

---

# Python Tutorial

*Relis 3.8.20*

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

**September 08, 2024**

**Python Software Foundation  
Email: [docs@python.org](mailto:docs@python.org)**



<b>1</b>	<b>Membangkitkan Selera Anda</b>	<b>3</b>
<b>2</b>	<b>Menggunakan Interpreter Python</b>	<b>5</b>
2.1	Memanggil Interpreter . . . . .	5
2.2	Interpreter dan Lingkungannya . . . . .	7
<b>3</b>	<b>Pengantar Informal Tentang Python</b>	<b>9</b>
3.1	Menggunakan Python sebagai Kalkulator . . . . .	10
3.2	Langkah Awal Menuju Pemrograman . . . . .	16
<b>4</b>	<b>Lebih Banyak Alat Pengatur Aliran <i>Control Flow</i></b>	<b>19</b>
4.1	Pernyataan <code>if</code> . . . . .	19
4.2	Pernyataan <code>for</code> . . . . .	20
4.3	Fungsi <code>range()</code> . . . . .	20
4.4	Pernyataan <code>break</code> dan <code>continue</code> , dan <code>else</code> Klausa pada Perulangan <i>Loops</i> . . . . .	21
4.5	Pernyataan <code>pass</code> . . . . .	22
4.6	Mendefinisikan Fungsi . . . . .	23
4.7	Lebih lanjut tentang Mendefinisikan Fungsi . . . . .	25
4.8	Intermezzo: Gaya <i>Coding</i> . . . . .	32
<b>5</b>	<b>Struktur Data</b>	<b>35</b>
5.1	Lebih Lanjut tentang Daftar <i>Lists</i> . . . . .	35
5.2	Pernyataan <code>del</code> . . . . .	40
5.3	Tuples and <i>Urutan Sequences</i> . . . . .	40
5.4	Himpunan <i>Set</i> . . . . .	42
5.5	Kamus <i>Dictionaries</i> . . . . .	42
5.6	Teknik Perulangan . . . . .	44
5.7	Lebih lanjut tentang Kondisi . . . . .	45
5.8	Membandingkan <i>Urutan Sequences</i> dan Jenis Lainnya . . . . .	46
<b>6</b>	<b>Modul-Modul</b>	<b>47</b>
6.1	Lebih lanjut tentang Modul . . . . .	48
6.2	Modul Standar . . . . .	51
6.3	Fungsi <code>dir()</code> . . . . .	51
6.4	Paket . . . . .	53
<b>7</b>	<b>Masukan dan Keluaran</b>	<b>57</b>

7.1	Pemformatan Keluaran yang Lebih Menarik . . . . .	57
7.2	Membaca dan Menulis Berkas . . . . .	61
<b>8</b>	<b>Kesalahan <i>errors</i> dan Pengecualian <i>exceptions</i></b>	<b>65</b>
8.1	Kesalahan Sintaksis . . . . .	65
8.2	Pengecualian . . . . .	65
8.3	Menangani Pengecualian . . . . .	66
8.4	Memunculkan Pengecualian . . . . .	69
8.5	Pengecualian yang Ditentukan Pengguna . . . . .	69
8.6	Mendefinisikan Tindakan Pembersihan . . . . .	70
8.7	Tindakan Pembersihan yang Sudah Ditentukan . . . . .	71
<b>9</b>	<b>Classes</b>	<b>73</b>
9.1	Sepatah Kata Tentang Nama dan Objek . . . . .	74
9.2	Lingkup Python dan <i>Namespaces</i> . . . . .	74
9.3	Pandangan Pertama tentang Kelas . . . . .	76
9.4	Keterangan Acak . . . . .	80
9.5	Pewarisan . . . . .	81
9.6	Variabel Privat . . . . .	83
9.7	Barang Sisa <i>Odds and Ends</i> . . . . .	84
9.8	<i>Iterators</i> . . . . .	84
9.9	Pembangkit <i>Generator</i> . . . . .	85
9.10	Ekspresi Pembangkit <i>Generator</i> . . . . .	86
<b>10</b>	<b>Tur Singkat Pustaka Standar</b>	<b>87</b>
10.1	Antarmuka Sistem Operasi . . . . .	87
10.2	Berkas <i>Wildcard</i> . . . . .	88
10.3	Baris Perintah Berargumen . . . . .	88
10.4	Pengalihan Output Kesalahan dan Pengakhiran Program . . . . .	88
10.5	Pencocokan Pola String . . . . .	89
10.6	Matematika . . . . .	89
10.7	Akses internet . . . . .	90
10.8	Tanggal dan Waktu . . . . .	90
10.9	Kompresi Data . . . . .	91
10.10	Pengukuran Kinerja . . . . .	91
10.11	Kontrol kualitas . . . . .	91
10.12	Dilengkapi Baterai . . . . .	92
<b>11</b>	<b>Tur Singkat Pustaka Standar --- Bagian II</b>	<b>93</b>
11.1	Pemformatan Output . . . . .	93
11.2	<i>Templating</i> . . . . .	94
11.3	Bekerja dengan Tata Letak Rekam Data Biner . . . . .	95
11.4	<i>Multi-threading</i> . . . . .	96
11.5	Pencatatan . . . . .	96
11.6	Referensi yang Lemah . . . . .	97
11.7	Alat untuk Bekerja dengan Daftar <i>Lists</i> . . . . .	98
11.8	Aritmatika Pecahan <i>Floating Point</i> Desimal . . . . .	99
<b>12</b>	<b>Lingkungan dan Paket Virtual</b>	<b>101</b>
12.1	Pengantar . . . . .	101
12.2	Menciptakan Lingkungan Virtual . . . . .	101
12.3	Mengelola Paket dengan <i>pip</i> . . . . .	102
<b>13</b>	<b>Lalu apa sekarang?</b>	<b>105</b>

<b>14 Pengeditan Input Interaktif dan Penggantian Riwayat</b>	<b>107</b>
14.1 Pelengkapan Tab dan Pengeditan Riwayat . . . . .	107
14.2 Alternatif untuk Interpreter Interaktif . . . . .	107
<b>15 Aritmatika Pecahan <i>Floating Point</i>: Masalah dan Keterbatasan</b>	<b>109</b>
15.1 Kesalahan Representasi . . . . .	112
<b>16 Lampiran</b>	<b>115</b>
16.1 Mode Interaktif . . . . .	115
<b>A Ikhtisar</b>	<b>117</b>
<b>B Tentang dokumen-dokumen ini</b>	<b>131</b>
B.1 Kontributor untuk dokumentasi Python . . . . .	131
<b>C Sejarah dan Lisensi</b>	<b>133</b>
C.1 Sejarah perangkat lunak . . . . .	133
C.2 Syarat dan ketentuan untuk mengakses atau menggunakan Python . . . . .	134
C.3 Lisensi dan Ucapan Terima Kasih untuk Perangkat Lunak yang Tergabung . . . . .	138
<b>D Hak Cipta</b>	<b>151</b>
<b>Indeks</b>	<b>153</b>



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/python3.8` 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:

```
python3.8
```

ke shell.<sup>1</sup> Karena pilihan direktori tempat interpreter berada adalah opsi instalasi, bisa jadi di tempat lain; tanyakan kepada guru Python Anda di sekitar atau administrator sistem. (Mis., `/usr/local/python` adalah lokasi alternatif yang populer.)

On Windows machines where you have installed Python from the Microsoft Store, the `python3.8` command will be available. If you have the `py.exe` launcher installed, you can use the `py` command. See `setting-envvars` for other ways to launch Python.

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()`.

Fitur pengeditan garis interpreter termasuk pengeditan interaktif, penggantian riwayat dan penyelesaian kode pada sistem yang mendukung pustaka [GNU Readline](#). Mungkin pemeriksaan tercepat untuk melihat apakah pengeditan baris perintah didukung dengan mengetik `Control-P` ke prompt Python pertama yang Anda dapatkan. Jika berbunyi bip, Anda memiliki pengeditan baris perintah; lihat Lampiran *Pengeditan Input Interaktif dan Penggantian Riwayat* untuk pengantar tombol. Jika tampaknya tidak ada yang terjadi, atau jika muncul `^P`, pengeditan baris perintah tidak tersedia; Anda hanya akan dapat menggunakan backspace untuk menghapus karakter dari baris saat ini.

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

---

<sup>1</sup> Pada Unix, interpreter Python 3.x secara default tidak diinstal sebagai aplikasi yang dapat dieksekusi dengan nama `python`, sehingga tidak bertentangan dengan aplikasi Python 2.x yang diinstal secara bersamaan.

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.

Semua opsi baris perintah dijelaskan dalam *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:

```
$ python3.8
Python 3.8 (default, Sep 16 2015, 09:25:04)
[GCC 4.8.2] on linux
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 = True
>>> 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

Secara bawaan, berkas sumber Python diperlakukan sebagai tulisan *encoded* dalam UTF-8. Dalam pengkodean itu, sebagian besar karakter bahasa di dunia dapat digunakan secara bersamaan dalam string literal, pengidentifikasi dan komentar --- meskipun pustaka standar hanya menggunakan karakter ASCII untuk pengidentifikasi, sebuah konvensi yang harus diikuti oleh setiap kode portabel. Untuk menampilkan semua karakter ini dengan benar, editor Anda harus mengenali bahwa berkas tersebut adalah UTF-8, dan itu harus menggunakan font yang mendukung semua karakter dalam berkas.

Untuk mendeklarasikan penyandian *encoding* selain yang bawaan, baris komentar khusus harus ditambahkan sebagai baris *first* pada berkas. Sintaksnya adalah sebagai berikut

```
# -*- 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 penyandian *encoding* harus ditambahkan sebagai baris kedua pada berkas. Sebagai contoh:

```
#!/usr/bin/env python3
# -*- 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*6) / 4
5.0
>>> 8 / 5 # division always returns a floating point number
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.

Division (`/`) selalu mengembalikan float atau bilangan pecahan. Untuk melakukan *floor division* dan mendapatkan hasil integer atau bilangan bulat (menghilangkan hasil pecahannya) Anda dapat menggunakan operator `//`; untuk menghitung sisanya Anda dapat menggunakan `%`:

```
>>> 17 / 3 # classic division returns a float
5.666666666666667
>>>
>>> 17 // 3 # floor division discards the fractional part
5
>>> 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 pemangkatan<sup>1</sup>:

```
>>> 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
```

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

---

<sup>1</sup> 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`.



```
>>> 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:

```
>>> 4 * 3.75 - 1
14.0
```

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 sama<sup>2</sup>. `\` 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.'
```

Dalam interpreter interaktif, string keluaran diapit dengan tanda kutip dan karakter khusus dipisahkan dengan garis miring terbalik. Meskipun ini kadang-kadang terlihat berbeda dari input (tanda kutip terlampir dapat berubah), kedua string tersebut setara. String disertakan dalam tanda kutip ganda jika string berisi kutipan tunggal dan tidak ada tanda

<sup>2</sup> 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.

kutip ganda, jika tidak maka akan dilampirkan dalam tanda kutip tunggal. Fungsi `print()` menghasilkan keluaran yang lebih mudah dibaca, dengan menghilangkan tanda kutip terlampir dan dengan mencetak karakter yang dipisahkan dan spesial:

```
>>> '"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
File "<stdin>", line 1
    prefix 'thon'
          ^
SyntaxError: invalid syntax
>>> ('un' * 3) 'ium'
File "<stdin>", line 1
    ('un' * 3) 'ium'
            ^
SyntaxError: invalid syntax
```

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

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

String dapat diindeks atau *indexed* (disandakan), 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.

Selain pengindeksan, *slicing* atau mengiris juga didukung. Sementara pengindeksan digunakan untuk mendapatkan karakter individual, *slicing* memungkinkan Anda untuk mendapatkan 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'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
>>> word[2:] = 'py'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
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:**

**textseq** String adalah contoh *sequence types* atau jenis urutan, dan mendukung operasi umum yang didukung oleh jenis tersebut.

**string-methods** String mendukung sejumlah besar metode untuk transformasi dasar dan pencarian.

**f-strings** String literal yang memiliki ekspresi yang tersemat.

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

**old-string-formatting** Operasi pemformatan lama dipanggil ketika string adalah operan kiri dari operator `%` dijelaskan secara lebih rinci di sini.

### 3.1.3 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 (*items*) antara tanda kurung siku. *List* atau daftar mungkin berisi *items* dari tipe yang berbeda, tetapi biasanya semua *items* memiliki tipe yang sama.

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

Seperti string (dan semua bawaan lainnya tipe *sequence*), list atau daftar tersebut dapat diindeks dan diiris:

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

Semua operasi iris mengembalikan list atau daftar baru yang berisi elemen yang diminta. Ini berarti bahwa irisan berikut mengembalikan shallow copy dari list:

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

List atau daftar juga mendukung operasi seperti perangkaian:

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

## 3.2 Langkah Awal Menuju Pemrograman

Tentu saja, kita bisa menggunakan Python untuk tugas yang lebih rumit daripada menambahkan dua dan dua bersamaan. Sebagai contoh, kita dapat menulis awal dari sub-urutan *Fibonacci series* sebagai berikut:

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

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
...
0
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.
- Perulangan `while` dieksekusi selama kondisi (di sini: `a < 10`) masih benar. Dalam Python, seperti dalam C, nilai integer bukan nol bernilai benar; nol itu bernilai salah. Kondisi ini juga bisa berupa nilai string atau daftar, sebenarnya urutan apa pun; apapun dengan panjang yang tidak nol bernilai benar, urutan kosong bernilai salah. Tes yang digunakan dalam contoh adalah perbandingan sederhana. Operator perbandingan standar ditulis sama seperti dalam C: `<` (kurang dari), `>` (lebih besar dari), `==` (sama dengan), `<=` (kurang dari atau sama dengan), `>=` (lebih besar atau sama dengan) dan `!=` (tidak sama dengan).
- *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.
- Fungsi `print()` menulis nilai argumen(-argumen) yang diberikan. Ini berbeda dari hanya menulis ekspresi yang ingin Anda tulis (seperti yang kami lakukan sebelumnya dalam contoh kalkulator) dalam cara menangani beberapa argumen, jumlah floating point, dan string. String dicetak tanpa tanda kutip, dan spasi dimasukkan di antara *items*, sehingga Anda dapat memformat sesuatu dengan baik, seperti ini:

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

Argumen kata kunci `end` dapat digunakan untuk menghindari baris baru setelah keluaran, atau mengakhiri keluaran dengan string yang berbeda:

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





---

## 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 Pernyataan `if`

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

```
>>> x = int(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
```

Mungkin ada nol atau lebih bagian `elif`, dan bagian `else` adalah opsional. Kata kunci `'elif'` adalah kependekan dari `'else if'`, dan berguna untuk menghindari indentasi yang berlebihan. Sebuah `if ... elif ... elif ...` adalah urutan pengganti untuk pernyataan `switch` atau `case` yang ditemukan dalam bahasa lain.

## 4.2 Pernyataan `for`

Pernyataan `for` dalam Python sedikit berbeda dari apa yang mungkin Anda gunakan di C atau Pascal. Alih-alih selalu mengulangi perkembangan angka dalam aritmatika (seperti dalam Pascal), atau memberikan pengguna kemampuan untuk menentukan langkah iterasi dan kondisi berhenti (seperti C), Python pernyataan `for` diulangi pada item-item dari urutan apa pun (daftar *list* atau string), dalam urutan yang muncul dalam urutan. Misalnya (tidak ada permainan kata-kata):

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

Kode yang memodifikasi koleksi *collection* sambil mengulangi koleksi yang sama bisa sulit untuk diperbaiki. Sebagai gantinya, biasanya lebih mudah untuk mengulang salinan koleksi atau membuat koleksi baru:

```
# Strategy: Iterate over a copy
for user, status in users.copy().items():
    if status == 'inactive':
        del users[user]

# Strategy: Create a new collection
active_users = {}
for user, status in users.items():
    if status == 'active':
        active_users[user] = status
```

## 4.3 Fungsi `range()`

Jika Anda perlu mengulangi urutan angka, fungsi bawaan `range()` berguna. Ini menghasilkan urutan *pregressions* aritmatika:

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

Titik akhir yang diberikan tidak pernah menjadi bagian dari urutan yang dihasilkan; `range(10)` menghasilkan 10 nilai, indeks sah *legal* untuk item dengan urutan panjang 10. Dimungkinkan untuk membiarkan rentang mulai dari nomor lain, atau untuk menentukan kenaikan yang berbeda (bahkan negatif; kadang-kadang ini disebut 'step'):

```
range(5, 10)
    5, 6, 7, 8, 9

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

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

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

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

Hal aneh terjadi jika Anda hanya mencetak rentang `range`:

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

Dalam banyak hal objek dikembalikan oleh `range()` berperilaku seolah-olah itu adalah daftar *list*, tetapi sebenarnya tidak. Ini adalah objek yang mengembalikan item berurutan dari urutan yang diinginkan ketika Anda mengulanginya, tetapi itu tidak benar-benar membuat daftar *list*, sehingga menghemat ruang.

Kami mengatakan bahwa objek seperti itu adalah *iterable*, yaitu, cocok sebagai target untuk fungsi dan konstruksi yang mengharapkan sesuatu dari mana mereka dapat memperoleh item berturut-turut sampai pasokan habis. Kita telah melihat bahwa pernyataan `for` adalah konstruksi seperti itu, sedangkan contoh fungsi yang membutuhkan sebuah *iterable* adalah `sum()`:

```
>>> sum(range(4)) # 0 + 1 + 2 + 3
6
```

Nanti kita akan melihat lebih banyak fungsi yang mengembalikan *iterables* dan menganggap *iterables* sebagai argumen. Terakhir, mungkin Anda ingin tahu tentang cara mendapatkan daftar *list* dari suatu rentang `range`. Ini solusinya:

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

Dalam bab `tut-structure`, kita akan membahas lebih detail tentang `list()`.

## 4.4 Pernyataan `break` dan `continue`, dan `else` Klausula pada Perulangan *Loops*

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

Pernyataan perulangan *loop* mungkin memiliki klausula `else`; itu dieksekusi ketika loop berakhir melalui selesainya *exhaustion iterable* (dengan `for`) atau ketika kondisi menjadi salah (dengan `while`), tetapi tidak ketika loop diakhiri oleh pernyataan `break`. Ini dicontohkan oleh perulangan berikut, yang mencari bilangan prima:

```
>>> 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`.)

Ketika digunakan dengan sebuah perulangan, klausa `else` memiliki lebih banyak kesamaan dengan klausa `else` dari pernyataan `try` dibandingkan dengan pernyataan `if`: sebuah klausa `else` pernyataan `try` berjalan ketika tidak ada pengecualian terjadi, dan klausa `else` perulangan berjalan ketika tidak ada `break` terjadi. Untuk lebih lanjut tentang pernyataan `try` dan pengecualian, lihat [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 an odd number", num)
Found an even number 2
Found an odd number 3
Found an even number 4
Found an odd number 5
Found an even number 6
Found an odd number 7
Found an even number 8
Found an odd number 9
```

## 4.5 Pernyataan `pass`

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
... 
```

Tempat lain `pass` dapat digunakan adalah sebagai tempat-penampung *place-holder* untuk fungsi atau badan bersyarat *conditional body* saat Anda bekerja pada kode baru, memungkinkan Anda untuk terus berpikir pada tingkat yang lebih abstrak. `pass` diabaikan secara diam-diam:

```
>>> 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, end=' ')
...         a, b = b, a+b
...     print()
...
>>> # 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.

*execution* dari suatu fungsi memperkenalkan tabel simbol baru yang digunakan untuk variabel lokal dari fungsi tersebut. Lebih tepatnya, semua tugas variabel dalam suatu fungsi menyimpan nilai dalam tabel simbol lokal; sedangkan referensi variabel pertama-tama terlihat pada tabel simbol lokal, kemudian pada tabel simbol lokal lampiran *enclosing* fungsi, kemudian pada tabel simbol global, dan akhirnya pada tabel nama bawaan. Dengan demikian, variabel global dan variabel lampiran *enclosing* fungsi tidak dapat secara langsung menetapkan nilai dalam suatu fungsi (kecuali, untuk variabel global, disebutkan dalam pernyataan `global`, atau, untuk variabel lampiran *enclosing* fungsi, dinamai dalam pernyataan `nonlocal`), meskipun mungkin direferensikan.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object).<sup>1</sup> When a function calls another function, or calls itself recursively, a new local symbol table is created for that call.

Definisi fungsi mengasosiasikan nama fungsi dengan objek fungsi dalam tabel simbol saat ini. Sebuah interpreter dapat mengenali objek yang ditunjuk dengan nama itu sebagai fungsi yang ditentukan oleh pengguna. Nama lain juga dapat menunjuk ke objek fungsi yang sama dan juga dapat digunakan untuk mengakses fungsi tersebut:

```
>>> fib
<function fib at 10042ed0>
>>> f = fib
```

(berlanjut ke halaman berikutnya)

<sup>1</sup> 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).

(lanjutan dari halaman sebelumnya)

```
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
```

Berasal dari bahasa lain, Anda mungkin keberatan bahwa `fib` bukan fungsi melainkan prosedur karena tidak mengembalikan nilai. Bahkan, fungsi bahkan tanpa pernyataan `return` mengembalikan nilai, meskipun yang agak membosankan. Nilai ini disebut `None` (ini adalah nama bawaan). Menulis nilai `None` biasanya dihilangkan *suppressed* oleh *interpreter* jika itu akan menjadi satu-satunya nilai yang ditulis. Anda dapat melihatnya jika Anda benar-benar ingin menggunakan `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:

- Pernyataan `return` kembali dengan nilai dari suatu fungsi. `return` tanpa argumen ekspresi mengembalikan `None`. Keluar dari akhir suatu fungsi juga mengembalikan `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.

## 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, reminder='Please try again!'):
    while True:
        ok = 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 ValueError('invalid user response')
        print(reminder)
```

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, end=' ')
    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'
```

Ketika parameter formal terakhir dari bentuk `**name` ada, ia menerima kamus *dictionary* (lihat *typesmapping*) yang berisi semua argumen kata kunci *keyword argument* kecuali yang terkait dengan parameter formal. Ini dapat digabungkan



dengan parameter formal dari bentuk `*name` (dijelaskan dalam subbagian berikutnya) yang menerima *tuple* yang berisi argumen posisi di luar daftar parameter formal. (`*name` harus ada sebelum `**name`.) Misalnya, jika kita mendefinisikan fungsi seperti ini:

```
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)
    for kw in keywords:
        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.
-----
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Perhatikan bahwa bagaimana urutan argumen kata kunci dicetak telah dijamin sesuai dengan urutan yang disediakan dalam pemanggilan fungsi.

### 4.7.3 Parameter spesial

Secara bawaan, argumen dapat diteruskan ke fungsi Python baik dengan posisi atau secara eksplisit oleh kata kunci. Untuk keterbacaan dan kinerja, masuk akal untuk membatasi cara argumen dapat dilewatkan sehingga pengembang hanya perlu melihat definisi fungsi untuk menentukan apakah item dilewatkan secara posisi saja, posisi atau kata kunci, atau kata kunci saja.

Definisi fungsi mungkin terlihat seperti:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
    -----
    |           |           |
    |           | Positional or keyword |
    |           |           |
    |           |           | - Keyword only
    -- Positional only
```

di mana `/` dan `*` adalah opsional. Jika digunakan, simbol-simbol ini menunjukkan jenis parameter dengan cara argumen dilewatkan ke fungsi: posisi-saja, posisi-atau-kata kunci, dan kata kunci-saja. Parameter kata kunci juga disebut sebagai parameter bernama.

## Argumen Posisi-atau-Kata Kunci

Jika / dan \* tidak ada dalam definisi fungsi, argumen dapat diteruskan ke fungsi dengan posisi atau kata kunci.

### Parameter Posisi-saja

Melihat ini sedikit lebih detail, dimungkinkan untuk menandai parameter tertentu sebagai *positional-only*. Jika *positional-only*, urutan parameter penting, dan parameter tidak dapat dilewatkan dengan kata kunci. Parameter posisi-saja ditempatkan sebelum / (garis miring). / Digunakan untuk secara logis memisahkan parameter posisi-saja dari parameter lainnya. Jika tidak ada / dalam definisi fungsi, tidak ada parameter posisi-saja.

Parameter yang mengikuti / dapat berupa *positional-or-keyword* atau *keyword-only*.

### Argumen Kata Kunci-saja

Untuk menandai parameter sebagai *keyword-only*, yang menunjukkan parameter harus dilewatkan dengan argumen kata kunci, tempatkan \* dalam daftar argumen tepat sebelum parameter *keyword-only*.

### Contoh Fungsi

Perhatikan definisi fungsi contoh berikut dengan memperhatikan marker / dan \*:

```
>>> def standard_arg(arg):
...     print(arg)
...
>>> def pos_only_arg(arg, /):
...     print(arg)
...
>>> def kwd_only_arg(*, arg):
...     print(arg)
...
>>> def combined_example(pos_only, /, standard, *, kwd_only):
...     print(pos_only, standard, kwd_only)
```

Definisi fungsi pertama, `standard_arg`, bentuk yang paling akrab, tidak menempatkan batasan pada konvensi pemanggilan dan argumen dapat dilewatkan dengan posisi atau kata kunci:

```
>>> standard_arg(2)
2

>>> standard_arg(arg=2)
2
```

Fungsi kedua `pos_only_arg` dibatasi hanya menggunakan parameter posisi karena ada / dalam definisi fungsi:

```
>>> pos_only_arg(1)
1

>>> pos_only_arg(arg=1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pos_only_arg() got an unexpected keyword argument 'arg'
```

Fungsi ketiga `kwd_only_args` hanya memungkinkan argumen kata kunci *keyword argument* seperti ditunjukkan oleh \* dalam definisi fungsi:

```
>>> kwd_only_arg(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: kwd_only_arg() takes 0 positional arguments but 1 was given

>>> kwd_only_arg(arg=3)
3
```

Dan yang terakhir menggunakan ketiga konvensi pemanggilan dalam definisi fungsi yang sama:

```
>>> combined_example(1, 2, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() takes 2 positional arguments but 3 were given

>>> combined_example(1, 2, kwd_only=3)
1 2 3

>>> combined_example(1, standard=2, kwd_only=3)
1 2 3

>>> combined_example(pos_only=1, standard=2, kwd_only=3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: combined_example() got an unexpected keyword argument 'pos_only'
```

Akhirnya, pertimbangkan definisi fungsi ini yang memiliki potensi tabrakan antara argumen posisi `name` dan `**kwds` yang memiliki `name` sebagai kunci:

```
def foo(name, **kwds):
    return 'name' in kwds
```

Tidak ada kemungkinan panggilan yang memungkinkan untuk mengembalikan ke dalam `"True"` karena kata kunci `"name"` akan selalu terikat ke parameter pertama. Sebagai contoh:

```
>>> foo(1, **{'name': 2})
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: foo() got multiple values for argument 'name'
>>>
```

Tetapi menggunakan `/` (argumen posisi saja), ini dimungkinkan karena memungkinkan `name` sebagai argumen posisi dan `'name'` sebagai kunci dalam argumen kata kunci *keyword argument*:

```
def foo(name, /, **kwds):
    return 'name' in kwds
>>> foo(1, **{'name': 2})
True
```

Dengan kata lain, nama-nama parameter posisi-saja dapat digunakan dalam `**kwds` tanpa ambiguitas.

## Rekap

Contoh kasus dimana akan menentukan parameter mana yang akan digunakan dalam definisi fungsi:

```
def f(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
```

Sebagai pedoman:

- Gunakan posisi-saja jika Anda ingin nama parameter tidak tersedia bagi pengguna. Ini berguna ketika nama parameter tidak memiliki arti nyata, jika Anda ingin menegaskan urutan argumen ketika fungsi dipanggil atau jika Anda perlu mengambil beberapa parameter posisi dan kata kunci bergantian *arbitrary*.
- Gunakan kata kunci-saja ketika nama memiliki makna dan definisi fungsi lebih mudah dipahami dengan secara eksplisit menggunakan nama atau Anda ingin mencegah pengguna mengandalkan posisi argumen yang dikirimkan.
- Untuk API, gunakan posisi-saja untuk mencegah perubahan yang merusak dari API jika nama parameter diubah di masa mendatang.

### 4.7.4 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 *tuples*). Sebelum jumlah variabel argumen, nol atau lebih argumen normal dapat muncul.

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

Biasanya, argumen *variadic* ini akan menjadi yang terakhir dalam daftar parameter formal, karena mereka mengambil semua argumen masukan yang tersisa yang diteruskan ke fungsi. Parameter formal apa pun yang muncul setelah parameter *\*args* adalah argumen 'keyword-only', yang berarti bahwa parameter itu hanya dapat digunakan sebagai kata kunci alih-alih argumen posisi.

```
>>> def concat(*args, sep="/"):  
...     return sep.join(args)  
...  
>>> concat("earth", "mars", "venus")  
'earth/mars/venus'  
>>> concat("earth", "mars", "venus", sep=".")  
'earth.mars.venus'
```

### 4.7.5 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:

```
>>> list(range(3, 6))           # normal call with separate arguments  
[3, 4, 5]  
>>> args = [3, 6]  
>>> list(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, end=' ')
...     print("if you put", voltage, "volts through it.", end=' ')
...     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 !
```

## 4.7.6 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')]
```

## 4.7.7 String Dokumentasi

Berikut adalah beberapa konvensi tentang konten dan format string dokumentasi.

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.
```

### 4.7.8 Anotasi Fungsi

Function annotations informasi metadata yang sepenuhnya opsional tentang jenis yang digunakan oleh fungsi yang ditentukan pengguna (lihat [PEP 3107](#) dan [PEP 484](#) untuk informasi lebih lanjut).

*Annotations* are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. The following example has a required argument, an optional argument, and the return value annotated:

```
>>> def f(ham: str, eggs: str = 'eggs') -> str:
...     print("Annotations:", f.__annotations__)
...     print("Arguments:", ham, eggs)
...     return ham + ' and ' + eggs
>>> f('spam')
Annotations: {'ham': <class 'str'>, 'return': <class 'str'>, 'eggs': <class 'str'>}
Arguments: spam eggs
'spam and eggs'
```

## 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)`.
- Beri nama kelas dan fungsi Anda secara konsisten; konvensi ini menggunakan `UpperCamelCase` untuk kelas dan `lowercase_with_underscores` untuk fungsi dan metode. Selalu gunakan `self` sebagai nama untuk argumen metode pertama (lihat `tut-firstclass` untuk lebih lanjut tentang kelas dan metode).
- Jangan gunakan pengkodean ajaib *fancy encodings* jika kode Anda dimaksudkan untuk digunakan di lingkungan internasional. Default Python, UTF-8, atau bahkan ASCII biasa berfungsi paling baik dalam hal apa pun.
- Demikian juga, jangan gunakan karakter non-ASCII dalam pengidentifikasi jika hanya ada sedikit kesempatan orang berbicara bahasa yang berbeda akan membaca atau merawat kode.





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)`

Tambahkan item ke akhir daftar *list*. Setara dengan `a[len(a):] = [x]`.

`list.extend(iterable)`

Perpanjang daftar *list* dengan menambahkan semua item dari *iterable*. Setara dengan `a[len(a):] = iterable`.

`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)`

Hapus item pertama dari daftar *list* yang nilainya sama dengan `x`. Ini memunculkan `ValueError` jika tidak ada item seperti itu.

`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.clear()`

Hapus semua item dari daftar *list*. Setara dengan `del a[:]`.

`list.index(x[, start[, end]])`

Kembalikan indeks berbasis nol dalam daftar item pertama yang nilainya sama dengan `x`. Menimbulkan `ValueError` jika tidak ada item seperti itu.

Argumen opsional *start* dan *end* ditafsirkan seperti dalam notasi *slice* dan digunakan untuk membatasi pencarian ke urutan tertentu dari daftar. Indeks yang dikembalikan dihitung relatif terhadap awal urutan penuh daripada argumen *start*.

`list.count(x)`

Kembalikan berapa kali *x* muncul dalam daftar.

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

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

`list.reverse()`

Balikkan elemen daftar *list* di tempatnya.

`list.copy()`

Kembalikan salinan daftar *list* yang dangkal. Setara dengan `a[:]`.

Contoh yang menggunakan sebagian besar metode daftar *list*:

```
>>> fruits = ['orange', 'apple', 'pear', 'banana', 'kiwi', 'apple', 'banana']
>>> fruits.count('apple')
2
>>> fruits.count('tangerine')
0
>>> fruits.index('banana')
3
>>> fruits.index('banana', 4)  # Find next banana starting a position 4
6
>>> fruits.reverse()
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange']
>>> fruits.append('grape')
>>> fruits
['banana', 'apple', 'kiwi', 'banana', 'pear', 'apple', 'orange', 'grape']
>>> fruits.sort()
>>> fruits
['apple', 'apple', 'banana', 'banana', 'grape', 'kiwi', 'orange', 'pear']
>>> fruits.pop()
'pear'
```

Anda mungkin telah memperhatikan bahwa metode seperti `insert`, `remove` atau `sort` yang hanya mengubah daftar *list* tidak memiliki nilai pengembalian yang dicetak -- mereka mengembalikan standar `None`.<sup>1</sup> Ini adalah prinsip desain untuk semua struktur data yang bisa berubah *mutable* dalam Python.

Hal lain yang mungkin Anda perhatikan adalah bahwa tidak semua data dapat diurutkan atau dibandingkan. Misalnya, `[None, 'hello', 10]` tidak mengurutkan karena bilangan bulat tidak dapat dibandingkan dengan string dan `None` tidak dapat dibandingkan dengan jenis lainnya. Juga, ada beberapa tipe yang tidak memiliki hubungan pengurutan yang ditentukan. Sebagai contoh, `3+4j < 5+7j` bukan perbandingan yang valid.

---

<sup>1</sup> Bahasa lain dapat mengembalikan objek bermutasi, yang memungkinkan metode berantai *chaining*, seperti `d->insert("a")->remove("b")->sort();`.

### 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 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):
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
...     squares.append(x**2)
...
>>> squares
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Perhatikan bahwa ini membuat (atau menimpa) variabel bernama `x` yang masih ada setelah loop selesai. Kami dapat menghitung daftar kotak tanpa efek samping menggunakan:

```
squares = list(map(lambda x: x**2, range(10)))
```

atau, dengan kata lain:

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

yang lebih ringkas dan mudah dibaca.

Pemahaman daftar *list comprehension* terdiri dari tanda kurung yang berisi ekspresi diikuti oleh klausa `for`, lalu nol atau lebih klausa `for` atau `if`. Hasilnya akan menjadi daftar baru yang dihasilkan dari mengevaluasi ekspresi dalam konteks dari klausa `for` dan `if` yang mengikutinya. Sebagai contoh, *listcomp* ini menggabungkan elemen dari dua daftar jika tidak sama:

```
>>> [(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)]
```

dan itu setara dengan:

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

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
```

(berlanjut ke halaman berikutnya)



(lanjutan dari halaman sebelumnya)

```
>>> 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:

```
>>> list(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 Pernyataan `del`

Ada cara untuk menghapus item dari daftar yang diberikan indeksinya, bukan nilainya: pernyataan `del`. Ini berbeda dari metode `pop()` yang mengembalikan nilai. Pernyataan `del` juga dapat digunakan untuk menghapus irisan dari daftar *list* atau menghapus seluruh daftar *list* (yang kami lakukan sebelumnya dengan menetapkan daftar kosong pada *slice*). Sebagai contoh:

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

`del` 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
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

((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; tupel 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

```

Ini disebut, cukup tepat, urutan membongkar *sequence unpacking* dan berfungsi untuk setiap urutan di sisi kanan. Urutan membongkar mensyaratkan bahwa ada banyak variabel di sisi kiri tanda sama dengan ada elemen dalam urutan. Perhatikan bahwa banyak tugas benar-benar hanya kombinasi dari *tuple packing* dan urutan pembongkaran *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'}
>>> print(basket)           # show that duplicates have been removed
{'orange', 'banana', 'pear', 'apple'}
>>> 'orange' in basket      # fast membership testing
True
>>> 'crabgrass' in basket
False

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

Sebaiknya pikirkan kamus *dictionary* sebagai satu set *key: value* berpasangan, dengan persyaratan bahwa kunci tersebut unik (dalam satu kamus *dictionary*). Sepasang kurung kurawal membuat kamus *dictionary* kosong: `{}`. Menempatkan



daftar pasangan kunci:nilai yang dipisah koma dalam kurung menambahkan pasangan kunci:nilai ke kamus *dictionary*; ini juga cara kamus *dictionary* ditulis pada keluaran.

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.

Melakukan `list(d)` pada kamus mengembalikan daftar *list* semua kunci yang digunakan dalam kamus, dalam urutan penyisipan (jika Anda ingin diurutkan, cukup gunakan `sorted(d)` sebagai gantinya). Untuk memeriksa apakah ada satu kunci dalam kamus, gunakan kaca kunci `in`.

Ini adalah contoh kecil menggunakan kamus *dictionary*:

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
>>> list(tel)
['jack', 'guido', 'irv']
>>> sorted(tel)
['guido', 'irv', 'jack']
>>> 'guido' in tel
True
>>> 'jack' not in tel
False
```

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

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

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, 'guido': 4127, 'jack': 4098}
```

## 5.6 Teknik Perulangan

Saat mengulang kamus *dictionaries*, kunci *key* dan nilai *value* terkait dapat diambil pada saat yang sama menggunakan metode `items()`.

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

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(range(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
```

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.

The comparison operators `in` and `not in` check whether a value occurs (does not occur) in a sequence. The operators `is` and `is not` compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

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'
```

Perhatikan bahwa dalam Python, tidak seperti C, penugasan di dalam ekspresi harus dilakukan secara eksplisit dengan operator *walrus* `:`. Ini menghindari masalah kelas umum yang dihadapi dalam program C: menyetikkan `=` dalam ekspresi ketika `==` dimaksudkan.

## 5.8 Membandingkan Urutan *Sequences* dan Jenis Lainnya

Objek urutan *sequence* biasanya dapat dibandingkan dengan objek lain dengan jenis urutan yang sama. Perbandingan menggunakan pengurutan *lexicographical*: pertama dua item pertama dibandingkan, dan jika mereka berbeda ini menentukan hasil perbandingan; jika mereka sama, dua item berikutnya dibandingkan, dan seterusnya, sampai urutan mana pun habis. Jika dua item yang akan dibandingkan adalah urutannya sendiri dari jenis yang sama, perbandingan leksikografis dilakukan secara rekursif. Jika semua item dari dua urutan membandingkan hasilnya sama, urutannya dianggap sama. Jika satu urutan adalah sub-urutan awal dari yang lain, urutan yang lebih pendek adalah yang lebih kecil (lebih pendek). Pengurutan leksikografis untuk string menggunakan nomor titik kode Unicode untuk mengurutkan masing-masing karakter. Beberapa contoh perbandingan antara urutan dengan tipe yang sama:

```
(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)
```

Perhatikan bahwa membandingkan objek dari berbagai jenis dengan `<` atau `>` adalah sah asalkan objek memiliki metode perbandingan yang sesuai. Misalnya, tipe numerik campuran dibandingkan menurut nilai numeriknya, sehingga 0 sama dengan 0.0, dll. Jika tidak, alih-alih memberikan penyusunan acak, interpreter akan memunculkan pengecualian `TypeError`.

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` diakhirinya. Dalam sebuah modul, nama modul (sebagai string) tersedia sebagai nilai variabel global `__name__`. Misalnya, gunakan editor teks favorit Anda untuk membuat bernama bernama `fibo.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 a < n:
        print(a, end=' ')
        a, b = b, a+b
    print()

def fib2(n):   # return Fibonacci series up to n
    result = []
    a, b = 0, 1
    while a < n:
        result.append(a)
        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)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> fibo.fib2(100)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
>>> fibo.__name__
'fibo'
```

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

```
>>> fib = fibo.fib
>>> fib(500)
0 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 `import`.<sup>1</sup> (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 `import`.

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

```
>>> from fibo import fib, fib2
>>> fib(500)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ini tidak memperkenalkan nama modul dari mana `import` 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)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

Ini mengimpor semua nama kecuali yang dimulai dengan garis bawah (`_`). Dalam kebanyakan kasus, programmer Python tidak menggunakan fasilitas ini karena ia memperkenalkan sekumpulan nama yang tidak diketahui ke dalam *interpreter*, mungkin menyembunyikan beberapa hal yang sudah Anda definisikan.

---

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

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.

Jika nama modul diikuti oleh `as`, maka nama setelah `as` terikat langsung ke modul yang diimpor.

```
>>> 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:** Untuk alasan efisiensi, setiap modul hanya diimpor satu kali per sesi *interpreter*. Karenanya, jika Anda mengubah modul, Anda harus memulai ulang *interpreter* -- atau, jika hanya satu modul yang ingin Anda uji secara interaktif, gunakan `importlib.reload()`, mis. `import importlib; importlib.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
0 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:

- Direktori yang berisi skrip masukan (atau direktori saat ini ketika tidak ada file ditentukan).
- `PYTHONPATH` (daftar nama direktori, dengan sintaksis yang sama dengan variabel shell `PATH`).
- Bawaan yang bergantung pada instalasi.

---

**Catatan:** Pada sistem file yang mendukung symlink, direktori yang berisi skrip masukan dihitung setelah symlink diikuti. Dengan kata lain direktori yang berisi symlink **not** ditambahkan ke jalur pencarian modul.

---

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"

Untuk mempercepat memuat modul, Python menyimpan *cache* versi terkompilasi dari setiap modul di direktori `__pycache__` dengan nama `module.version.pyc`, di mana versi menyandikan format berkas terkompilasi; umumnya berisi nomor versi Python. Misalnya, dalam rilis CPython 3.3 versi yang dikompilasi dari `spam.py` akan di-*cache* sebagai `__pycache__/spam.cpython-33.pyc`. Konvensi penamaan ini memungkinkan modul yang dikompilasi dari rilis yang beragam dan versi Python yang berbeda untuk hidup berdampingan.

Python memeriksa tanggal modifikasi sumber terhadap versi yang dikompilasi untuk melihat apakah itu kedaluwarsa dan perlu dikompilasi ulang. Ini adalah proses yang sepenuhnya otomatis. Juga, modul yang dikompilasi adalah platform-independen, sehingga perpustakaan yang sama dapat dibagi di antara sistem dengan arsitektur yang berbeda.

Python tidak memeriksa *cache* dalam dua keadaan. Pertama, selalu mengkompilasi ulang dan tidak menyimpan hasil untuk modul yang dimuat langsung dari baris perintah. Kedua, itu tidak memeriksa *cache* jika tidak ada modul sumber. Untuk mendukung distribusi non-sumber (dikompilasi saja), modul yang dikompilasi harus ada di direktori sumber, dan tidak boleh ada modul sumber.

Beberapa tips untuk para ahli:

- Anda dapat menggunakan `-O` atau `-OO` mengaktifkan perintah Python untuk mengurangi ukuran modul yang dikompilasi. Saklar `-O` menghapus pernyataan tegas `assert`, saklar `-OO` menghapus pernyataan tegas `assert` dan string `__doc__`. Karena beberapa program bergantung pada ketersediaannya, Anda hanya boleh menggunakan opsi ini jika Anda tahu apa yang Anda lakukan. Modul "Optimized" memiliki tag `opt-` dan biasanya lebih kecil. Rilis di masa depan dapat mengubah efek pengoptimalan.
- Suatu program tidak berjalan lebih cepat ketika itu dibaca dari file `.pyc` daripada ketika itu dibaca dari file `.py`; satu-satunya hal yang lebih cepat tentang berkas `.pyc` adalah kecepatan memuatnya.
- Modul `compileall` dapat membuat berkas `.pyc` untuk semua modul dalam direktori.
- Ada detail lebih lanjut tentang proses ini, termasuk bagan alur keputusan, di [PEP 3147](#).



## 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__', '__loader__', '__name__',
 '__package__', '__stderr__', '__stdin__', '__stdout__',
 '_clear_type_cache', '_current_frames', '_debugmallocstats', '_getframe',
 '_home', '_mercurial', '_xoptions', 'abiflags', 'api_version', 'argv',
 'base_exec_prefix', 'base_prefix', 'builtin_module_names', 'byteorder',
 'call_tracing', 'callstats', 'copyright', 'displayhook',
 'dont_write_bytecode', 'exc_info', 'excepthook', 'exec_prefix',
 'executable', 'exit', 'flags', 'float_info', 'float_repr_style',
 'getcheckinterval', 'getdefaultencoding', 'getdlopenflags',
 'getfilesystemencoding', 'getobjects', 'getprofile', 'getrecursionlimit',
 'getrefcount', 'getsizeof', 'getswitchinterval', 'gettotalrefcount',
 'gettrace', 'hash_info', 'hexversion', 'implementation', 'int_info',
 'intern', 'maxsize', 'maxunicode', 'meta_path', 'modules', 'path',
 'path_hooks', 'path_importer_cache', 'platform', 'prefix', 'ps1',
 'setcheckinterval', 'setdlopenflags', 'setprofile', 'setrecursionlimit',
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
'setswitchinterval', 'settrace', 'stderr', 'stdin', 'stdout',  
'thread_info', '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__', 'a', 'fib', 'fibo', 'sys']
```

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

`dir()` tidak mencantumkan nama fungsi dan variabel bawaan. Jika Anda ingin daftar itu, mereka didefinisikan dalam modul standar `builtins`:

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

## 6.4 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):

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

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

Berkas `__init__.py` diperlukan untuk membuat Python memperlakukan direktori yang berisi file sebagai paket. Ini mencegah direktori dengan nama umum, seperti `string`, menyembunyikan modul valid yang muncul kemudian pada jalur pencarian modul. Dalam kasus yang paling sederhana, file: `__init__.py` dapat berupa file kosong, tetapi juga dapat menjalankan kode inisialisasi untuk paket atau mengatur variabel `__all__`, dijelaskan nanti.

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 mengekspor 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

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`.

Anda juga dapat menulis impor relatif, dengan bentuk `from module import name` pada pernyataan impor. Impor ini menggunakan titik-titik di awalan untuk menunjukkan paket saat ini dan induk yang terlibat dalam impor relatif. Dari modul `surround` misalnya, Anda dapat menggunakan:

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

Perhatikan bahwa impor relatif didasarkan pada nama modul saat ini. Karena nama modul utama selalu `"__main__"`, modul yang dimaksudkan untuk digunakan sebagai modul utama aplikasi Python harus selalu menggunakan impor absolut.

### 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

Sejauh ini kami telah menemukan dua cara penulisan nilai: *expression statements* dan fungsi `print()`. (Cara ketiga menggunakan `write()` metode objek berkas; berkas standar keluaran dapat dirujuk sebagai `sys.stdout`. Lihat Referensi Pustaka untuk informasi lebih lanjut tentang ini.)

Seringkali Anda akan menginginkan lebih banyak kontrol atas pemformatan keluaran Anda daripada sekadar mencetak nilai yang dipisahkan ruang. Ada beberapa cara untuk memformat keluaran.

- Untuk menggunakan *formatted string literals*, mulailah string dengan `f` atau `F` sebelum tanda kutip pembuka atau tanda kutip tiga. Di dalam string ini, Anda bisa menulis ekspresi Python antara karakter `{` dan `}` yang dapat merujuk ke variabel atau nilai literal.

```
>>> year = 2016
>>> event = 'Referendum'
>>> f'Results of the {year} {event}'
'Results of the 2016 Referendum'
```

- Metode `str.format()` dari string membutuhkan lebih banyak upaya manual. Anda masih akan menggunakan `{}` dan `}` untuk menandai di mana variabel akan diganti dan dapat memberikan arahan pemformatan terperinci, tetapi Anda juga harus memberikan informasi yang akan diformat.

```
>>> yes_votes = 42_572_654
>>> no_votes = 43_132_495
>>> percentage = yes_votes / (yes_votes + no_votes)
>>> '{:-9} YES votes  {:.2%}'.format(yes_votes, percentage)
' 42572654 YES votes  49.67%'
```

- Akhirnya, Anda dapat melakukan semua string yang menangani diri Anda sendiri dengan menggunakan operasi *slicing* string dan *concatenation* untuk membuat tata letak yang dapat Anda bayangkan. Tipe string memiliki beberapa metode yang melakukan operasi yang berguna untuk string *padding* ke lebar kolom yang diberikan.

Ketika Anda tidak membutuhkan keluaran yang menarik tetapi hanya ingin tampilan cepat dari beberapa variabel untuk keperluan debugging, Anda dapat mengonversi nilai apa pun menjadi string dengan fungsi `repr()` atau `str()`.

Fungsi `str()` dimaksudkan untuk mengembalikan representasi nilai-nilai yang terbaca oleh manusia, sementara `repr()` dimaksudkan untuk menghasilkan representasi yang dapat dibaca oleh penerjemah (atau akan memaksa `SyntaxError` jika tidak ada sintaks yang setara). Untuk objek yang tidak memiliki representasi khusus untuk konsumsi manusia, `str()` akan mengembalikan nilai yang sama dengan `repr()`. Banyak nilai, seperti angka atau struktur seperti daftar dan kamus, memiliki representasi yang sama menggunakan kedua fungsi tersebut. String, khususnya, memiliki dua representasi berbeda.

Beberapa contoh:

```
>>> s = 'Hello, world.'
>>> str(s)
'Hello, world.'
>>> repr(s)
"'Hello, world.'"
>>> str(1/7)
'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'))'
```

Modul `string` berisi kelas `Template` yang menawarkan cara lain untuk mengganti nilai menjadi string, menggunakan penampung seperti `$x` dan menggantinya dengan nilai-nilai dari *dictionary*, tetapi menawarkan kontrol format yang jauh lebih sedikit.

## 7.1.1 Literal String Terformat

Formatted string literals (juga disebut f-string) memungkinkan Anda menyertakan nilai ekspresi Python di dalam string dengan mengawali string dengan `f` atau `F` dan menulis ekspresi sebagai `{expression}`.

Penentu format opsional dapat mengikuti ekspresi. Ini memungkinkan kontrol yang lebih besar atas bagaimana nilai diformat. Contoh berikut ini pembulatan `pi` ke tiga tempat setelah desimal:

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

Melewatkan bilangan bulat setelah `:` akan menyebabkan *field* itu menjadi jumlah minimum lebar karakter. Ini berguna untuk membuat kolom berbaris.



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

Pengubah lain dapat digunakan untuk mengonversi nilai sebelum diformat. `'!a'` berlaku `ascii()`, `'!s'` berlaku `str()`, dan `'!r'` berlaku `repr()`:

```
>>> animals = 'eels'
>>> print(f'My hovercraft is full of {animals}.')
My hovercraft is full of eels.
>>> print(f'My hovercraft is full of {animals!r}.')
My hovercraft is full of 'eels'.
```

Untuk referensi tentang spesifikasi format ini, lihat panduan referensi untuk `formatpec`.

## 7.1.2 Metode String format()

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

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

Tanda kurung dan karakter di dalamnya (disebut *fields format*) diganti dengan objek yang diteruskan ke metode `str.format()`. Angka dalam tanda kurung dapat digunakan untuk merujuk ke posisi objek yang dilewatkan ke dalam metode `str.format()`.

```
>>> 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.
```

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.

Sebagai contoh, baris-baris berikut menghasilkan kumpulan kolom yang disejajarkan rapi memberikan bilangan bulat dan kotak dan kubusnya:

```
>>> 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
```

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

### 7.1.3 Pemformatan String Manual

Inilah tabel kotak dan kubus yang sama, yang diformat secara manual:

```
>>> for x in range(1, 11):
...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
...     # Note use of 'end' 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
```

(Perhatikan bahwa satu spasi di antara setiap kolom ditambahkan dengan cara `print()` berfungsi: selalu menambah spasi di antara argumennya.)

Metode `str.rjust()` dari objek string merata-kanan-kan sebuah string dalam bidang dengan lebar tertentu dengan menambahkannya dengan spasi di sebelah kiri. Ada metode serupa `str.ljust()` dan `str.center()`. Metode ini tidak menulis apa pun, mereka hanya mengembalikan string baru. Jika string input terlalu panjang, mereka tidak memotongnya, tetapi mengembalikannya tidak berubah; ini akan mengacaukan tata letak kolom Anda, tetapi itu biasanya lebih baik daripada alternatif, yang akan berbohong tentang nilai. (Jika Anda benar-benar menginginkan pemotongan, Anda selalu dapat menambahkan operasi *slice*, seperti pada `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'
```

## 7.1.4 Pemformatan string lama

Operator `%` (modulo) juga dapat digunakan untuk pemformatan string. Diberikan `'string' % values`, instansiasi dari `%` in `string` diganti dengan nol atau elemen yang lebih dari `values`. Operasi ini umumnya dikenal sebagai interpolasi string. Sebagai contoh:

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

Informasi lebih lanjut dapat ditemukan di bagian `old-string-formatting`.

## 7.2 Membaca dan Menulis Berkas

`open()` mengembalikan sebuah *file object*, dan paling umum digunakan dengan dua argumen: `open(filename, mode)`.

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

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.

Biasanya, file dibuka di *text mode*, yang berarti, Anda membaca dan menulis string dari dan ke file, yang dikodekan dalam pengkodean tertentu. Jika pengkodean tidak ditentukan, standarnya adalah bergantung platform (lihat `:func: open`). `'b'` ditambahkan ke mode membuka berkas di *binary mode*: sekarang data dibaca dan ditulis dalam bentuk objek byte. Mode ini harus digunakan untuk semua file yang tidak mengandung teks.

Dalam mode teks, standar saat membaca adalah mengonversi akhir baris spesifik platform (`\n` pada Unix, `\r\n` pada Windows) menjadi hanya `\n`. Saat menulis dalam mode teks, defaultnya adalah mengonversi kemunculan `\n` kembali ke akhir baris spesifik platform. Modifikasi di balik layar ini untuk mengarsipkan data baik untuk file teks, tetapi akan merusak data biner seperti itu di JPEG atau berkas EXE. Berhati-hatilah untuk menggunakan mode biner saat membaca dan menulis file seperti itu.

Ini adalah praktik yang baik untuk menggunakan kata kunci `with` saat berurusan dengan objek file. Keuntungannya adalah bahwa file ditutup dengan benar setelah rangkaiannya selesai, bahkan jika suatu pengecualian muncul di beberapa titik. Menggunakan `with` juga jauh lebih pendek daripada penulisan yang setara `try`-blok `finally`:

```
>>> with open('workfile') as f:
...     read_data = f.read()
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
>>> # We can check that the file has been automatically closed.
>>> f.closed
True
```

Jika anda tidak menggunakan kata kunci `with`, maka anda harus memanggil `f.close()` untuk menutup *file* dan membebaskan sumber daya sistem yang digunakan secara langsung.

**Peringatan:** Memanggil `f.write()` tanpa menggunakan kata kunci `with` atau memanggil `f.close()` **dapat** menyebabkan argumen-argumen dari `f.write()` tidak dituliskan ke dalam *disk* secara utuh, meskipun program berhenti dengan sukses.

Setelah objek file ditutup, baik dengan pernyataan `with` atau dengan memanggil `f.close()`, upaya untuk menggunakan objek file akan secara otomatis gagal.

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

## 7.2.1 Metode Objek Berkas

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

Untuk membaca konten file, panggil `f.read(size)`, yang membaca sejumlah kuantitas data dan mengembalikannya sebagai string (dalam mode teks) atau objek byte (dalam mode biner). *size* adalah argumen numerik opsional. Ketika *size* dihilangkan atau negatif, seluruh isi file akan dibaca dan dikembalikan; itu masalah Anda jika file tersebut dua kali lebih besar dari memori mesin Anda. Kalau tidak, paling banyak *size* karakter (dalam mode teks) atau *size* byte (dalam mode biner) dibaca dan dikembalikan. Jika akhir file telah tercapai, `f.read()` akan mengembalikan string kosong ('').

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

`f.readline()` membaca satu baris dari file; karakter baris baru (`\n`) dibiarkan di akhir string, dan hanya dihapus pada baris terakhir file jika file tidak berakhir pada baris baru. Ini membuat nilai pengembalian tidak ambigu; jika `f.readline()` mengembalikan string kosong, akhir file telah tercapai, sementara baris kosong diwakili oleh `' \n '`, string yang hanya berisi satu baris baru.

```
>>> 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, end='')

```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
...
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)` menulis konten *string* ke berkas, mengembalikan jumlah karakter yang ditulis.

```
>>> f.write('This is a test\n')
15
```

Jenis objek lain perlu dikonversi -- baik menjadi string (dalam mode teks) atau objek byte (dalam mode biner) -- sebelum menulisnya:

```
>>> value = ('the answer', 42)
>>> s = str(value) # convert the tuple to string
>>> f.write(s)
18
```

`f.tell()` mengembalikan integer yang memberikan posisi objek file saat ini dalam berkas yang direpresentasikan sebagai jumlah byte dari awal berkas ketika dalam mode biner dan angka buram *opaque* ketika dalam mode teks.

Untuk mengubah posisi objek file, gunakan `f.seek(offset, whence)`. Posisi dihitung dari menambahkan *offset* ke titik referensi; titik referensi dipilih oleh argumen *whence*. Nilai *A whence* dari 0 mengukur dari awal berkas, 1 menggunakan posisi file saat ini, dan 2 menggunakan akhir file sebagai titik referensi. *whence* dapat dihilangkan dan default ke 0, menggunakan awal file sebagai titik referensi.

```
>>> f = open('workfile', 'rb+')
>>> f.write(b'0123456789abcdef')
16
>>> f.seek(5) # Go to the 6th byte in the file
5
>>> f.read(1)
b'5'
>>> f.seek(-3, 2) # Go to the 3rd byte before the end
13
>>> f.read(1)
b'd'
```

Dalam file teks (yang dibuka tanpa `b` dalam mode string), hanya mencari relatif ke awal file yang diizinkan (pencualian sedang mencari sampai akhir file dengan `seek(0, 2)`) dan satu-satunya nilai *offset* yang valid adalah yang dikembalikan dari `f.tell()`, atau nol. Nilai *offset* lainnya menghasilkan perilaku tidak terdefinisi.

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"]'
```

Varian lain dari fungsi `dumps()`, disebut `dump()`, dengan mudah membuat serialisasi objek menjadi *term: text file*. Jadi jika `f` adalah objek *text file* dibuka untuk menulis, kita dapat melakukan ini:

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

Untuk menerjemahkan *decode* objek lagi, jika `f` adalah objek *text file* yang telah dibuka untuk membaca:

```
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
```

Pengurai *parser* mengulangi baris yang menyinggung dan menampilkan sedikit 'arrow' yang menunjuk pada titik paling awal di baris di mana kesalahan terdeteksi. Kesalahan disebabkan oleh (atau setidaknya terdeteksi pada) token *preceding* panah: dalam contoh, kesalahan terdeteksi pada fungsi `print()`, karena titik dua (':') hilang sebelum itu. Nama file dan nomor baris dicetak sehingga Anda tahu ke mana harus mencari kalau-kalau masukan berasal dari skrip.

### 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: division 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: Can't convert 'int' object to str implicitly
```

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.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

builtin-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(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.



Pernyataan `try` mungkin memiliki lebih dari satu klausa *except*, untuk menentukan penanganan dari berbagai pengecualian. Paling banyak satu penanganan akan dieksekusi. Penanganan hanya menangani pengecualian yang terjadi pada klausa *try* yang sesuai, bukan pada penanganan lain yang sama pernyataan *try*. Klausa *except* dapat menyebutkan beberapa pengecualian sebagai tanda kurung *tuple*, misalnya:

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

Kelas dalam klausa *except* kompatibel dengan pengecualian jika itu adalah kelas yang sama atau kelas basisnya (tapi bukan sebaliknya --- sebuah klausa *except* dari daftar kelas turunan tidak kompatibel dengan kelas). Misalnya, kode berikut akan mencetak B, C, D dalam urutan itu:

```
class B(Exception):
    pass

class C(B):
    pass

class D(C):
    pass

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

Perhatikan bahwa jika klausa *except* dibalik (dengan *except B* dahulu), itu akan dicetak B, B, B --- pencocokan pertama klausa *except* dipicu.

Klausa *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 OSError as err:
    print("OS error: {0}".format(err))
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 OSError:
        print('cannot open', arg)
    else:
        print(arg, 'has', len(f.readlines()), 'lines')
        f.close()
```

Penggunaan klausa `else` lebih baik daripada menambahkan kode tambahan ke klausa `try` karena menghindari secara tidak sengaja menangkap pengecualian yang tidak dimunculkan oleh kode yang dilindungi oleh pernyataan `try ...`:keyword: *!except*.

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

Klausa `except` dapat menentukan variabel setelah nama pengecualian. Variabel terikat pada *instance* pengecualian dengan argumen yang disimpan dalam `instance.args`. Untuk kenyamanan, *instance* pengecualian mendefinisikan `__str__()` sehingga argumen dapat dicetak langsung tanpa harus merujuk `.args`. Seseorang juga dapat membuat instansiasi pengecualian terlebih dahulu sebelum menimbulkannya dan menambahkan atribut apa pun yang diinginkan.

```
>>> try:
...     raise Exception('spam', 'eggs')
... 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,
...                          # but may be overridden in exception subclasses
...     x, y = inst.args     # unpack args
...     print('x =', x)
...     print('y =', y)
...
<class 'Exception'>
('spam', 'eggs')
('spam', 'eggs')
x = spam
y = eggs
```

Jika pengecualian memiliki argumen, mereka dicetak sebagai bagian terakhir ('detail') dari pesan untuk pengecualian yang tidak ditangani.

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 err:
...     print('Handling run-time error:', err)
...
Handling run-time error: division 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
```

Satu-satunya argumen untuk `raise` menunjukkan pengecualian yang dimunculkan. Ini harus berupa *instance* pengecualian atau kelas pengecualian (kelas yang berasal dari `Exception`). Jika kelas pengecualian dikirimkan, itu akan secara implisit diinstansiasi dengan memanggil pembangunnya *constructor* tanpa argumen:

```
raise ValueError # shorthand for 'raise ValueError()'
```

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

Program dapat memberi nama pengecualian mereka sendiri dengan membuat kelas pengecualian baru (lihat tut-class untuk informasi lebih lanjut tentang kelas Python). Pengecualian biasanya berasal dari kelas `Exception`, baik secara langsung atau tidak langsung.

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:
        expression -- input expression in which the error occurred
        message -- explanation of the error
    """

    def __init__(self, expression, message):
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

        self.expression = expression
        self.message = message

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

    Attributes:
        previous -- state at beginning of transition
        next -- attempted new state
        message -- explanation of why the specific transition is not allowed
    """

    def __init__(self, previous, next, message):
        self.previous = previous
        self.next = next
        self.message = message

```

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!
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
KeyboardInterrupt

```

Jika ada klausa `finally`, klausa untuk `finally` akan dijalankan sebagai tugas terakhir sebelum pernyataan untuk `try` selesai. Klausa untuk `finally` dapat berjalan baik atau tidak apabila pernyataan `try` menghasilkan suatu pengecualian. Poin-poin berikut membahas kasus yang lebih kompleks saat pengecualian terjadi:

- Jika pengecualian terjadi selama eksekusi klausa untuk `try`, maka pengecualian tersebut dapat ditangani oleh klausa `except`. Jika pengecualian tidak ditangani oleh klausa `except`, maka pengecualian dimunculkan kembali setelah klausa `finally` dieksekusi.
- Pengecualian dapat terjadi selama pelaksanaan klausa `except` atau `else`. Sekali lagi, pengecualian akan muncul kembali setelah klausa `finally` telah dieksekusi.
- Jika pernyataan klausa untuk `try` mencapai klausa `break`, `continue` atau `return`, maka pernyataan untuk klausa `finally` akan dieksekusi sebelum `break`, `continue` atau `return` dieksekusi.
- Jika klausa untuk `finally` telah menyertakan pernyataan `return`, nilai yang dikembalikan akan menjadi salah satu dari pernyataan untuk `finally` dan dari klausa `return`, bukan nilai dari `try` pernyataan untuk `return`.

Sebagai contoh:

```
>>> def bool_return():
...     try:
...         return True
...     finally:
...         return False
...
>>> bool_return()
False
```

Contoh yang lebih rumit:

```
>>> 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 2.0
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>
  File "<stdin>", line 3, in divide
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

Seperti yang Anda lihat, klausa `finally` dieksekusi dalam peristiwa apa pun. `TypeError` yang ditimbulkan dengan membagi dua string tidak ditangani oleh klausa `except` dan karenanya kembali muncul setelah klausa `finally` telah dieksekusi.

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, end="")
```

Masalah dengan kode ini adalah bahwa ia membiarkan berkas terbuka untuk jumlah waktu yang tidak ditentukan setelah bagian kode ini selesai dieksekusi. Ini bukan masalah dalam skrip sederhana, tetapi bisa menjadi masalah untuk aplikasi yang lebih besar. Pernyataan `with` memungkinkan objek seperti berkas digunakan dengan cara yang memastikan mereka selalu dibersihkan secepatnya dan dengan benar.

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

Setelah pernyataan dieksekusi, file *f* selalu ditutup, bahkan jika ada masalah saat pemrosesan baris-baris. Objek yang, seperti berkas-berkas, memberikan tindakan pembersihan yang telah ditentukan, akan menunjukkan ini dalam dokumentasinya.

*Classes* atau kelas-kelas menyediakan sarana untuk menggabungkan data dan fungsionalitas bersama. Membuat sebuah *class* baru menghasilkan objek dengan *type* baru, memungkinkan dibuat *instance* baru dari tipe itu. Setiap *instance* dari *class* dapat memiliki atribut yang melekat padanya untuk mempertahankan kondisinya. *Instance* dari sebuah *class* juga dapat memiliki metode (ditentukan oleh *class*) untuk memodifikasi kondisinya.

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 *Variabel Privat*), 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!<sup>1</sup>

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`.

*Namespace* dibuat pada saat yang berbeda dan memiliki masa hidup yang berbeda. *Namespace* yang berisi nama-nama bawaan dibuat ketika interpreter Python dimulai, dan tidak pernah dihapus. *Namespace* global untuk modul dibuat ketika definisi modul dibaca; biasanya, *namespace* modul juga bertahan hingga *interpreter* berhenti. Pernyataan yang dieksekusi oleh pemanggilan *interpreter* tingkat atas, baik membaca dari file skrip atau secara interaktif, dianggap sebagai bagian dari modul yang disebut `__main__`, sehingga mereka memiliki *namespace* global sendiri. (Nama bawaan sebenarnya juga hidup dalam modul; ini disebut *builtins*.)

*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

---

<sup>1</sup> 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*.



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 3 atau 4 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

Jika sebuah nama dinyatakan global, maka semua referensi dan penugasan langsung ke lingkup *scope* tengah yang berisi nama global modul. Untuk mengembalikan variabel yang ditemukan di luar cakupan terdalam, pernyataan `nonlocal` dapat digunakan; jika tidak dideklarasikan nonlokal, variabel-variabel itu hanya baca-saja (upaya untuk menulis ke variabel seperti itu hanya akan membuat variabel lokal *baru* dalam cakupan terdalam, membiarkan variabel luar yang dinamai identik tidak berubah).

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.)

Sebuah kekhasan khusus dari Python adalah bahwa -- jika tidak ada pernyataan `global` atau pernyataan `nonlocal` yang berlaku -- pemberian nilai untuk nama selalu masuk ke ruang lingkup terdalam. Pemberian nilai tidak menyalin data --- mereka hanya mengikat nama ke objek. Hal yang sama berlaku untuk penghapusan: pernyataan `del x` menghapus pengikatan `x` dari *namespace* yang dirujuk oleh lingkup *scope* lokal. Bahkan, semua operasi yang memperkenalkan nama-nama baru menggunakan lingkup lokal: khususnya, pernyataan `import` dan definisi fungsi mengikat modul atau nama fungsi di lingkup lokal.

Pernyataan `global` dapat digunakan untuk menunjukkan bahwa variabel tertentu hidup dalam lingkup global dan harus kembali ke sana; pernyataan `nonlocal` menunjukkan bahwa variabel tertentu hidup dalam cakupan terlampir dan harus dikembalikan ke sana.

## 9.2.1 Contoh Lingkup Scopes dan Ruang Nama Namespaces

Ini adalah contoh yang menunjukkan cara mereferensikan lingkup *scopes* dan ruang nama *namespaces* yang berbeda, dan bagaimana `global` dan `nonlocal` memengaruhi pengikatan variabel:

```
def scope_test():
    def do_local():
        spam = "local spam"

    def do_nonlocal():
        nonlocal spam
        spam = "nonlocal spam"
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
def do_global():
    global spam
    spam = "global spam"

spam = "test spam"
do_local()
print("After local assignment:", spam)
do_nonlocal()
print("After nonlocal assignment:", spam)
do_global()
print("After global assignment:", spam)

scope_test()
print("In global scope:", spam)
```

Keluaran dari contoh kode adalah:

```
After local assignment: test spam
After nonlocal assignment: nonlocal spam
After global assignment: nonlocal spam
In global scope: global spam
```

Perhatikan bagaimana pemberian nilai *local* (yang bawaan) tidak mengubah *scope\_tests* pengikatan *spam*. Pemberian nilai *nonlocal* mengubah *scope\_test*'s pengikatan *spam*, dan pemberian nilai *global* mengubah pengikatan level modul.

Anda juga dapat melihat bahwa tidak ada pengikatan sebelumnya untuk *spam* sebelum pemberian nilai *global*.

## 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...

Sebenarnya, Anda mungkin sudah menebak jawabannya: hal khusus tentang metode adalah objek *instance* dilewatkan sebagai argumen pertama dari fungsi. Dalam contoh kita, panggilan `x.f()` persis sama dengan `MyClass.f(x)`. Secara umum, memanggil metode dengan daftar argumen *n* setara dengan memanggil fungsi yang sesuai dengan daftar argumen yang dibuat dengan menyisipkan objek contoh metode sebelum argumen pertama.

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

Jika nama atribut yang sama muncul di kedua *instance* dan di kelas, maka pencarian atribut memprioritaskan *instance*:

```
>>> class Warehouse:
    purpose = 'storage'
    region = 'west'

>>> w1 = Warehouse()
>>> print(w1.purpose, w1.region)
storage west
>>> w2 = Warehouse()
>>> w2.region = 'east'
>>> print(w2.purpose, w2.region)
storage east
```

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
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
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, penca-

rian 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 menerima 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`.
- Gunakan `issubclass()` untuk memeriksa warisan kelas: `issubclass(bool, int)` adalah `True` karena `bool` adalah subkelas dari `int`. Namun, `issubclass(float, int)` adalah `False` karena `float` bukan subkelas dari `int`.

## 9.5.1 Pewarisan Berganda

Python mendukung bentuk pewarisan berganda juga. Definisi kelas dengan beberapa kelas dasar terlihat seperti ini:

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

Untuk sebagian besar tujuan, dalam kasus paling sederhana, Anda dapat menganggap pencarian atribut yang diwarisi dari kelas induk sebagai kedalaman-pertama *depth-first*, kiri-ke-kanan, bukan mencari dua kali di kelas yang sama di mana ada tumpang tindih dalam hierarki. Dengan demikian, jika atribut tidak ditemukan di `DerivedClassName`, itu dicari di `Base1`, kemudian (secara rekursif) di kelas dasar dari `Base1`, dan jika tidak ditemukan di sana, itu dicari di `Base2`, dan seterusnya.

Faktanya, ini sedikit lebih kompleks dari itu; urutan resolusi metode berubah secara dinamis untuk mendukung pemanggilan kooperatif ke `super()`. Pendekatan ini dikenal dalam beberapa bahasa warisan ganda sebagai metode panggilan-berikutnya *call-next-method* dan lebih berdaya daripada panggilan `super` yang ditemukan dalam bahasa warisan tunggal.

Urutan dinamis diperlukan karena semua kasus pewarisan berganda menunjukkan satu atau lebih hubungan intan *diamond relationships* (di mana setidaknya satu dari kelas induk dapat diakses melalui beberapa jalur dari kelas bawah). Sebagai contoh, semua kelas mewarisi dari `object`, jadi segala kasus pewarisan berganda menyediakan lebih dari satu jalur untuk mencapai `:class: 'object'`. Untuk menjaga agar kelas dasar tidak diakses lebih dari sekali, algoritma dinamis linier mengurutkan urutan pencarian dengan cara yang mempertahankan urutan kiri-ke-kanan yang ditentukan dalam setiap kelas, yang memanggil setiap induk hanya sekali, dan itu monoton (artinya suatu kelas dapat di-subklas-kan tanpa memengaruhi urutan prioritas orang tuanya). Secara bersama-sama, properti ini memungkinkan untuk merancang kelas yang andal dan dapat diperluas dengan banyak pewarisan. Untuk detail lebih lanjut, lihat <https://www.python.org/download/releases/2.3/mro/>.



## 9.6 Variabel Privat

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 menempa metode tanpa memutus panggilan metode *in-traclass*. 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*.

Perhatikan bahwa kode yang dilewatkan ke `exec()` atau `eval()` tidak menganggap nama kelas *classname* dari kelas yang dipanggil sebagai kelas saat ini; ini mirip dengan efek pernyataan `global`, yang efeknya juga terbatas pada kode yang dikompilasi-byte *byte-compiled* bersama. Pembatasan yang sama berlaku untuk `getattr()`, `setattr()` dan `delattr()`, serta saat mereferensikan `__dict__` secara langsung.

## 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.

Objek metode *instance* memiliki atribut, juga: `m.__self__` adalah objek instan dengan metode `m()`, dan `m.__func__` adalah objek fungsi yang sesuai dengan metode tersebut.

## 9.8 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, end='')
```

Gaya akses ini jelas, ringkas, dan nyaman. Penggunaan *iterator* meliputi *pervades* dan menyatukan Python. Di belakang layar, pernyataan `for` memanggil `iter()` pada objek penampung *container*. Fungsi mengembalikan objek *iterator* yang mendefinisikan metode `__next__()` yang mengakses elemen dalam penampung *container* satu per satu. Ketika tidak ada lagi elemen, `__next__()` memunculkan pengecualian `StopIteration` yang memberi tahu perulangan `for` untuk mengakhiri. Anda dapat memanggil metode `__next__()` menggunakan `next()` fungsi bawaan; contoh ini menunjukkan cara kerjanya:

```
>>> s = 'abc'
>>> it = iter(s)
>>> it
<iterator object at 0x00A1DB50>
>>> next(it)
'a'
>>> next(it)
'b'
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
>>> next(it)
'c'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    next(it)
StopIteration
```

Setelah melihat mekanisme di balik protokol *iterator*, mudah untuk menambahkan perilaku *iterator* ke kelas Anda. Definisikan metode `__iter__()` yang mengembalikan objek dengan metode `__next__()`. Jika kelas mendefinisikan `__next__()`, maka `__iter__()` bisa langsung mengembalikan `self`:

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

## 9.9 Pembangkit Generator

*Generators* adalah sebuah tool yang sederhana dan simpel untuk membuat sebuah iterasi. Itu ditulis seperti fungsi biasa tapi menggunakan pernyataan `yield` setiap kali ingin mengembalikan sebuah data. Tiap kali `next()` itu dipanggil, generators tersebut akan melanjutkan di mana hal itu berhenti (itu akan mengingat semua nilai dan pernyataan mana yang terakhir dieksekusi). Sebuah contoh menunjukkan bahwa generator sangat 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
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
l
o
g
```

Apa pun yang dapat dilakukan dengan pembangkit *generator* juga dapat dilakukan dengan *iterator* berbasis kelas seperti yang dijelaskan pada bagian sebelumnya. Apa yang membuat pembangkit *generator* sangat kompak adalah bahwa metode `__iter__()` dan `__next__()` dibuat secara otomatis.

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.

## 9.10 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

>>> 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:\\Python38'
>>> 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')
'archive.db'
>>> shutil.move('/build/executables', 'installdir')
'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']
```

Modul `argparse` menyediakan mekanisme yang lebih canggih untuk memproses argumen baris perintah. Script berikut mengekstrak satu atau lebih nama file dan sejumlah baris opsional untuk ditampilkan:

```
import argparse

parser = argparse.ArgumentParser(prog = 'top',
                                description = 'Show top lines from each file')
parser.add_argument('filenames', nargs='+')
parser.add_argument('-l', '--lines', type=int, default=10)
args = parser.parse_args()
print(args)
```

Ketika dijalankan pada baris perintah dengan `python top.py --lines=5 alpha.txt beta.txt`, skrip mengatur `args.lines` menjadi 5 dan `args.filenames` menjadi `['alpha.txt', 'beta.txt']`.

## 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)
0.70710678118654757
>>> math.log(1024, 2)
10.0
```

Modul `random` menyediakan alat untuk membuat pilihan acak:

```
>>> import random
>>> random.choice(['apple', 'pear', 'banana'])
'apple'
>>> random.sample(range(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
```

Modul `statistics` menghitung sifat statistik dasar (rata-rata, median, varian, dll.) dari data numerik:

```
>>> import statistics
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> statistics.mean(data)
1.6071428571428572
>>> statistics.median(data)
1.25
>>> statistics.variance(data)
1.3720238095238095
```

Proyek SciPy <<https://scipy.org>> memiliki banyak modul lain untuk perhitungan numerik.

## 10.7 Akses internet

Ada sejumlah modul untuk mengakses internet dan memproses protokol internet. Dua yang paling sederhana adalah `urllib.request` untuk mengambil data dari URL dan `smtplib` untuk mengirim email:

```
>>> from urllib.request import urlopen
>>> with urlopen('http://tycho.usno.navy.mil/cgi-bin/timer.pl') as response:
...     for line in response:
...         line = line.decode('utf-8') # Decoding the binary data to text.
...         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

Format pengarsipan dan kompresi data umum didukung langsung oleh modul-modul yang ada antara lain: `zlib`, `gzip`, `bz2`, `lzma`, `zipfile` dan `tarfile`.

```
>>> import zlib
>>> s = b'witch which has which witches wrist watch'
>>> len(s)
41
>>> t = zlib.compress(s)
>>> len(t)
37
>>> zlib.decompress(t)
b'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 `doctest` menyediakan alat untuk memindai modul dan memvalidasi tes yang tertanam dalam dokumen program. Konstruksi pengujian sesederhana 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) / len(values)
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
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:

- Modul `xmlrpc.client` dan `xmlrpc.server` membuat penerapan panggilan prosedur jarak jauh menjadi tugas yang hampir sepele. Terlepas dari nama-nama modul, tidak diperlukan pengetahuan atau penanganan XML secara langsung.
- Paket `email` adalah pustaka untuk mengelola pesan email, termasuk MIME dan lainnya [RFC 2822](#) dokumen pesan berbasis. Tidak seperti `smtplib` dan `poplib` yang benar-benar mengirim dan menerima pesan, paket email memiliki toolset lengkap untuk membangun atau mendekodekan struktur pesan kompleks (termasuk lampiran) dan untuk mengimplementasikan pengkodean internet dan protokol header.
- Paket `json` menyediakan dukungan kuat untuk mengurai format pertukaran data populer ini. Modul `csv` mendukung pembacaan dan penulisan berkas secara langsung dalam format Nilai Terpisah-Koma atau CSV, umumnya didukung oleh database dan spreadsheet. Pemrosesan XML didukung oleh paket `xml.etree.ElementTree`, `xml.dom` dan `xml.sax`. Bersama-sama, modul dan paket ini sangat menyederhanakan pertukaran data antara aplikasi Python dan alat lainnya.
- Modul `sqlite3` adalah pembungkus untuk pustaka basis data SQLite, menyediakan basis data persisten yang dapat diperbarui dan diakses menggunakan sintaks SQL yang sedikit tidak standar.
- 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

Modul `reprlib` menyediakan versi `repr()` yang disesuaikan untuk tampilan yang disingkat dari wadah *containers* yang besar atau sangat bersarang

```
>>> import reprlib
>>> reprlib.repr(set('supercalifragilisticexpialidocious'))
'{'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
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
... a list of strings instead of one big string with newlines to separate
... the wrapped lines."""
...
>>> 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 memungkinkannya 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 = 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

with open('myfile.zip', 'rb') as f:
    data = f.read()

start = 0
for i in range(3):
    # show the first 3 file headers
    start += 14
    fields = struct.unpack('<IIIHH', 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.

Sementara alat-alat itu berdaya, kesalahan desain kecil dapat menyebabkan masalah yang sulit untuk direproduksi. Jadi, pendekatan yang lebih disukai untuk koordinasi tugas adalah untuk memusatkan semua akses ke sumber daya dalam satu utas *thread* dan kemudian menggunakan modul *queue* untuk menyuapi utas *thread* tersebut dengan permintaan dari utas lainnya. Aplikasi yang menggunakan objek *Queue* untuk komunikasi dan koordinasi antar-utas *inter-thread* lebih mudah untuk dirancang, lebih mudah dibaca, dan lebih dapat diandalkan.

## 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, socket, 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:/python38/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. Perbedaanannya menjadi signifikan jika hasilnya dibulatkan ke sen terdekat:

```
>>> from decimal import *
>>> round(Decimal('0.70') * Decimal('1.05'), 2)
Decimal('0.74')
>>> round(.70 * 1.05, 2)
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')
```



---

## Lingkungan dan Paket Virtual

---

### 12.1 Pengantar

Aplikasi Python akan sering menggunakan paket dan modul yang tidak datang sebagai bagian dari pustaka standar. Aplikasi kadang-kadang membutuhkan versi pustaka tertentu, karena aplikasi mungkin mensyaratkan bug tertentu telah diperbaiki atau aplikasi dapat ditulis menggunakan versi usang dari antarmuka pustaka.

Ini berarti tidak mungkin bagi satu instalasi Python untuk memenuhi persyaratan setiap aplikasi. Jika aplikasi A membutuhkan versi 1.0 dari modul tertentu tetapi aplikasi B membutuhkan versi 2.0, maka persyaratannya bertentangan dan menginstal versi 1.0 atau 2.0 akan membuat satu aplikasi tidak dapat berjalan.

Solusi untuk masalah ini adalah membuat *virtual environment*, sebuah struktur direktori mandiri yang berisi instalasi Python untuk versi tertentu dari Python, serta sejumlah paket tambahan.

Aplikasi yang berbeda kemudian dapat menggunakan lingkungan virtual yang berbeda. Untuk menyelesaikan contoh sebelumnya dari persyaratan yang saling bertentangan, aplikasi A dapat memiliki lingkungan virtual sendiri dengan versi 1.0 yang diinstal sementara aplikasi B memiliki lingkungan virtual lain dengan versi 2.0. Jika aplikasi B membutuhkan pustaka ditingkatkan ke versi 3.0, ini tidak akan mempengaruhi lingkungan aplikasi A.

### 12.2 Menciptakan Lingkungan Virtual

Modul yang digunakan untuk membuat dan mengelola lingkungan virtual disebut `venv`. `venv` biasanya akan menginstal versi Python terbaru yang Anda miliki. Jika Anda memiliki beberapa versi Python di sistem Anda, Anda dapat memilih versi Python tertentu dengan menjalankan `python3` atau versi mana pun yang Anda inginkan.

Untuk membuat lingkungan virtual, tentukan direktori tempat Anda ingin meletakkannya, dan jalankan modul `venv` sebagai skrip dengan jalur direktori:

```
python3 -m venv tutorial-env
```

Ini akan membuat direktori `tutorial-env` jika tidak ada, dan juga membuat direktori di dalamnya yang berisi salinan interpreter Python, pustaka standar, dan berbagai berkas pendukung.

Lokasi direktori yang umum dipakai untuk lingkungan virtual adalah `.venv`. Nama ini membuat direktori biasanya tersembunyi di shell Anda dan dengan demikian terpencil sambil memberikan nama yang menjelaskan mengapa direktori itu ada. Ini juga mencegah bentrok dengan berkas definisi variabel lingkungan `.env` yang didukung beberapa peralatan.

Setelah Anda membuat lingkungan virtual, Anda dapat mengaktifkannya.

Di Windows, operasikan:

```
tutorial-env\Scripts\activate.bat
```

Pada Unix atau MacOS, operasikan

```
source tutorial-env/bin/activate
```

(Skrip ini ditulis untuk bash shell. Jika Anda menggunakan **csh** atau **fish** shells, ada pilihan skrip `activate.csh` dan `activate.fish` alternatif yang dapat Anda gunakan.)

Mengaktifkan lingkungan virtual akan mengubah prompt shell Anda untuk menunjukkan lingkungan virtual apa yang Anda gunakan, dan memodifikasi lingkungan sehingga menjalankan `python` akan membuat Anda mendapatkan versi dan instalasi Python tertentu. Sebagai contoh:

```
$ source ~/envs/tutorial-env/bin/activate
(tutorial-env) $ python
Python 3.5.1 (default, May 6 2016, 10:59:36)
...
>>> import sys
>>> sys.path
['', '/usr/local/lib/python35.zip', ...,
 '~/envs/tutorial-env/lib/python3.5/site-packages']
>>>
```

## 12.3 Mengelola Paket dengan pip

Anda dapat menginstal, mengupgrade, dan menghapus paket menggunakan program yang disebut **pip**. Secara bawaan `pip` akan menginstal paket dari Python Package Index, <https://pypi.org>. Anda dapat menelusuri Python Package Index dengan membukanya di peramban atau browser web Anda.

`pip` memiliki sejumlah sub-perintah: "install", "uninstall", "freeze", dls. (Konsultasikan ke panduan `installing-index` untuk dokumentasi lengkap dari `pip`.)

Anda dapat menginstal versi terbaru dari suatu paket dengan menyebutkan nama suatu paket:

```
(tutorial-env) $ pip install novas
Collecting novas
  Downloading novas-3.1.1.3.tar.gz (136kB)
Installing collected packages: novas
  Running setup.py install for novas
Successfully installed novas-3.1.1.3
```

Anda juga dapat menginstal versi spesifik suatu paket dengan memberikan nama paket diikuti dengan `==` dan nomor versi:

```
(tutorial-env) $ pip install requests==2.6.0
Collecting requests==2.6.0
  Using cached requests-2.6.0-py2.py3-none-any.whl
Installing collected packages: requests
Successfully installed requests-2.6.0
```

Jika Anda menjalankan kembali perintah ini, `pip` akan melihat bahwa versi yang diminta sudah diinstal dan tidak melakukan apa-apa. Anda dapat memberikan nomor versi yang berbeda untuk mendapatkan versi itu, atau Anda dapat menjalankan `pip install --upgrade` untuk meningkatkan paket ke versi terbaru:

```
(tutorial-env) $ pip install --upgrade requests
Collecting requests
Installing collected packages: requests
  Found existing installation: requests 2.6.0
    Uninstalling requests-2.6.0:
      Successfully uninstalled requests-2.6.0
Successfully installed requests-2.7.0
```

`pip uninstall` diikuti oleh satu atau beberapa nama paket akan menghapus paket-paket dari lingkungan virtual.

`pip show` akan menampilkan informasi tentang paket tertentu:

```
(tutorial-env) $ pip show requests
---
Metadata-Version: 2.0
Name: requests
Version: 2.7.0
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.com
License: Apache 2.0
Location: /Users/akuchling/envs/tutorial-env/lib/python3.4/site-packages
Requires:
```

`pip list` akan menampilkan semua paket yang diinstal di lingkungan virtual:

```
(tutorial-env) $ pip list
novas (3.1.1.3)
numpy (1.9.2)
pip (7.0.3)
requests (2.7.0)
setuptools (16.0)
```

`pip freeze` akan menghasilkan daftar yang sama dari paket yang diinstal, tetapi keluarannya menggunakan format yang diharapkan oleh `pip install`. Sebuah konvensi yang umum digunakan adalah meletakkan daftar ini dalam file `requirements.txt`:

```
(tutorial-env) $ pip freeze > requirements.txt
(tutorial-env) $ cat requirements.txt
novas==3.1.1.3
numpy==1.9.2
requests==2.7.0
```

`requirements.txt` kemudian dapat dikirimkan atau commit ke sistem kontrol versi dan dikirim sebagai bagian dari aplikasi. Pengguna kemudian dapat menginstal semua paket yang diperlukan dengan `install -r`:

```
(tutorial-env) $ pip install -r requirements.txt
Collecting novas==3.1.1.3 (from -r requirements.txt (line 1))
...
Collecting numpy==1.9.2 (from -r requirements.txt (line 2))
...
Collecting requests==2.7.0 (from -r requirements.txt (line 3))
...
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
Installing collected packages: novas, numpy, requests
  Running setup.py install for novas
Successfully installed novas-3.1.1.3 numpy-1.9.2 requests-2.7.0
```

`pip` memiliki banyak opsi lagi. Baca panduan `install-index` untuk dokumentasi lengkap `pip`. Ketika Anda telah menulis sebuah paket dan ingin membuatnya tersedia di Python Package Index, bacalah panduan `distributing-index`.

---

## 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` menjelaskan cara memasang modul tambahan yang ditulis oleh pengguna Python lainnya.
- `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>: *Python Package Index*, sebelumnya juga dijuluki Toko Keju<sup>1</sup>, adalah indeks modul Python yang dibuat pengguna yang tersedia untuk diunduh. Setelah Anda mulai menghasilkan kode, Anda dapat mendaftarkannya di sini sehingga orang lain dapat menemukannya.
- <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.)

---

<sup>1</sup> "Cheese Shop" atau Toko Keju adalah sketsa Monty Python: seorang pelanggan memasuki sebuah toko keju, tetapi keju apa pun yang dia minta, petugas toko mengatakan itu hilang.

- <http://www.pyvideo.org> mengumpulkan tautan ke video terkait Python dari konferensi dan pertemuan kelompok pengguna.
- <https://scipy.org>: Proyek Scientific Python mencakup modul untuk perhitungan dan manipulasi array cepat ditambah sejumlah paket untuk hal-hal seperti aljabar linier, transformasi Fourier, pemecah non-linear, distribusi angka acak, distribusi angka acak, analisis statistik, dan sejenisnya.

Untuk pertanyaan terkait Python dan laporan masalah, Anda dapat memposting ke newsgroup *comp.lang.python*, atau mengirimnya ke milis di [python-list@python.org](mailto:python-list@python.org). Newsgroup dan mailing list adalah gerbang, sehingga pesan yang diposting ke salah satu secara otomatis akan diteruskan ke yang lain. Ada ratusan posting sehari, menanyakan (dan menjawab) pertanyaan, menyarankan fitur baru, dan mengumumkan modul baru. Arsip milis tersedia di <https://mail.python.org/pipermail/>.

Sebelum memposting, pastikan untuk memeriksa daftar :ref:`Pertanyaan yang Sering Diajukan <faq-index>` (juga disebut FAQ). FAQ menjawab banyak pertanyaan yang muncul berulang-ulang, dan mungkin sudah mengandung solusi untuk masalah Anda.



---

## Pengeditan Input Interaktif dan Penggantian Riwayat

---

Beberapa versi interpreter Python mendukung pengeditan jalur input aktual dan penggantian riwayat, mirip dengan fasilitas yang ditemukan di shell Korn dan shell Bash GNU. Ini diimplementasikan menggunakan pustaka [GNU Readline](#), yang mendukung berbagai gaya pengeditan. Pustaka ini memiliki dokumentasi sendiri yang tidak akan kami duplikat di sini.

### 14.1 Pelengkapan Tab dan Pengeditan Riwayat

Penyelesaian nama variabel dan modul adalah diaktifkan secara otomatis pada startup interpreter sehingga tombol Tab memanggil fungsi pelengkapan; terlihat pada nama pernyataan Python, variabel lokal saat ini, dan nama modul yang tersedia. Untuk ekspresi putus-putus seperti `string.a`, itu akan mengevaluasi ekspresi hingga akhir `' '` dan kemudian menyarankan penyelesaian dari atribut dari objek yang dihasilkan. Perhatikan bahwa ini dapat mengeksekusi kode yang-ditentukan-aplikasi jika suatu objek dengan metode `__getattr__()` menjadi bagian dari ekspresi. Konfigurasi bawaan juga menyimpan riwayat Anda menjadi berkas bernama `.python_history` di direktori pengguna Anda. Riwayat akan tersedia lagi saat sesi interpreter interaktif berikutnya.

### 14.2 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 menampilkan 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

```
0.00011001100110011001100110011001100110011001100110011...
```

Berhenti pada jumlah bit yang terbatas, dan Anda mendapatkan perkiraan. Pada kebanyakan mesin saat ini, *float* diperkirakan menggunakan pecahan biner dengan pembilang menggunakan 53 bit pertama dimulai dengan bit paling signifikan dan dengan penyebut sebagai pangkat dua. Dalam kasus  $1/10$ , fraksi biner adalah  $3602879701896397 / 2^{55}$  yang dekat dengan tetapi tidak persis sama dengan nilai sebenarnya dari  $1/10$ .

Banyak pengguna tidak menyadari pendekatan tentang bagaimana cara nilai ditampilkan. Python hanya mencetak perkiraan desimal ke nilai desimal sebenarnya dari perkiraan biner yang disimpan oleh mesin. Pada kebanyakan mesin, jika Python mencetak nilai desimal sebenarnya dari perkiraan biner yang disimpan untuk 0.1, ia harus menampilkan

```
>>> 0.1
0.10000000000000000055511151231257827021181583404541015625
```

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

```
>>> 1 / 10
0.1
```

Hanya ingat, meskipun hasil cetakannya terlihat seperti nilai tepat  $1/10$ , nilai sebenarnya yang disimpan adalah pecahan biner terdekat yang dapat direpresentasikan.

Menariknya, ada banyak angka desimal berbeda yang memiliki pecahan biner perkiraan terdekat yang sama. Misalnya, angka 0.1 dan 0.100000000000000001 dan 0.10000000000000000055511151231257827021181583404541015625 semuanya didekati oleh  $3602879701896397 / 2^{55}$ . Karena semua nilai desimal ini memiliki perkiraan yang sama, salah satu dari nilai tersebut dapat ditampilkan sambil tetap mempertahankan invarian lainnya `eval(repr(x)) == x`.

Secara historis, Python prompt dan fungsi bawaan `repr()` akan memilih satu dengan 17 digit signifikan, 0.100000000000000001. Dimulai dengan Python 3.1, Python (pada kebanyakan sistem) sekarang dapat memilih yang paling pendek dan hanya menampilkan 0.1.

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).

Untuk hasil yang lebih menyenangkan, Anda mungkin ingin menggunakan pemformatan string untuk menghasilkan jumlah digit signifikan yang terbatas:

```
>>> format(math.pi, '.12g') # give 12 significant digits
'3.14159265359'

>>> format(math.pi, '.2f')  # give 2 digits after the point
'3.14'

>>> repr(math.pi)
'3.141592653589793'
```

Sangat penting untuk menyadari bahwa ini adalah, dalam arti sebenarnya, sebuah ilusi: Anda hanya membulatkan *display* dari nilai mesin yang sebenarnya.

Satu ilusi mungkin melahirkan yang lain. Misalnya, karena 0.1 tidak tepat  $1/10$ , menjumlahkan tiga nilai 0.1 mungkin tidak menghasilkan tepat 0.3, baik:

```
>>> .1 + .1 + .1 == .3
False
```

Juga, karena 0.1 tidak bisa mendekati nilai tepat 1/10 dan 0.3 tidak bisa mendekati nilai tepat 3/10, maka pra-pembulatan dengan fungsi `round()` tidak dapat membantu:

```
>>> round(.1, 1) + round(.1, 1) + round(.1, 1) == round(.3, 1)
False
```

Meskipun angka tidak dapat dibuat lebih dekat dengan nilai pastinya, fungsi `round()` dapat berguna untuk *post-rounding* sehingga hasil dengan nilai yang tidak tepat menjadi sebanding satu sama lain:

```
>>> round(.1 + .1 + .1, 10) == round(.3, 10)
True
```

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.

Seperti yang dikatakan menjelang akhir, "tidak ada jawaban yang mudah." Namun, jangan terlalu waspada terhadap pecahan *floating point*! Kesalahan dalam operasi float Python diwarisi dari pecahan *floating point* perangkat keras, dan pada kebanyakan mesin ada di urutan tidak lebih dari 1 bagian dalam  $2^{53}$  per operasi. Itu lebih dari cukup untuk sebagian besar tugas, tetapi Anda perlu ingat bahwa itu bukan aritmatika desimal dan bahwa setiap operasi *float* dapat mengalami kesalahan pembulatan baru.

Sementara kasus patologis memang ada, untuk sebagian besar penggunaan aritmatika *floating-point* yang santai Anda akan melihat hasil yang Anda harapkan pada akhirnya jika Anda hanya membulatkan tampilan hasil akhir Anda ke jumlah angka desimal yang Anda harapkan. `str()` biasanya mencukupi, dan untuk kontrol yang lebih baik lihat format `str.format()` penentu format di `formatstrings`.

Untuk kasus penggunaan yang memerlukan representasi desimal yang tepat, coba gunakan modul `decimal` yang mengimplementasikan aritmatika desimal yang cocok untuk aplikasi akuntansi dan aplikasi presisi tinggi.

Bentuk lain dari aritmatika yang tepat didukung oleh modul `fractions` yang mengimplementasikan aritmatika berdasarkan bilangan rasional (sehingga angka seperti 1/3 dapat direpresentasikan secara tepat).

Jika Anda adalah pengguna berat operasi *floating point*, Anda harus melihat pada paket *Numerical Python* dan banyak paket lainnya untuk operasi matematika dan statistik yang disediakan oleh proyek SciPy. Lihat <https://scipy.org>.

Python menyediakan alat yang dapat membantu pada saat-saat langka ketika Anda benar-benar *do* ingin tahu nilai pasti float. Metode `float.as_integer_ratio()` menyatakan nilai *float* sebagai pecahan:

```
>>> x = 3.14159
>>> x.as_integer_ratio()
(3537115888337719, 1125899906842624)
```

Karena rasio ini tepat, dapat digunakan untuk membuat ulang nilai asli tanpa berkurang *lossless*:

```
>>> x == 3537115888337719 / 1125899906842624
True
```

Metode `float.hex()` mengekspresikan float dalam heksadesimal (basis 16), sekali lagi memberikan nilai tepat yang disimpan oleh komputer Anda:

```
>>> x.hex()
'0x1.921f9f01b866ep+1'
```

Representasi heksadesimal yang tepat ini dapat digunakan untuk merekonstruksi nilai *float* dengan tepat:

```
>>> x == float.fromhex('0x1.921f9f01b866ep+1')
True
```

Karena representasinya tepat, maka berguna untuk porting nilai secara andal di berbagai versi Python (platform independensi) dan pertukaran data dengan bahasa lain yang mendukung format yang sama (seperti Java dan C99).

Alat lain yang bermanfaat adalah fungsi `math.fsum()` yang membantu mengurangi kehilangan presisi selama penjumlahan. Ini melacak "lost digits" karena nilai ditambahkan ke total yang sedang berlangsung. Itu dapat membuat perbedaan dalam akurasi keseluruhan sehingga kesalahan tidak terakumulasi ke titik di mana mereka mempengaruhi total akhir:

```
>>> sum([0.1] * 10) == 1.0
False
>>> math.fsum([0.1] * 10) == 1.0
True
```

## 15.1 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* mengacu pada fakta bahwa beberapa pecahan desimal (sebagian besar, sebenarnya) tidak dapat direpresentasikan persis sebagai pecahan biner (basis 2). Ini adalah alasan utama mengapa Python (atau Perl, C, C++, Java, Fortran, dan banyak lainnya) sering tidak akan menampilkan angka desimal tepat yang Anda harapkan.

Mengapa demikian?  $1/10$  tidak tepat direpresentasikan sebagai pecahan biner. Hampir semua mesin saat ini (November 2000) menggunakan aritmetika pecahan *floating point* IEEE-754, dan hampir semua platform memetakan *float* Python ke IEEE-754 "double precision". 754 *double* mengandung 53 bit presisi, sehingga pada input komputer berusaha untuk mengkonversi 0.1 ke fraksi terdekat dari bentuk  $J/2^{**N}$  di mana  $J$  adalah bilangan bulat yang mengandung persis 53 bit. Menulis ulang

```
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 <= 2**56 // 10 < 2**53
True
```

Artinya, 56 adalah satu-satunya nilai untuk  $N$  yang meninggalkan  $J$  dengan tepat 53 bit. Nilai terbaik untuk  $J$  adalah bahwa hasil bagi dibulatkan:

```
>>> 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
```

Oleh karena itu perkiraan terbaik untuk 1/10 dalam 754 presisi *double* adalah

```
7205759403792794 / 2 ** 56
```

Membagi pembilang dan penyebut dengan dua mengurangi pecahan menjadi:

```
3602879701896397 / 2 ** 55
```

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:

```
>>> 0.1 * 2 ** 55
3602879701896397.0
```

Jika kita mengalikan pecahan itu dengan 10\*\*55, kita bisa melihat nilainya menjadi 55 angka desimal:

```
>>> 3602879701896397 * 10 ** 55 // 2 ** 55
100000000000000000055511151231257827021181583404541015625
```

artinya angka persis yang disimpan di komputer sama dengan nilai desimal 0.10000000000000000055511151231257827021181583404541015625. Alih-alih menampilkan nilai desimal penuh, banyak bahasa (termasuk versi Python yang lebih lama), bulatkan hasilnya menjadi 17 digit signifikan

```
>>> format(0.1, '.17f')
'0.10000000000000001'
```

Modul `fractions` dan `decimal` membuat perhitungan ini mudah:

```
>>> from decimal import Decimal
>>> from fractions import Fraction

>>> Fraction.from_float(0.1)
Fraction(3602879701896397, 36028797018963968)

>>> (0.1).as_integer_ratio()
(3602879701896397, 36028797018963968)

>>> Decimal.from_float(0.1)
Decimal('0.10000000000000000055511151231257827021181583404541015625')

>>> format(Decimal.from_float(0.1), '.17f')
'0.10000000000000001'
```





## 16.1 Mode Interaktif

### 16.1.1 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.<sup>1</sup> Mengetik interupsi saat sebuah perintah dieksekusi memunculkan pengecualian `KeyboardInterrupt`, yang dapat ditangani oleh pernyataan :keyword:'try'.

### 16.1.2 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 python3.5
```

(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`.

---

<sup>1</sup> 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.

### 16.1.3 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)
```

### 16.1.4 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/python3.5/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*.

. . . Dapat mengacu ke:

- Prompt Python bawaan dari *shell* interaktif saat memasukkan kode untuk blok kode indentasi, ketika berada dalam sepasang pembatas kiri dan kanan yang cocok (tanda kurung, kurung kotak, kurung kurawal atau tanda kutip tiga), atau setelah menentukan *decorator*.
- Konstanta Ellipsis bawaan.

**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.abc` module), numbers (in the `numbers` module), streams (in the `io` module), import finders and loaders (in the `importlib.abc` module). You can create your own ABCs with the `abc` module.

**anotasi** A label associated with a variable, a class attribute or a function parameter or return value, used by convention as a *type hint*.

Annotations of local variables cannot be accessed at runtime, but annotations of global variables, class attributes, and functions are stored in the `__annotations__` special attribute of modules, classes, and functions, respectively.

See *variable annotation*, *function annotation*, **PEP 484** and **PEP 526**, which describe this functionality.

**argumen** A value passed to a *function* (or *method*) when calling the function. There are two kinds of argument:

- *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, the FAQ question on the difference between arguments and parameters, and [PEP 362](#).

**manajer konteks asinkron** An object which controls the environment seen in an `async with` statement by defining `__aenter__()` and `__aexit__()` methods. Introduced by [PEP 492](#).

**pembangkit asinkron** A function which returns an *asynchronous generator iterator*. It looks like a coroutine function defined with `async def` except that it contains `yield` expressions for producing a series of values usable in an `async for` loop.

Usually refers to an asynchronous generator function, but may refer to an *asynchronous generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.

An asynchronous generator function may contain `await` expressions as well as `async for`, and `async with` statements.

**iterator generator asinkron** Sebuah objek dibuat oleh fungsi *asynchronous generator*.

This is an *asynchronous iterator* which when called using the `__anext__()` method returns an awaitable object which will execute the body of the asynchronous generator function until the next `yield` expression.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *asynchronous generator iterator* effectively resumes with another awaitable returned by `__anext__()`, it picks up where it left off. See [PEP 492](#) and [PEP 525](#).

**asynchronous iterable** An object, that can be used in an `async for` statement. Must return an *asynchronous iterator* from its `__aiter__()` method. Introduced by [PEP 492](#).

**iterator asinkron** An object that implements the `__aiter__()` and `__anext__()` methods. `__anext__()` must return an *awaitable* object. `async for` resolves the awaitables returned by an asynchronous iterator's `__anext__()` method until it raises a `StopAsyncIteration` exception. Introduced by [PEP 492](#).

**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*.

**menunggu** An object that can be used in an `await` expression. Can be a *coroutine* or an object with an `__await__()` method. See also [PEP 492](#).

**BDFL** Benevolent Dictator For Life, a.k.a. [Guido van Rossum](#), Python's creator.

**berkas biner** A *file object* able to read and write *bytes-like objects*. Examples of binary files are files opened in binary mode ('rb', 'wb' or 'rb+'), `sys.stdin.buffer`, `sys.stdout.buffer`, and instances of `io.BytesIO` and `gzip.GzipFile`.

See also *text file* for a file object able to read and write `str` objects.

**bytes-like object** An object that supports the `bufferobjects` and can export a *C-contiguous* buffer. This includes all `bytes`, `bytearray`, and `array.array` objects, as well as many common `memoryview` objects. Bytes-like objects can be used for various operations that work with binary data; these include compression, saving to a binary file, and sending over a socket.

Some operations need the binary data to be mutable. The documentation often refers to these as "read-write bytes-like objects". Example mutable buffer objects include `bytearray` and a `memoryview` of a `bytearray`. Other operations require the binary data to be stored in immutable objects ("read-only bytes-like objects"); examples of these include `bytes` and a `memoryview` of a `bytes` object.

**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` 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.

**callback** A subroutine function which is passed as an argument to be executed at some point in the future.

**kelas** A template for creating user-defined objects. Class definitions normally contain method definitions which operate on instances of the class.

**class variable** A variable defined in a class and intended to be modified only at class level (i.e., not in an instance of the class).

**paksaan** 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`. 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](#).

**context variable** A variable which can have different values depending on its context. This is similar to Thread-Local Storage in which each execution thread may have a different value for a variable. However, with context variables, there may be several contexts in one execution thread and the main usage for context variables is to keep track of variables in concurrent asynchronous tasks. See `contextvars`.

**contiguous** A buffer is considered contiguous exactly if it is either *C-contiguous* or *Fortran contiguous*. Zero-dimensional buffers are C and Fortran contiguous. In one-dimensional arrays, the items must be laid out in memory next to each other, in order of increasing indexes starting from zero. In multidimensional C-contiguous arrays, the last index varies the fastest when visiting items in order of memory address. However, in Fortran contiguous arrays, the first index varies the fastest.

**coroutine** Coroutines are a more generalized form of subroutines. Subroutines are entered at one point and exited at another point. Coroutines can be entered, exited, and resumed at many different points. They can be implemented with the `async def` statement. See also [PEP 492](#).

**coroutine function** A function which returns a *coroutine* object. A coroutine function may be defined with the `async def` statement, and may contain `await`, `async for`, and `async with` keywords. These were introduced by [PEP 492](#).

**CPython** The canonical implementation of the Python programming language, as distributed on [python.org](https://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  
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 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 comprehension** A compact way to process all or part of the elements in an iterable and return a dictionary with the results. `results = {n: n ** 2 for n in range(10)}` generates a dictionary containing key `n` mapped to value `n ** 2`. See [comprehensions](#).

**dictionary view** The objects returned from `dict.keys()`, `dict.values()`, and `dict.items()` 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 `while`. 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.

**f-string** String literals prefixed with 'f' or 'F' are commonly called "f-strings" which is short for formatted string literals. See also [PEP 498](#).

**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 that is being imported.

Since Python 3.3, there are two types of finder: *meta path finders* for use with `sys.meta_path`, and *path entry finders* for use with `sys.path_hooks`.

See [PEP 302](#), [PEP 420](#) and [PEP 451](#) for much more detail.

**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.

**anotasi fungsi** An *annotation* of a function parameter or return value.

Function annotations are usually used for *type hints*: for example, this function is expected to take two `int` arguments and is also expected to have an `int` return value:

```
def sum_two_numbers(a: int, b: int) -> int:
    return a + b
```

Function annotation syntax is explained in section function.

See *variable annotation* and [PEP 484](#), which describe this functionality.

**\_\_future\_\_** A pseudo-module which programmers can use to enable new language features which are not compatible with the current interpreter.

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 becomes 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. The garbage collector can be controlled using the `gc` module.

**pembangkit** A function which returns a *generator iterator*. It looks like a normal function except that it contains `yield` expressions for producing a series of values usable in a for-loop or that can be retrieved one at a time with the `next()` function.

Usually refers to a generator function, but may refer to a *generator iterator* in some contexts. In cases where the intended meaning isn't clear, using the full terms avoids ambiguity.



**generator iterator** An object created by a *generator* function.

Each `yield` temporarily suspends processing, remembering the location execution state (including local variables and pending try-statements). When the *generator iterator* 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` clause defining a loop variable, range, and an optional `if` clause. 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
```

**fungsi generik** A function composed of multiple functions implementing the same operation for different types. Which implementation should be used during a call is determined by the dispatch algorithm.

See also the *single dispatch* glossary entry, the `functools.singledispatch()` decorator, and [PEP 443](#).

**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.

**hash-based pyc** A bytecode cache file that uses the hash rather than the last-modified time of the corresponding source file to determine its validity. See *pyc-invalidation*.

**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__()` 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.

Most of Python's immutable built-in objects are hashable; mutable containers (such as lists or dictionaries) are not; immutable containers (such as tuples and frozensets) are only hashable if their elements are hashable. 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.

**import path** A list of locations (or *path entries*) that are searched by the *path based finder* for modules to import. During import, this list of locations usually comes from `sys.path`, but for subpackages it may also come from the parent package's `__path__` attribute.

**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*.

**interpreter shutdown** When asked to shut down, the Python interpreter enters a special phase where it gradually releases all allocated resources, such as modules and various critical internal structures. It also makes several calls to the *garbage collector*. This can trigger the execution of code in user-defined destructors or weakref callbacks. Code executed during the shutdown phase can encounter various exceptions as the resources it relies on may not function anymore (common examples are library modules or the warnings machinery).

The main reason for interpreter shutdown is that the `__main__` module or the script being run has finished executing.

**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`, *file objects*, and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements *Sequence* semantics.

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 (or passing it to the built-in function `next()`) 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.merge()`, `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, a 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 = ['{:04x}'.format(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 and `importlib.abc.Loader` for an *abstract base class*.

**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`.

**meta path finder** A *finder* returned by a search of `sys.meta_path`. Meta path finders are related to, but different from *path entry finders*.

See `importlib.abc.MetaPathFinder` for the methods that meta path finders implement.

**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*.

**module spec** A namespace containing the import-related information used to load a module. An instance of `importlib.machinery.ModuleSpec`.

**MRO** Lihat *method resolution order*.

**mutable** Mutable objects can change their value but keep their `id()`. See also *immutable*.

**named tuple** The term “named tuple” applies to any type or class that inherits from `tuple` and whose indexable elements are also accessible using named attributes. The type or class may have other features as well.

Several built-in types are named tuples, including the values returned by `time.localtime()` and `os.stat()`. Another example is `sys.float_info`:

```

>>> sys.float_info[1]           # indexed access
1024
>>> sys.float_info.max_exp      # named field access
1024
>>> isinstance(sys.float_info, tuple) # kind of tuple
True

```

Some named tuples are built-in types (such as the above examples). Alternatively, a named tuple can be created from a regular class definition that inherits from `tuple` and that defines named fields. Such a class can be written by hand or it can be created with the factory function `collections.namedtuple()`. The latter technique also adds some extra methods that may not be found in hand-written or built-in named tuples.

**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 `builtins.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.islice()` makes it clear that those functions are implemented by the `random` and `itertools` modules, respectively.

**namespace package** A [PEP 420 package](#) which serves only as a container for subpackages. Namespace packages may have no physical representation, and specifically are not like a [regular package](#) because they have no `__init__.py` file.

Lihat juga [module](#).

**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 by default work only for reference and not for assignment. Local variables both read and write in the innermost scope. Likewise, global variables read and write to the global namespace. The `nonlocal` allows writing to outer scopes.

**new-style class** Old name for the flavor of classes now used for all class objects. In earlier Python versions, only new-style classes could use Python's newer, versatile features like `__slots__`, descriptors, properties, `__getattr__()`, class methods, and static methods.

**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.

See also [regular package](#) and [namespace package](#).

**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 five kinds of parameter:

- *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. Positional-only parameters can be defined by including a `/` character in the parameter list of the function definition after them, for example `posonly1` and `posonly2` in the following:

```
def func(posonly1, posonly2, /, positional_or_keyword): ...
```

- *keyword-only*: specifies an argument that can be supplied only by keyword. Keyword-only parameters can be defined by including a single var-positional parameter or bare `*` in the parameter list of the function definition before them, for example `kw_only1` and `kw_only2` in the following:

```
def func(arg, *, kw_only1, kw_only2): ...
```

- *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, the `inspect.Parameter` class, the function section, and [PEP 362](#).

**path entry** A single location on the *import path* which the *path based finder* consults to find modules for importing.

**path entry finder** A *finder* returned by a callable on `sys.path_hooks` (i.e. a *path entry hook*) which knows how to locate modules given a *path entry*.

See `importlib.abc.PathEntryFinder` for the methods that path entry finders implement.

**path entry hook** A callable on the `sys.path_hook` list which returns a *path entry finder* if it knows how to find modules on a specific *path entry*.

**path based finder** One of the default *meta path finders* which searches an *import path* for modules.

**path-like object** An object representing a file system path. A path-like object is either a `str` or `bytes` object representing a path, or an object implementing the `os.PathLike` protocol. An object that supports the `os.PathLike` protocol can be converted to a `str` or `bytes` file system path by calling the `os.fspath()` function; `os.fsdecode()` and `os.fsencode()` can be used to guarantee a `str` or `bytes` result instead, respectively. Introduced by [PEP 519](#).

**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](#).

**porsi** A set of files in a single directory (possibly stored in a zip file) that contribute to a namespace package, as defined in [PEP 420](#).

**positional argument** Lihat [argument](#).

**provisional API** A provisional API is one which has been deliberately excluded from the standard library's backwards compatibility guarantees. While major changes to such interfaces are not expected, as long as they are marked provisional, backwards incompatible changes (up to and including removal of the interface) may occur if deemed necessary by core developers. Such changes will not be made gratuitously -- they will occur only if serious fundamental flaws are uncovered that were missed prior to the inclusion of the API.

Even for provisional APIs, backwards incompatible changes are seen as a "solution of last resort" - every attempt will still be made to find a backwards compatible resolution to any identified problems.

This process allows the standard library to continue to evolve over time, without locking in problematic design errors for extended periods of time. See [PEP 411](#) for more details.

**provisional package** Lihat *provisional API*.

**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)
```

**nama yang memenuhi syarat** A dotted name showing the "path" from a module's global scope to a class, function or method defined in that module, as defined in **PEP 3155**. For top-level functions and classes, the qualified name is the same as the object's name:

```
>>> class C:
...     class D:
...         def meth(self):
...             pass
...
>>> C.__qualname__
'C'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__qualname__
'C.D.meth'
```

When used to refer to modules, the *fully qualified name* means the entire dotted path to the module, including any parent packages, e.g. `email.mime.text`:

```
>>> import email.mime.text
>>> email.mime.text.__name__
'email.mime.text'
```

**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.

**paket biasa** A traditional *package*, such as a directory containing an `__init__.py` file.

Lihat juga *namespace package*.

**\_\_slots\_\_** A declaration inside a 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 `bytes`. 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.

The `collections.abc.Sequence` abstract base class defines a much richer interface that goes beyond just `__getitem__()` and `__len__()`, adding `count()`, `index()`, `__contains__()`, and `__reversed__()`. Types that implement this expanded interface can be registered explicitly using `register()`.

**set comprehension** A compact way to process all or part of the elements in an iterable and return a set with the results. `results = {c for c in 'abracadabra' if c not in 'abc'}` generates the set of strings `{'r', 'd'}`. See [comprehensions](#).

**single dispatch** A form of [generic function](#) dispatch where the implementation is chosen based on the type of a single argument.

**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.

**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`.

**text encoding** A codec which encodes Unicode strings to bytes.

**berkas teks** A [file object](#) able to read and write `str` objects. Often, a text file actually accesses a byte-oriented datastream and handles the [text encoding](#) automatically. Examples of text files are files opened in text mode (`'r'` or `'w'`), `sys.stdin`, `sys.stdout`, and instances of `io.StringIO`.

See also [binary file](#) for a file object able to read and write [bytes-like objects](#).

**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)`.

**type alias** A synonym for a type, created by assigning the type to an identifier.

Type aliases are useful for simplifying [type hints](#). For example:

```
from typing import List, Tuple

def remove_gray_shades(
    colors: List[Tuple[int, int, int]]) -> List[Tuple[int, int, int]]:
    pass
```

could be made more readable like this:

```
from typing import List, Tuple

Color = Tuple[int, int, int]

def remove_gray_shades(colors: List[Color]) -> List[Color]:
    pass
```

See [typing](#) and [PEP 484](#), which describe this functionality.

**type hint** An [annotation](#) that specifies the expected type for a variable, a class attribute, or a function parameter or return value.

Type hints are optional and are not enforced by Python but they are useful to static type analysis tools, and aid IDEs with code completion and refactoring.

Type hints of global variables, class attributes, and functions, but not local variables, can be accessed using `typing.get_type_hints()`.

See `typing` and [PEP 484](#), which describe this functionality.

**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 `bytes.splitlines()` for an additional use.

**anotasi variabel** An *annotation* of a variable or a class attribute.

When annotating a variable or a class attribute, assignment is optional:

```
class C:
    field: 'annotation'
```

Variable annotations are usually used for *type hints*: for example this variable is expected to take `int` values:

```
count: int = 0
```

Variable annotation syntax is explained in section `annassign`.

See *function annotation*, [PEP 484](#) and [PEP 526](#), which describe this functionality.

**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.

Lihat juga `venv`.

**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

Python software and documentation are licensed under the *PSF License Agreement*.

Starting with Python 3.8.6, examples, recipes, and other code in the documentation are dual licensed under the PSF License Agreement and the *Zero-Clause BSD license*.

Some software incorporated into Python is under different licenses. The licenses are listed with code falling under that license. See *Lisensi dan Ucapan Terima Kasih untuk Perangkat Lunak yang Tergabung* for an incomplete list of these licenses.

### C.2.1 LISENSI PERJANJIAN PSF UNTUK PYTHON 3.8.20

1. This LICENSE AGREEMENT is between the Python Software Foundation ("PSF"),  
→and  
the Individual or Organization ("Licensee") accessing and otherwise using  
→Python  
3.8.20 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 3.8.20 alone or in any derivative  
version, provided, however, that PSF's License Agreement and PSF's notice  
→of  
copyright, i.e., "Copyright © 2001–2023 Python Software Foundation; All  
→Rights  
Reserved" are retained in Python 3.8.20 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 3.8.20 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  
3.8.20.
4. PSF is making Python 3.8.20 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 3.8.20 WILL NOT INFRINGE ANY THIRD PARTY RIGHTS.
5. PSF SHALL NOT BE LIABLE TO LICENSEE OR ANY OTHER USERS OF PYTHON 3.8.20  
 FOR ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT  
 ↳OF  
 MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 3.8.20, 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 3.8.20, 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,

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

MODIFYING OR DISTRIBUTING THE SOFTWARE, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

5. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
6. This License Agreement shall be governed by and interpreted in all respects by the law of the State of California, excluding conflict of law provisions. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between BeOpen and Licensee. This License Agreement does not grant permission to use BeOpen trademarks or trade names in a trademark sense to endorse or promote products or services of Licensee, or any third party. As an exception, the "BeOpen Python" logos available at <http://www.pythonlabs.com/logos.html> may be used according to the permissions granted on that web page.
7. By copying, installing or otherwise using the software, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## 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

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

ANY INCIDENTAL, SPECIAL, OR CONSEQUENTIAL DAMAGES OR LOSS AS A RESULT OF MODIFYING, DISTRIBUTING, OR OTHERWISE USING PYTHON 1.6.1, OR ANY DERIVATIVE THEREOF, EVEN IF ADVISED OF THE POSSIBILITY THEREOF.

6. This License Agreement will automatically terminate upon a material breach of its terms and conditions.
7. This License Agreement shall be governed by the federal intellectual property law of the United States, including without limitation the federal copyright law, and, to the extent such U.S. federal law does not apply, by the law of the Commonwealth of Virginia, excluding Virginia's conflict of law provisions. Notwithstanding the foregoing, with regard to derivative works based on Python 1.6.1 that incorporate non-separable material that was previously distributed under the GNU General Public License (GPL), the law of the Commonwealth of Virginia shall govern this License Agreement only as to issues arising under or with respect to Paragraphs 4, 5, and 7 of this License Agreement. Nothing in this License Agreement shall be deemed to create any relationship of agency, partnership, or joint venture between CNRI and Licensee. This License Agreement does not grant permission to use CNRI trademarks or trade name in a trademark sense to endorse or promote products or services of Licensee, or any third party.
8. By clicking on the "ACCEPT" button where indicated, or by copying, installing or otherwise using Python 1.6.1, Licensee agrees to be bound by the terms and conditions of this License Agreement.

## 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.2.5 ZERO-CLAUSE BSD LICENSE FOR CODE IN THE PYTHON 3.8.20 DOCUMENTATION

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, 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

(berlanjut ke halaman berikutnya)



(lanjutan dari halaman sebelumnya)

```
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 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 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.3.4 Pengelolaan *Cookie*

Modul `http.cookies` berisi pemberitahuan berikut:

```
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.3.5 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.3.6 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.3.7 XML Remote Procedure Calls

Modul `xmlrpc.client` berisi pemberitahuan berikut:

```
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.3.8 test\_epoll

Modul `test_epoll` berisi pemberitahuan berikut:

```
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.3.9 Pilih kqueue

Modul select berisi pemberitahuan berikut untuk antarmuka kqueue:

```
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.3.10 SipHash24

The file `Python/pyhash.c` contains Marek Majkowski's implementation of Dan Bernstein's SipHash24 algorithm. It contains the following note:

```
<MIT License>
Copyright (c) 2013 Marek Majkowski <marek@popcount.org>

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.
</MIT License>
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
Original location:
  https://github.com/majek/csiphash/

Solution inspired by code from:
  Samuel Neves (supercop/crypto_auth/siphhash24/little)
  djb (supercop/crypto_auth/siphhash24/little2)
  Jean-Philippe Aumasson (https://131002.net/siphhash/siphhash24.c)
```

### C.3.11 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
 * 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.3.12 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.
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

*
* 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:
*    "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.

```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```

*
* 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:
*    "This product includes cryptographic software written by
*    Eric Young (eay@cryptsoft.com)"
*    The word 'cryptographic' can be left out if the rouines 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.3.13 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.3.14 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.3.15 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) 1995-2011 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
```

### C.3.16 cfuhash

The implementation of the hash table used by the `tracemalloc` is based on the `cfuhash` project:

```
Copyright (c) 2005 Don Owens
All rights reserved.

This code is released under the BSD license:

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

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

* 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.

* Neither the name of the author 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 COPYRIGHT HOLDERS AND CONTRIBUTORS
"AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
```

(berlanjut ke halaman berikutnya)

(lanjutan dari halaman sebelumnya)

```
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.
```

### C.3.17 libmpdec

The `_decimal` module is built using an included copy of the libmpdec library unless the build is configured `--with-system-libmpdec`:

```
Copyright (c) 2008-2016 Stefan Krah. 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.3.18 Rangkaian pengujian W3C C14N

The C14N 2.0 test suite in the `test` package (`Lib/test/xmltestdata/c14n-20/`) was retrieved from the W3C website at <https://www.w3.org/TR/xml-c14n2-testcases/> and is distributed under the 3-clause BSD license:

Hak Cipta (c) 2013 W3C(R) (MIT, ERCIM, Keio, Beihang), Semua Hak Dilindungi Undang-Undang.

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

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

- Neither the name of the W3C nor the names of its contributors may be used to endorse or promote products derived from this work 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.

## LAMPIRAN D

---

### Hak Cipta

---

Python dan dokumentasi ini adalah:

Copyright © 2001-2023 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

..., [117](#)  
 # (*hash*)  
     comment, [9](#)  
 \* (*asterisk*)  
     in function calls, [30](#)  
 \*\*  
     in function calls, [30](#)  
 2ke3, [117](#)  
 : (*colon*)  
     function annotations, [32](#)  
 ->  
     function annotations, [32](#)  
 >>>, [117](#)  
 \_\_all\_\_, [54](#)  
 \_\_future\_\_, [121](#)  
 \_\_slots\_\_, [127](#)

## A

annotations  
     function, [32](#)  
 anotasi, [117](#)  
 anotasi fungsi, [121](#)  
 anotasi variabel, [129](#)  
 argumen, [117](#)  
 argumen kata kunci, [123](#)  
 asynchronous iterable, [118](#)  
 atribut, [118](#)

## B

BDFL, [118](#)  
 berkas biner, [118](#)  
 berkas teks, [128](#)  
 bilangan kompleks, [119](#)  
 builtins  
     modul, [52](#)  
 bytecode, [119](#)  
 bytes-like object, [119](#)

## C

callback, [119](#)  
 C-contiguous, [119](#)  
 class variable, [119](#)  
 coding  
     style, [32](#)  
 context variable, [119](#)  
 contiguous, [119](#)  
 coroutine, [119](#)  
 coroutine function, [119](#)  
 CPython, [120](#)

## D

daftar, [124](#)  
 descriptor, [120](#)  
 dictionary comprehension, [120](#)  
 dictionary view, [120](#)  
 diinterpretasi, [123](#)  
 docstring, [120](#)  
 docstrings, [23](#), [31](#)  
 documentation strings, [23](#), [31](#)  
 duck-typing, [120](#)

## E

EAFP, [120](#)  
 ekspresi, [120](#)

## F

f-string, [121](#)  
 file  
     object, [61](#)  
 file-like object, [121](#)  
 finder, [121](#)  
 floor division, [121](#)  
 for  
     statement, [20](#)  
 Fortran contiguous, [119](#)  
 function  
     annotations, [32](#)

fungsi, [121](#)  
fungsi built-in  
    help, [87](#)  
    open, [61](#)  
fungsi generik, [122](#)  
fungsi kunci, [123](#)

## G

generator, [121](#)  
generator expression, [122](#)  
generator iterator, [122](#)  
GIL, [122](#)

## H

hash-based pyc, [122](#)  
hashable, [122](#)  
help  
    fungsi built-in, [87](#)

## I

IDLE, [122](#)  
immutable, [122](#)  
import path, [122](#)  
importer, [122](#)  
importing, [122](#)  
interaktif, [123](#)  
interpreter shutdown, [123](#)  
iterable, [123](#)  
iterator, [123](#)  
iterator asinkron, [118](#)  
iterator generator asinkron, [118](#)

## J

json  
    modul, [64](#)  
jumlah referensi, [127](#)

## K

kamus, [120](#)  
kelas, [119](#)  
kelas basis abstrak, [117](#)  
kunci interpreter global, [122](#)

## L

lambda, [123](#)  
LBYL, [124](#)  
lingkungan virtual, [129](#)  
list comprehension, [124](#)  
loader, [124](#)

## M

magic  
    method, [124](#)

magic method, [124](#)  
manajer konteks, [119](#)  
manajer konteks asinkron, [118](#)  
mangling  
    name, [83](#)  
menunggu, [118](#)  
mesin virtual, [129](#)  
meta path finder, [124](#)  
metaclass, [124](#)  
method, [124](#)  
    magic, [124](#)  
    object, [78](#)  
    special, [128](#)  
method resolution order, [124](#)  
modul, [124](#)  
    builtins, [52](#)  
    json, [64](#)  
    sys, [51](#)  
modul tambahan, [121](#)  
module  
    search path, [50](#)  
module spec, [124](#)  
MRO, [124](#)  
mutable, [124](#)

## N

nama yang memenuhi syarat, [127](#)  
name  
    mangling, [83](#)  
named tuple, [124](#)  
namespace, [125](#)  
namespace package, [125](#)  
nested scope, [125](#)  
new-style class, [125](#)

## O

object  
    file, [61](#)  
    method, [78](#)  
objek, [125](#)  
objek berkas, [121](#)  
open  
    fungsi built-in, [61](#)

## P

paket, [125](#)  
paket biasa, [127](#)  
paksaan, [119](#)  
parameter, [125](#)  
PATH, [50](#), [115](#)  
path  
    module search, [50](#)  
path based finder, [126](#)  
path entry, [126](#)



path entry finder, [126](#)  
 path entry hook, [126](#)  
 path-like object, [126](#)  
 pembangkit, [121](#)  
 pembangkit asinkron, [118](#)  
 pemetaan, [124](#)  
 penghias, [120](#)  
 pengumpulan sampah, [121](#)  
 PEP, [126](#)  
 pernyataan, [128](#)  
 porsi, [126](#)  
 positional argument, [126](#)  
 provisional API, [126](#)  
 provisional package, [127](#)  
 Python 3000, [127](#)  
 Python Enhancement Proposals  
     PEP 1, [126](#)  
     PEP 8, [32](#)  
     PEP 238, [121](#)  
     PEP 278, [129](#)  
     PEP 302, [121](#), [124](#)  
     PEP 343, [119](#)  
     PEP 362, [118](#), [126](#)  
     PEP 411, [126](#)  
     PEP 420, [121](#), [125](#), [126](#)  
     PEP 443, [122](#)  
     PEP 451, [121](#)  
     PEP 484, [32](#), [117](#), [121](#), [128](#), [129](#)  
     PEP 492, [118](#), [119](#)  
     PEP 498, [121](#)  
     PEP 519, [126](#)  
     PEP 525, [118](#)  
     PEP 526, [117](#), [129](#)  
     PEP 3107, [32](#)  
     PEP 3116, [129](#)  
     PEP 3147, [50](#)  
     PEP 3155, [127](#)  
 Pythonic, [127](#)  
 PYTHONPATH, [50](#), [51](#)  
 PYTHONSTARTUP, [116](#)

## R

RFC  
     RFC 2822, [92](#)

## S

search  
     path, module, [50](#)  
 set comprehension, [128](#)  
 single dispatch, [128](#)  
 slice, [128](#)  
 special  
     method, [128](#)  
 special method, [128](#)

statement  
     for, [20](#)  
 strings, documentation, [23](#), [31](#)  
 style  
     coding, [32](#)  
 sys  
     modul, [51](#)

## T

teks tiga-kutip, [128](#)  
 text encoding, [128](#)  
 tipe, [128](#)  
 type alias, [128](#)  
 type hint, [128](#)

## U

universal newlines, [129](#)  
 urutan, [127](#)

## V

variabel environment  
     PATH, [50](#), [115](#)  
     PYTHONPATH, [50](#), [51](#)  
     PYTHONSTARTUP, [116](#)

## Z

Zen of Python, [129](#)