used ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. (Note, however, that duck-typing can be complemented with *abstract base classes*.) Instead, it typically employs `hasattr()` tests or *EAFP* programming.

EAFP Easier to ask for forgiveness than permission. This common Python coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false. This clean and fast style is characterized by the presence of many `try` and `except` statements. The technique contrasts with the *LBYL* style common to many other languages such as C.

ekspresi A piece of syntax which can be evaluated to some value. In other words, an expression is an accumulation of expression elements like literals, names, attribute access, operators or function calls which all return a value. In contrast to many other languages, not all language constructs are expressions. There are also *statements* which cannot be used as expressions, such as `print` or `if`. Assignments are also statements, not expressions.

modul tambahan A module written in C or C++, using Python's C API to interact with the core and with user code.

objek berkas An object exposing a file-oriented API (with methods such as `read()` or `write()`) to an underlying resource. Depending on the way it was created, a file object can mediate access to a real on-disk file or to another type of storage or communication device (for example standard input/output, in-memory buffers, sockets, pipes, etc.). File objects are also called *file-like objects* or *streams*.

There are actually three categories of file objects: raw binary files, buffered binary files and text files. Their interfaces are defined in the `io` module. The canonical way to create a file object is by using the `open()` function.

file-like object A synonym for *file object*.

finder An object that tries to find the *loader* for a module. It must implement a method named `find_module()`. See [PEP 302](#) for details.

floor division Mathematical division that rounds down to nearest integer. The floor division operator is `//`. For example, the expression `11 // 4` evaluates to 2 in contrast to the `2.75` returned by float true division. Note that `(-11) // 4` is `-3` because that is `-2.75` rounded *downward*. See [PEP 238](#).

fungsi A series of statements which returns some value to a caller. It can also be passed zero or more [arguments](#) which may be used in the execution of the body. See also [parameter](#), [method](#), and the function section.

__future__ A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter. For example, the expression `11/4` currently evaluates to 2. If the module in which it is executed had enabled *true division* by executing:

```
from __future__ import division
```

the expression `11/4` would evaluate to `2.75`. By importing the `__future__` module and evaluating its variables, you can see when a new feature was first added to the language and when it will become the default:

```
>>> import __future__
>>> __future__.division
_Feature((2, 2, 0, 'alpha', 2), (3, 0, 0, 'alpha', 0), 8192)
```

pengumpulan sampah The process of freeing memory when it is not used anymore. Python performs garbage collection via reference counting and a cyclic garbage collector that is able to detect and break reference cycles.

pembangkit A function which returns an iterator. It looks like a normal function except that it contains `yield` statements for producing a series of values usable in a `for`-loop or that can be retrieved one at a time with the `next()` function. Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the generator resumes, it picks up where it left off (in contrast to functions which start fresh on every invocation).

generator expression An expression that returns an iterator. It looks like a normal expression followed by a `for` expression defining a loop variable, range, and an optional `if` expression. The combined expression generates values for an enclosing function:

```
>>> sum(i*i for i in range(10))           # sum of squares 0, 1, 4, ... 81
285
```

GIL Lihat [global interpreter lock](#).

kunci interpreter global The mechanism used by the [CPython](#) interpreter to assure that only one thread executes Python [bytecode](#) at a time. This simplifies the CPython implementation by making the object model (including critical built-in types such as `dict`) implicitly safe against concurrent access. Locking the entire interpreter makes it easier for the interpreter to be multi-threaded, at the expense of much of the parallelism afforded by multi-processor machines.

However, some extension modules, either standard or third-party, are designed so as to release the GIL when doing computationally-intensive tasks such as compression or hashing. Also, the GIL is always released when doing I/O.

Past efforts to create a "free-threaded" interpreter (one which locks shared data at a much finer granularity) have not been successful because performance suffered in the common single-processor case. It is believed that overcoming this performance issue would make the implementation much more complicated and therefore costlier to maintain.

hashable An object is *hashable* if it has a hash value which never changes during its lifetime (it needs a `__hash__()` method), and can be compared to other objects (it needs an `__eq__()` or `__cmp__()` method). Hashable objects which compare equal must have the same hash value.

Hashability makes an object usable as a dictionary key and a set member, because these data structures use the hash value internally.

All of Python's immutable built-in objects are hashable, while no mutable containers (such as lists or dictionaries) are. Objects which are instances of user-defined classes are hashable by default; they all compare unequal (except

with themselves), and their hash value is derived from their `id()`.

IDLE Sebuah Lingkungan Pengembangan Terpadu untuk Python. IDLE adalah editor dasar dan lingkungan interpreter yang digabungkan dengan distribusi standar dari Python.

immutable An object with a fixed value. Immutable objects include numbers, strings and tuples. Such an object cannot be altered. A new object has to be created if a different value has to be stored. They play an important role in places where a constant hash value is needed, for example as a key in a dictionary.

integer division Mathematical division discarding any remainder. For example, the expression `11 / 4` currently evaluates to 2 in contrast to the `2.75` returned by float division. Also called *floor division*. When dividing two integers the outcome will always be another integer (having the floor function applied to it). However, if one of the operands is another numeric type (such as a `float`), the result will be coerced (see [coercion](#)) to a common type. For example, an integer divided by a float will result in a float value, possibly with a decimal fraction. Integer division can be forced by using the `//` operator instead of the `/` operator. See also [__future__](#).

importing The process by which Python code in one module is made available to Python code in another module.

importer An object that both finds and loads a module; both a [finder](#) and [loader](#) object.

interaktif Python has an interactive interpreter which means you can enter statements and expressions at the interpreter prompt, immediately execute them and see their results. Just launch `python` with no arguments (possibly by selecting it from your computer's main menu). It is a very powerful way to test out new ideas or inspect modules and packages (remember `help(x)`).

diinterpretasi Python is an interpreted language, as opposed to a compiled one, though the distinction can be blurry because of the presence of the bytecode compiler. This means that source files can be run directly without explicitly creating an executable which is then run. Interpreted languages typically have a shorter development/debug cycle than compiled ones, though their programs generally also run more slowly. See also [interactive](#).

iterable An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as `list`, `str`, and `tuple`) and some non-sequence types like `dict` and `file` and objects of any classes you define with an `__iter__()` or `__getitem__()` method. Iterables can be used in a `for` loop and in many other places where a sequence is needed (`zip()`, `map()`, ...). When an iterable object is passed as an argument to the built-in function `iter()`, it returns an iterator for the object. This iterator is good for one pass over the set of values. When using iterables, it is usually not necessary to call `iter()` or deal with iterator objects yourself. The `for` statement does that automatically for you, creating a temporary unnamed variable to hold the iterator for the duration of the loop. See also [iterator](#), [sequence](#), and [generator](#).

iterator An object representing a stream of data. Repeated calls to the iterator's `next()` method return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead. At this point, the iterator object is exhausted and any further calls to its `next()` method just raise `StopIteration` again. Iterators are required to have an `__iter__()` method that returns the iterator object itself so every iterator is also iterable and may be used in most places where other iterables are accepted. One notable exception is code which attempts multiple iteration passes. A container object (such as a `list`) produces a fresh new iterator each time you pass it to the `iter()` function or use it in a `for` loop. Attempting this with an iterator will just return the same exhausted iterator object used in the previous iteration pass, making it appear like an empty container.

Informasi lebih lanjut dapat ditemukan di [typeiter](#).

fungsi kunci A key function or collation function is a callable that returns a value used for sorting or ordering. For example, `locale.strxfrm()` is used to produce a sort key that is aware of locale specific sort conventions.

A number of tools in Python accept key functions to control how elements are ordered or grouped. They include `min()`, `max()`, `sorted()`, `list.sort()`, `heapq.nsmallest()`, `heapq.nlargest()`, and `itertools.groupby()`.

There are several ways to create a key function. For example, the `str.lower()` method can serve as a key function for case insensitive sorts. Alternatively, an ad-hoc key function can be built from a `lambda` expression such as `lambda r: (r[0], r[2])`. Also, the `operator` module provides three key function constructors:

`attrgetter()`, `itemgetter()`, and `methodcaller()`. See the [Sorting HOW TO](#) for examples of how to create and use key functions.

argumen kata kunci Lihat [argument](#).

lambda An anonymous inline function consisting of a single [expression](#) which is evaluated when the function is called. The syntax to create a lambda function is `lambda [parameters]: expression`

LBYL Look before you leap. This coding style explicitly tests for pre-conditions before making calls or lookups. This style contrasts with the [EAFP](#) approach and is characterized by the presence of many `if` statements.

In a multi-threaded environment, the LBYL approach can risk introducing a race condition between "the looking" and "the leaping". For example, the code, `if key in mapping: return mapping[key]` can fail if another thread removes *key* from *mapping* after the test, but before the lookup. This issue can be solved with locks or by using the EAFP approach.

daftar A built-in Python [sequence](#). Despite its name it is more akin to an array in other languages than to a linked list since access to elements is $O(1)$.

list comprehension A compact way to process all or part of the elements in a sequence and return a list with the results. `result = ["0x%02x" % x for x in range(256) if x % 2 == 0]` generates a list of strings containing even hex numbers (0x..) in the range from 0 to 255. The `if` clause is optional. If omitted, all elements in `range(256)` are processed.

loader An object that loads a module. It must define a method named `load_module()`. A loader is typically returned by a [finder](#). See [PEP 302](#) for details.

magic method An informal synonym for [special method](#).

pemetaan A container object that supports arbitrary key lookups and implements the methods specified in the `Mapping` or `MutableMapping` abstract base classes. Examples include `dict`, `collections.defaultdict`, `collections.OrderedDict` and `collections.Counter`.

metaclass The class of a class. Class definitions create a class name, a class dictionary, and a list of base classes. The metaclass is responsible for taking those three arguments and creating the class. Most object oriented programming languages provide a default implementation. What makes Python special is that it is possible to create custom metaclasses. Most users never need this tool, but when the need arises, metaclasses can provide powerful, elegant solutions. They have been used for logging attribute access, adding thread-safety, tracking object creation, implementing singletons, and many other tasks.

Informasi lebih lanjut dapat ditemukan di [metaclasses](#).

method A function which is defined inside a class body. If called as an attribute of an instance of that class, the method will get the instance object as its first [argument](#) (which is usually called `self`). See [function](#) and [nested scope](#).

method resolution order Method Resolution Order is the order in which base classes are searched for a member during lookup. See [The Python 2.3 Method Resolution Order](#) for details of the algorithm used by the Python interpreter since the 2.3 release.

modul An object that serves as an organizational unit of Python code. Modules have a namespace containing arbitrary Python objects. Modules are loaded into Python by the process of [importing](#).

Lihat juga [package](#).

MRO Lihat [method resolution order](#).

mutable Mutable objects can change their value but keep their `id()`. See also [immutable](#).

named tuple Any tuple-like class whose indexable elements are also accessible using named attributes (for example, `time.localtime()` returns a tuple-like object where the *year* is accessible either with an index such as `t[0]` or with a named attribute like `t.tm_year`).

A named tuple can be a built-in type such as `time.struct_time`, or it can be created with a regular class definition. A full featured named tuple can also be created with the factory function `collections.namedtuple()`. The latter approach automatically provides extra features such as a self-documenting representation like `Employee(name='jones', title='programmer')`.

namespace The place where a variable is stored. Namespaces are implemented as dictionaries. There are the local, global and built-in namespaces as well as nested namespaces in objects (in methods). Namespaces support modularity by preventing naming conflicts. For instance, the functions `__builtin__.open()` and `os.open()` are distinguished by their namespaces. Namespaces also aid readability and maintainability by making it clear which module implements a function. For instance, writing `random.seed()` or `itertools.izip()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

nested scope The ability to refer to a variable in an enclosing definition. For instance, a function defined inside another function can refer to variables in the outer function. Note that nested scopes work only for reference and not for assignment which will always write to the innermost scope. In contrast, local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace.

new-style class Any class which inherits from `object`. This includes all built-in types like `list` and `dict`. Only new-style classes can use Python's newer, versatile features like `__slots__`, descriptors, properties, and `__getattr__()`.

More information can be found in `newstyle`.

objek Any data with state (attributes or value) and defined behavior (methods). Also the ultimate base class of any *new-style class*.

paket A Python *module* which can contain submodules or recursively, subpackages. Technically, a package is a Python module with an `__path__` attribute.

parameter A named entity in a *function* (or method) definition that specifies an *argument* (or in some cases, arguments) that the function can accept. There are four types of parameters:

- *positional-or-keyword*: specifies an argument that can be passed either *positionally* or as a *keyword argument*. This is the default kind of parameter, for example `foo` and `bar` in the following:

```
def func(foo, bar=None): ...
```

- *positional-only*: specifies an argument that can be supplied only by position. Python has no syntax for defining positional-only parameters. However, some built-in functions have positional-only parameters (e.g. `abs()`).
- *var-positional*: specifies that an arbitrary sequence of positional arguments can be provided (in addition to any positional arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `*`, for example `args` in the following:

```
def func(*args, **kwargs): ...
```

- *var-keyword*: specifies that arbitrarily many keyword arguments can be provided (in addition to any keyword arguments already accepted by other parameters). Such a parameter can be defined by prepending the parameter name with `**`, for example `kwargs` in the example above.

Parameters can specify both optional and required arguments, as well as default values for some optional arguments.

See also the *argument* glossary entry, the FAQ question on the difference between arguments and parameters, and the function section.

PEP Python Enhancement Proposal. A PEP is a design document providing information to the Python community, or describing a new feature for Python or its processes or environment. PEPs should provide a concise technical specification and a rationale for proposed features.

PEPs are intended to be the primary mechanisms for proposing major new features, for collecting community input on an issue, and for documenting the design decisions that have gone into Python. The PEP author is responsible

for building consensus within the community and documenting dissenting opinions.

Lihat **PEP 1**.

positional argument Lihat *argument*.

Python 3000 Nickname for the Python 3.x release line (coined long ago when the release of version 3 was something in the distant future.) This is also abbreviated "Py3k".

Pythonic An idea or piece of code which closely follows the most common idioms of the Python language, rather than implementing code using concepts common to other languages. For example, a common idiom in Python is to loop over all elements of an iterable using a `for` statement. Many other languages don't have this type of construct, so people unfamiliar with Python sometimes use a numerical counter instead:

```
for i in range(len(food)):
    print food[i]
```

As opposed to the cleaner, Pythonic method:

```
for piece in food:
    print piece
```

jumlah referensi The number of references to an object. When the reference count of an object drops to zero, it is deallocated. Reference counting is generally not visible to Python code, but it is a key element of the *CPython* implementation. The `sys` module defines a `getrefcount()` function that programmers can call to return the reference count for a particular object.

__slots__ A declaration inside a *new-style class* that saves memory by pre-declaring space for instance attributes and eliminating instance dictionaries. Though popular, the technique is somewhat tricky to get right and is best reserved for rare cases where there are large numbers of instances in a memory-critical application.

urutan An *iterable* which supports efficient element access using integer indices via the `__getitem__()` special method and defines a `len()` method that returns the length of the sequence. Some built-in sequence types are `list`, `str`, `tuple`, and `unicode`. Note that `dict` also supports `__getitem__()` and `__len__()`, but is considered a mapping rather than a sequence because the lookups use arbitrary *immutable* keys rather than integers.

slice An object usually containing a portion of a *sequence*. A slice is created using the subscript notation, `[]` with colons between numbers when several are given, such as in `variable_name[1:3:5]`. The bracket (subscript) notation uses `slice` objects internally (or in older versions, `__getslice__()` and `__setslice__()`).

special method A method that is called implicitly by Python to execute a certain operation on a type, such as addition. Such methods have names starting and ending with double underscores. Special methods are documented in `specialnames`.

pernyataan A statement is part of a suite (a "block" of code). A statement is either an *expression* or one of several constructs with a keyword, such as `if`, `while` or `for`.

struct sequence A tuple with named elements. Struct sequences expose an interface similar to *named tuple* in that elements can be accessed either by index or as an attribute. However, they do not have any of the named tuple methods like `_make()` or `_asdict()`. Examples of struct sequences include `sys.float_info` and the return value of `os.stat()`.

teks tiga-kutip A string which is bound by three instances of either a quotation mark (") or an apostrophe ('). While they don't provide any functionality not available with single-quoted strings, they are useful for a number of reasons. They allow you to include unescaped single and double quotes within a string and they can span multiple lines without the use of the continuation character, making them especially useful when writing docstrings.

type The type of a Python object determines what kind of object it is; every object has a type. An object's type is accessible as its `__class__` attribute or can be retrieved with `type(obj)`.

universal newlines A manner of interpreting text streams in which all of the following are recognized as ending a line: the Unix end-of-line convention `'\n'`, the Windows convention `'\r\n'`, and the old Macintosh convention `'\r'`. See [PEP 278](#) and [PEP 3116](#), as well as `str.splitlines()` for an additional use.

lingkungan virtual Lingkungan runtime kooperatif yang memungkinkan pengguna dan aplikasi Python untuk menginstal dan memperbarui paket distribusi Python tanpa mengganggu perilaku aplikasi Python lain yang berjalan pada sistem yang sama.

mesin virtual A computer defined entirely in software. Python's virtual machine executes the *bytecode* emitted by the bytecode compiler.

Zen of Python Listing of Python design principles and philosophies that are helpful in understanding and using the language. The listing can be found by typing `"import this"` at the interactive prompt.

Tentang dokumen-dokumen ini

Dokumen-dokumen ini dihasilkan dari [reStructuredText](#) dengan [Sphinx](#), sebuah pemroses dokumen yang khusus ditulis untuk dokumentasi Python.

Pengembangan dokumentasi dan perangkat pengembangannya sepenuhnya upaya sukarela, seperti halnya Python. Jika anda ingin berkontribusi, silakan lihat halaman [reporting-bugs](#) untuk informasi cara melakukannya. Relawan baru selalu diterima!

Terima kasih banyak untuk:

- Fred L. Drake, Jr., pembuat awal kumpulan alat dokumentasi Python dan penulis banyak konten;
- [Docutils](#) proyek untuk membuat [reStructuredText](#) dan [Docutils](#) suite;
- Fredrik Lundh untuk [Alternative Python Reference](#) proyek dimana Sphinx mendapatkan banyak ide bagus.

B.1 Kontributor untuk dokumentasi Python

Banyak orang telah berkontribusi ke bahasa Python, pustaka standar Python, dan dokumentasi Python. Lihat [Misc/ACKS](#) di distribusi kode sumber Python untuk sebagian daftar kontributor-kontributor.

Hanya dengan masukan dan kontribusi dari komunitas Python sehingga Python memiliki dokumentasi yang sangat baik. Terima kasih!

Sejarah dan Lisensi

C.1 Sejarah perangkat lunak

Python diciptakan pada awal 1990-an oleh Guido van Rossum di Stichting Mathematisch Centrum (CWI, lihat <https://www.cwi.nl/>) di Belanda sebagai penerus bahasa yang disebut ABC. Guido tetap menjadi penulis utama Python, meskipun ia memasukkan banyak kontribusi dari orang lain.

Pada tahun 1995, Guido melanjutkan karyanya tentang Python di Corporation for National Research Initiatives (CNRI, lihat <https://www.cnri.reston.va.us/>) di Reston, Virginia di mana ia merilis beberapa versi perangkat lunak.

In May 2000, Guido and the Python core development team moved to BeOpen.com to form the BeOpen PythonLabs team. In October of the same year, the PythonLabs team moved to Digital Creations (now Zope Corporation; see <https://www.zope.org/>). In 2001, the Python Software Foundation (PSF, see <https://www.python.org/psf/>) was formed, a non-profit organization created specifically to own Python-related Intellectual Property. Zope Corporation is a sponsoring member of the PSF.

Semua rilis Python adalah Sumber Terbuka (lihat <https://opensource.org/> untuk Definisi Sumber Terbuka). Secara historis, sebagian besar, tetapi tidak semua, rilis Python juga kompatibel dengan GPL; tabel di bawah ini merangkum berbagai rilis.

Rilis	Berasal dari	Tahun	Pemilik	GPL compatible?
0.9.0 hingga 1.2	t/a	1991-1995	CWI	ya
1.3 hingga 1.5.2	1.2	1995-1999	CNRI	ya
1.6	1.5.2	2000	CNRI	tidak
2.0	1.6	2000	BeOpen.com	tidak
1.6.1	1.6	2001	CNRI	tidak
2.1	2.0+1.6.1	2001	PSF	tidak
2.0.1	2.0+1.6.1	2001	PSF	ya
2.1.1	2.1+2.0.1	2001	PSF	ya
2.1.2	2.1.1	2002	PSF	ya
2.1.3	2.1.2	2002	PSF	ya
2.2 dan ke atas	2.1.1	2001-sekarang	PSF	ya

Catatan: GPL-compatible doesn't mean that we're distributing Python under the GPL. All Python licenses, unlike the GPL, let you distribute a modified version without making your changes open source. The GPL-compatible licenses make it possible to combine Python with other software that is released under the GPL; the others don't.

Terima kasih kepada banyak sukarelawan eksternal yang telah bekerja di bawah arahan Guido untuk mewujudkan rilis-rilis ini.

C.2 Syarat dan ketentuan untuk mengakses atau menggunakan Python

C.2.1 LISENSI PERJANJIAN PSF UNTUK PYTHON 2.7.18

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),
→ and
the Individual or Organization ("Licensee") accessing and otherwise using
→ Python
2.7.18 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 2.7.18 alone or in any derivative
version, provided, however, that PSF's License Agreement and PSF's notice
→ of
copyright, i.e., "Copyright © 2001-2020 Python Software Foundation; All
→ Rights
Reserved" are retained in Python 2.7.18 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 2.7.18 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
2.7.18.
4. PSF is making Python 2.7.18 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 2.7.18 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 2.7.18
FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT
→ OF

- MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 2.7.18, 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 2.7.18, Licensee agrees
 to be bound by the terms and conditions of this License Agreement.

C.2.2 LISENSI PERJANJIAN BEOPEN.COM UNTUK PYTHON 2.0

LISENSI PERJANJIAN BEOPEN SUMBER TERBUKA PYTHON VERSI 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

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

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.

C.2.3 LISENSI PERJANJIAN CNRI UNTUK 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

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

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.

C.2.4 LISENSI PERJANJIAN CWI UNTUK PYTHON 0.9.0 SAMPAI 1.2

Copyright © 1991 - 1995, Stichting Mathematisch Centrum Amsterdam, The Netherlands. 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 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.

C.3 Lisensi dan Ucapan Terima Kasih untuk Perangkat Lunak yang Tergabung

Bagian ini tidak lengkap, tetapi daftar lisensi dan ucapan terima kasih yang terus bertambah untuk perangkat lunak pihak ketiga yang tergabung dalam distribusi Python.

C.3.1 Mersenne Twister

The `_random` module includes code based on a download from <http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/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.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html
email: m-mat @ math.sci.hiroshima-u.ac.jp (remove space)
```

C.3.2 Soket

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/>.

```
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
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

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 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 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.

C.3.3 Floating point exception control

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

```

-----
/               Copyright (c) 1996.               \
|           The Regents of the University of California.   |
|               All rights reserved.                   |
|                                                       |
|  Permission to use, copy, modify, and distribute this software for  |
|  any purpose without fee is hereby granted, provided that this en-  |
|  tire notice is included in all copies of any software which is or  |
|  includes a copy or modification of this software and in all      |
|  copies of the supporting documentation for such software.         |
|                                                       |
|  This work was produced at the University of California, Lawrence  |
|  Livermore National Laboratory under contract no. W-7405-ENG-48    |
|  between the U.S. Department of Energy and The Regents of the    |
|  University of California for the operation of UC LLNL.           |
|                                                       |
|               DISCLAIMER                                   |
|                                                       |
|  This software was prepared as an account of work sponsored by an  |
|  agency of the United States Government. Neither the United States  |
|  Government nor the University of California nor any of their em-  |
|  ployees, makes any warranty, express or implied, or assumes any  |
|  liability or responsibility for the accuracy, completeness, or    |
|  usefulness of any information, apparatus, product, or process    |
|  disclosed, or represents that its use would not infringe         |
|  privately-owned rights. Reference herein to any specific commer-  |
|  cial products, process, or service by trade name, trademark,     |
|  manufacturer, or otherwise, does not necessarily constitute or   |
|  imply its endorsement, recommendation, or favoring by the United  |

```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
| States Government or the University of California. The views and |
| opinions of authors expressed herein do not necessarily state or |
| reflect those of the United States Government or the University |
| of California, and shall not be used for advertising or product |
| \ endorsement purposes. /
| -----
```

C.3.4 MD5 message digest algorithm

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

```
Copyright (C) 1999, 2002 Aladdin Enterprises. All rights reserved.
```

```
This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.
```

```
Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:
```

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

```
L. Peter Deutsch
ghost@aladdin.com
```

```
Independent implementation of MD5 (RFC 1321).
```

```
This code implements the MD5 Algorithm defined in RFC 1321, whose
text is available at
```

```
http://www.ietf.org/rfc/rfc1321.txt
```

```
The code is derived from the text of the RFC, including the test suite
(section A.5) but excluding the rest of Appendix A. It does not include
any code or documentation that is identified in the RFC as being
copyrighted.
```

```
The original and principal author of md5.h is L. Peter Deutsch
<ghost@aladdin.com>. Other authors are noted in the change history
that follows (in reverse chronological order):
```

```
2002-04-13 lpd Removed support for non-ANSI compilers; removed
           references to Ghostscript; clarified derivation from RFC 1321;
           now handles byte order either statically or dynamically.
1999-11-04 lpd Edited comments slightly for automatic TOC extraction.
1999-10-18 lpd Fixed typo in header comment (ansi2knr rather than md5);
           added conditionalization for C++ compilation from Martin
           Purschke <purschke@bnl.gov>.
1999-05-03 lpd Original version.
```

C.3.5 Layanan soket asinkron

Modul `asynchat` dan `asyncore` berisi pemberitahuan berikut:

```
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.
```

C.3.6 Pengelolaan *Cookie*

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.
```

C.3.7 Pelacakan eksekusi

Modul trace berisi pemberitahuan berikut:

```
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.
```

C.3.8 UUencode and UUdecode functions

Modul uu berisi pemberitahuan berikut:

```
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
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
version is still 5 times faster, though.
- Arguments more compliant with Python standard
```

C.3.9 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.
```

C.3.10 test_epoll

The `test_epoll` contains the following notice:

```
Copyright (c) 2001-2006 Twisted Matrix Laboratories.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be
included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE
LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION
OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION
WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

C.3.11 Pilih kqueue

The select and contains the following notice for the kqueue interface:

```
Copyright (c) 2000 Doug White, 2006 James Knight, 2007 Christian Heimes
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.

```
THIS SOFTWARE IS PROVIDED BY THE AUTHOR 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 AUTHOR 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.
```

C.3.12 strtod dan dtoa

The file `Python/dtoa.c`, which supplies C functions `dtoa` and `strtod` for conversion of C doubles to and from strings, is derived from the file of the same name by David M. Gay, currently available from <http://www.netlib.org/fp/>. The original file, as retrieved on March 16, 2009, contains the following copyright and licensing notice:

```
/*****
 *
 * The author of this software is David M. Gay.
 *
 * Copyright (c) 1991, 2000, 2001 by Lucent Technologies.
 *
 * Permission to use, copy, modify, and distribute this software for any
 * purpose without fee is hereby granted, provided that this entire notice
 * is included in all copies of any software which is or includes a copy
 * or modification of this software and in all copies of the supporting
 * documentation for such software.
 *
 * THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
* WARRANTY.  IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY
* REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY
* OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.
*
*****/
```

C.3.13 OpenSSL

The modules `hashlib`, `posix`, `ssl`, `crypt` use the OpenSSL library for added performance if made available by the operating system. Additionally, the Windows and Mac OS X installers for Python may include a copy of the OpenSSL libraries, so we include a copy of the OpenSSL license here:

```
LICENSE ISSUES
=====
```

The OpenSSL toolkit stays under a dual license, i.e. both the conditions of the OpenSSL License and the original SSLeay license apply to the toolkit. See below for the actual license texts. Actually both licenses are BSD-style Open Source licenses. In case of any license issues related to OpenSSL please contact openssl-core@openssl.org.

```
OpenSSL License
-----
```

```
/* =====
 * Copyright (c) 1998-2008 The OpenSSL 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. All advertising materials mentioning features or use of this
 *    software must display the following acknowledgment:
 *    "This product includes software developed by the OpenSSL Project
 *    for use in the OpenSSL Toolkit. (http://www.openssl.org/)"
 *
 * 4. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to
 *    endorse or promote products derived from this software without
 *    prior written permission. For written permission, please contact
 *    openssl-core@openssl.org.
 *
 * 5. Products derived from this software may not be called "OpenSSL"
 *    nor may "OpenSSL" appear in their names without prior written
 *    permission of the OpenSSL Project.
 *
 * 6. Redistributions of any form whatsoever must retain the following
 *    acknowledgment:
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

*      "This product includes software developed by the OpenSSL Project
*      for use in the OpenSSL Toolkit (http://www.openssl.org/) "
*
* THIS SOFTWARE IS PROVIDED BY THE OpenSSL PROJECT ``AS IS'' AND ANY
* EXPRESSED 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 OpenSSL PROJECT OR
* ITS 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.
* =====
*
* This product includes cryptographic software written by Eric Young
* (eay@cryptsoft.com).  This product includes software written by Tim
* Hudson (tjh@cryptsoft.com).
*
*/

```

Original SSLeay License

```

-----

/* Copyright (C) 1995-1998 Eric Young (eay@cryptsoft.com)
 * All rights reserved.
 *
 * This package is an SSL implementation written
 * by Eric Young (eay@cryptsoft.com).
 * The implementation was written so as to conform with Netscapes SSL.
 *
 * This library is free for commercial and non-commercial use as long as
 * the following conditions are aheared to.  The following conditions
 * apply to all code found in this distribution, be it the RC4, RSA,
 * lhash, DES, etc., code; not just the SSL code.  The SSL documentation
 * included with this distribution is covered by the same copyright terms
 * except that the holder is Tim Hudson (tjh@cryptsoft.com).
 *
 * Copyright remains Eric Young's, and as such any Copyright notices in
 * the code are not to be removed.
 * If this package is used in a product, Eric Young should be given attribution
 * as the author of the parts of the library used.
 * This can be in the form of a textual message at program startup or
 * in documentation (online or textual) provided with the package.
 *
 * 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 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. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:

```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

*   "This product includes cryptographic software written by
*   Eric Young (eay@cryptsoft.com)"
*   The word 'cryptographic' can be left out if the routines from the library
*   being used are not cryptographic related :-).
* 4. If you include any Windows specific code (or a derivative thereof) from
*   the apps directory (application code) you must include an acknowledgement:
*   "This product includes software written by Tim Hudson (tjh@cryptsoft.com)"
*
* THIS SOFTWARE IS PROVIDED BY ERIC YOUNG ``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 AUTHOR 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.
*
* The licence and distribution terms for any publically available version or
* derivative of this code cannot be changed. i.e. this code cannot simply be
* copied and put under another distribution licence
* [including the GNU Public Licence.]
*/

```

C.3.14 expat

The pyexpat extension is built using an included copy of the expat sources unless the build is configured `--with-system-expat`:

```

Copyright (c) 1998, 1999, 2000 Thai Open Source Software Center Ltd
and Clark Cooper

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
"Software"), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

```

C.3.15 libffi

The `_ctypes` extension is built using an included copy of the libffi sources unless the build is configured `--with-system-libffi`:

```
Copyright (c) 1996-2008 Red Hat, Inc and others.

Permission is hereby granted, free of charge, to any person obtaining
a copy of this software and associated documentation files (the
``Software''), to deal in the Software without restriction, including
without limitation the rights to use, copy, modify, merge, publish,
distribute, sublicense, and/or sell copies of the Software, and to
permit persons to whom the Software is furnished to do so, subject to
the following conditions:

The above copyright notice and this permission notice shall be included
in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED ``AS IS'', WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND
NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT
HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE SOFTWARE.
```

C.3.16 zlib

The `zlib` extension is built using an included copy of the `zlib` sources if the `zlib` version found on the system is too old to be used for the build:

```
Copyright (C) 1995-2010 Jean-loup Gailly and Mark Adler

This software is provided 'as-is', without any express or implied
warranty. In no event will the authors be held liable for any damages
arising from the use of this software.

Permission is granted to anyone to use this software for any purpose,
including commercial applications, and to alter it and redistribute it
freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not
   claim that you wrote the original software. If you use this software
   in a product, an acknowledgment in the product documentation would be
   appreciated but is not required.

2. Altered source versions must be plainly marked as such, and must not be
   misrepresented as being the original software.

3. This notice may not be removed or altered from any source distribution.

Jean-loup Gailly          Mark Adler
jloup@gzip.org            madler@alumni.caltech.edu
```

LAMPIRAN D

Hak Cipta

Python dan dokumentasi ini adalah:

Copyright © 2001-2020 Python Software Foundation. All rights reserved.

Hak Cipta © 2000 BeOpen.com. Seluruh hak cipta.

Hak Cipta © 1995-2000 Corporation for National Research Initiatives. Seluruh hak cipta.

Hak Cipta © 1991-1995 Stichting Mathematisch Centrum. Seluruh hak cipta.

Lihat *Sejarah dan Lisensi* untuk lisensi lengkap dan informasi perizinan.

Non-abjad

..., [109](#)

*

statement, [28](#)

**

statement, [29](#)

2ke3, [109](#)

>>>, [109](#)

__all__, [50](#)

__builtin__

modul, [48](#)

__future__, [112](#)

__slots__, [116](#)

A

argumen, [109](#)

argumen kata kunci, [114](#)

atribut, [110](#)

B

BDFL, [110](#)

bilangan kompleks, [110](#)

bytecode, [110](#)

bytes-like object, [110](#)

C

classic class, [110](#)

coding

style, [30](#)

compileall

modul, [46](#)

CPython, [110](#)

D

daftar, [114](#)

descriptor, [111](#)

dictionary view, [111](#)

diinterpretasi, [113](#)

docstring, [111](#)

docstrings, [24, 29](#)

documentation strings, [24, 29](#)

duck-typing, [111](#)

E

EAFP, [111](#)

ekspresi, [111](#)

F

file

object, [56](#)

file-like object, [111](#)

finder, [111](#)

floor division, [112](#)

for

statement, [22](#)

fungsi, [112](#)

fungsi built-in

help, [83](#)

open, [56](#)

unicode, [16](#)

fungsi kunci, [113](#)

G

generator, [112](#)

generator expression, [112](#)

GIL, [112](#)

H

hashable, [112](#)

help

fungsi built-in, [83](#)

I

IDLE, [113](#)

immutable, [113](#)

importer, [113](#)

importing, [113](#)

integer division, [113](#)

interaktif, [113](#)
iterable, [113](#)
iterator, [113](#)

J

json
 modul, [58](#)
jumlah referensi, [116](#)

K

kamus, [111](#)
kelas, [110](#)
kelas basis abstrak, [109](#)
kunci interpreter global, [112](#)

L

lambda, [114](#)
LBYL, [114](#)
lingkungan virtual, [117](#)
list comprehension, [114](#)
loader, [114](#)

M

magic
 method, [114](#)
magic method, [114](#)
manajer konteks, [110](#)
mangling
 name, [78](#)
mesin virtual, [117](#)
metaclass, [114](#)
method, [114](#)
 magic, [114](#)
 object, [73](#)
 special, [116](#)
method resolution order, [114](#)
modul, [114](#)
 __builtin__, [48](#)
 compileall, [46](#)
 json, [58](#)
 readline, [100](#)
 rlcompleter, [100](#)
 sys, [47](#)
modul tambahan, [111](#)
module
 search path, [46](#)
MRO, [114](#)
mutable, [114](#)

N

name
 mangling, [78](#)
named tuple, [114](#)
namespace, [115](#)

nested scope, [115](#)
new-style class, [115](#)

O

object
 file, [56](#)
 method, [73](#)
objek, [115](#)
objek berkas, [111](#)
open
 fungsi built-in, [56](#)

P

paket, [115](#)
paksaan, [110](#)
parameter, [115](#)
PATH, [46](#), [107](#)
path
 module search, [46](#)
pembangkit, [112](#)
pemetaan, [114](#)
penghias, [110](#)
pengumpulan sampah, [112](#)
PEP, [115](#)
pernyataan, [116](#)
positional argument, [116](#)
Python 3000, [116](#)
Python Enhancement Proposals
 PEP 1, [116](#)
 PEP 8, [30](#)
 PEP 238, [112](#)
 PEP 278, [117](#)
 PEP 302, [111](#), [114](#)
 PEP 343, [110](#)
 PEP 3116, [117](#)
Pythonic, [116](#)
PYTHONPATH, [46](#), [47](#)
PYTHONSTARTUP, [100](#), [108](#)

R

readline
 modul, [100](#)
rlcompleter
 modul, [100](#)

S

search
 path, module, [46](#)
slice, [116](#)
special
 method, [116](#)
special method, [116](#)
statement
 *, [28](#)

T

U

V

Z

Zen of Python, 117