Serial Python Minimalis:
Tutorial #01-Python Dasar

Ahmad R. T. Nugraha dan Meng E. Ong

ver. 21 April 2026

Seri tutorial "Python Minimalis" ini ditujukan bagi para pelajar fisika secara khusus dan sains secara umum agar dapat "secukupnya" menguasai pemrograman dalam bahasa Python untuk keperluan komputasi numerik. Karena "minimalis", paradigma pemrograman yang ditekankan adalah pemrograman prosedural walaupun Python mendukung teknik lebih lanjut seperti pemrograman berorientasi objek (*object-oriented programming*/OOP).

Sebelum memulai pemrograman numerik, kita perlu menyiapkan kelengkapan "persenjataan" kita. Pemrograman Python untuk keperluan kita ini membutuhkan dua komponen utama: **interpreter** (mesin penerjemah kode) dan **integrated development environment** (IDE) atau tempat menulis kode. Kombinasi yang direkomendasikan adalah *interpreter* **Python** dari situs web resminya (atau dari repositori standar setiap distribusi Linux) dan IDE **Visual Studio Code (VS Code)**. Jika "malas" menginstal di komputer sendiri, kita masih bisa menggunakan lingkungan pemrograman Python berbasis *cloud* seperti `Google Colab`.

1 Instalasi Python dan Kelengkapannya

Untuk pemrograman Python yang profesional, kita tidak disarankan menginstal pustaka (*library*) secara global di sistem komputer karena dapat menyebabkan konflik versi. Praktik terbaik (*best practice*) adalah dengan menggunakan **virtual environment** (`venv`), yakni semacam "ruang isolasi" tempat kita bisa menginstal Python dan paket-paket tertentu (seperti NumPy atau Jupyter) khusus untuk satu proyek saja tanpa mengganggu sistem utama.

Langkah 1: Instalasi Python & VS Code

Pastikan dua komponen dasar berikut ini terinstal:

1. **Python** dari `python.org` untuk pengguna Windows.
   Penting: Saat instalasi, pastikan mencentang opsi **"Add Python to PATH"**. Sementara itu, bagi pengguna Linux, tergantung distribusi yang digunakan,

biasanya sudah ada instalasi Python dasar. Jika belum, misalnya untuk distribusi berbasis Debian/Ubuntu, kita bisa jalankan:

```
sudo apt install python3 python3-venv
```

2. **Visual Studio Code (VS Code).**
   Unduh (*download*) *installer* VS Code dari `code.visualstudio.com` dan lakukan proses instalasi standar sesuai sistem operasi yang digunakan.

Langkah 2: Membuat Folder Proyek & Virtual Environment

Kita akan membuat folder khusus untuk belajar Python dan membuat lingkungan virtual di dalamnya agar rapi.

1. Buat folder baru di komputer, misal bernama `PythonMinimalis`.
2. Buka folder tersebut menggunakan VS Code (*File* > *Open Folder*).
3. Buka Terminal di dalam VS Code dengan menekan `Ctrl + ‘` (tanda *backtick* di sebelah angka 1) atau menu *Terminal* > *New Terminal*.
4. Ketik perintah berikut di terminal untuk membuat `venv`:

   ```
   python -m venv .venv
   ```

   Jika sukses, folder baru bernama `.venv` akan muncul di panel kiri VS Code yang bertanda “Explorer”.

Langkah 3: Aktivasi Environment

Sebelum menginstal paket, kita harus "masuk" ke dalam lingkungan virtual tersebut. Untuk pengguna Linux/macOS, pada terminal bisa eksekusi:

```
source .venv/bin/activate
```

Sementara itu, untuk pengguna Windows, di terminal VS Code (biasanya PowerShell), ketik dan enter:

```
.venv\Scripts\Activate.ps1
```

Masalah Umum di Windows: UnauthorizedAccess

Jika kita mendapat pesan merah: *"...cannot be loaded because running scripts is disabled on this system"*, fitur keamanan Windows telah memblokir eksekusi skrip. Solusinya, jalankan perintah berikut di terminal untuk mengizinkan skrip lokal:

```
Set-ExecutionPolicy -ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Setelah itu, jalankan perintah aktivasi (`.venv\Scripts\Activate.ps1`) sekali lagi.

Jika berhasil, di sebelah kiri baris perintah akan muncul tanda kurung hijau (atau warna lain) dengan label semacam ini: `(.venv)`.

Langkah 4: Instalasi Paket Ilmiah

Setelah tanda `(.venv)` muncul (artinya kita sudah berada di dalam lingkungan virtual atau ruang isolasi), saatnya menginstal alat tempur kita, di antaranya:

- **numpy**: komputasi numerik array/matriks;
- **scipy**: paket komputasi tingkat lanjut yang siap pakai (integral, optimasi, dll.);
- **matplotlib**: untuk membuat grafik/plot; dan
- **jupyter**: untuk membuat *notebook* interaktif.

Jalankan perintah berikut di terminal:

```
pip install numpy scipy matplotlib jupyter
```

Tunggu hingga proses pengunduhan dan instalasi selesai.

Langkah 5: Menghubungkan VS Code dengan venv

Agar VS Code menggunakan Python yang ada di dalam `.venv` (bukan Python global), kita perlu melakukan konfigurasi yang tepat.

A. Format Jupyter Notebook (`*.ipynb`)

1. Buka atau buat file baru dengan akhiran `.ipynb` (misal: `catatan.ipynb`).
2. Klik tombol kernel di pojok kanan atas editor (biasanya tertulis *Select Kernel* atau *Python 3...*).
3. Pilih **Select Another Kernel...** $\rightarrow$ **Python Environments**.
4. Cari pilihan yang memiliki label `Star` ($\star$) atau yang mengarah ke folder `.venv` kita. *Contoh:* `Python 3.10.0 ('.venv': venv)`.

B. Format Skrip Python Biasa (`*.py`)

1. Buka atau buka file berakhiran `.py`.
2. Klik indikator versi Python di pojok kanan bawah jendela status bar VS Code.
3. Pilih *interpreter* yang mengarah ke `./.venv/Scripts/python.exe` (untuk Windows) atau `.venv` (untuk Linux/macOS).

Sekarang lingkungan kerja kita sudah ala profesional, terisolasi, dan siap untuk komputasi berat tanpa mengganggu sistem operasi utama.

2 Python sebagai Kalkulator

Di antara kegunaan paling sederhana pemrograman Python adalah sebagai kalkulator canggih. Kita dapat menjumlahkan, mengalikan, mengurangi angka atau bilangan, dan semacamnya. Ada tiga jenis bilangan utama dalam Python:

- **Integer** (bilangan bulat) diwakili oleh tipe `int`.
- **Bilangan riil** diwakili oleh tipe `float`. Nama `float` merujuk pada *floating-point numbers* (bilangan titik kambang), yaitu representasi perkiraan dari bilangan riil yang digunakan oleh Python (dan sebagian besar bahasa komputer modern lainnya).
- **Bilangan kompleks** (yang memiliki bagian riil dan imajiner) diwakili oleh tipe `complex`.

Dalam sesi Python interaktif seperti Jupyter Notebook, angka-angka dari tipe bilangan tersebut dapat digabungkan menggunakan operator dalam satu ekspresi yang dievaluasi dan hasilnya dikembalikan ke *prompt* (baris) keluaran. Contohnya:

```
1 + 2
```

Output:

```
3
```

```
10 / 4
```

Output:

```
2.5
```

```
2356 * 911
```

Output:

```
2146316
```

(**Catatan:** Pada mode interaktif, untuk dapat memunculkan "output" yang relevan, bisa dengan klik tombol semacam ▶Run yang ada pada editor/lingkungan pemrograman masing-masing atau dengan jalan pintas variasi tombol kibor tertentu, seperti `CTRL+Enter` atau `SHIFT+Enter`.)

Untuk membuat kode lebih mudah dipahami, ada baiknya kita rutin menambahkan komentar. Bentuk komentar singkat dapat berupa segala sesuatu pada satu baris setelah karakter # yang akan diabaikan oleh *interpreter* Python. Komentar ini berguna sebagai catatan yang dapat dibaca kita sendiri atau orang lain sehingga konteks kode akan lebih jelas. Contohnya:

```python
# Entalpi fusi molar es: konversi dari kJ.mol-1 ke J.mol-1.
6.01 * 1000
```

Output:

```
6010.0
```

```python
6.518 / 1013.25 * 760 # tekanan atmosfer di Mars, dalam Torr
```

Output:

```
4.888902047865778
```

Operator dan Urutan Operasi

Operator aljabar dasar tercantum dalam Tabel 1. Dalam penggunaannya, penting bagi kita untuk memperhatikan urutan prioritas operasi (Tabel 2), yakni urutan operator tersebut diterjemahkan dalam sebuah ekspresi. Contohnya:

```python
1 + 3 * 4
```

Output:

```
13
```

Di sini, $3 * 4$ dievaluasi terlebih dahulu karena operator perkalian, `*`, memiliki urutan prioritas yang lebih tinggi daripada `+` dan `*`. Hasilnya, 12, kemudian ditambahkan pada 1. Jika operator memiliki urutan prioritas yang sama, bagian-bagian dari ekspresi umumnya dievaluasi dari kiri ke kanan (kecuali pemangkatan atau eksponensial yang dievaluasi dari kanan ke kiri). Urutan prioritas ini dapat dianulir dengan menggunakan tanda kurung biasa:

```python
6 / 3 ** 2       # sama dengan 6 / 9
```

Output:

```
0.6666666666666666
```

```
(6 / 3) ** 2     # sama dengan 2 ** 2
```

Output:

```
4.0
```

Tabel 1: Operator aritmetika dasar Python.

Simbol	Operasi
`+`	Penjumlahan
`-`	Pengurangan
`*`	Perkalian
`/`	Pembagian riil (*float*)
`//`	Pembagian bulat khusus
`%`	Modulus (sisa bagi)
`**`	Pemangkatan

Tabel 2: Urutan prioritas operator aritmetika Python.

Operator	Tingkat Prioritas
`**`	(prioritas tertinggi)
`*`, `/`, `//`, `%`	(pertengahan & setara)
`+`, `-`	(prioritas terendah)

Pembagian

Perhatikan bahwa ekspresi di atas menghasilkan bilangan *floating point* meskipun kita mengoperasikan bilangan bulat. Kejadian tersebut dikarenakan operator pembagian, `/`, selalu mengembalikan `float`, bahkan ketika hasilnya adalah bilangan bulat.

Perlu diketahui bahwa ada operator pembagian khusus integer, `//`, yang mengembalikan hasil bagi bulat dari pembagian ("berapa kali angka kedua cukup dekat ke angka pertama"). Operator yang terkait, yakni `modulus`, `%`, memberikan sisa pembagiannya. Mari kita lihat contoh-contoh berikut ini.

```
7 / 3
```

Output:

```
2.3333333333333335
```

```
7 // 3
```

Output:

```
2
```

```
7 % 3
```

Output:

```
1
```

Sebagai catatan tambahan, amati hasil dari `7 / 3`. Nilai tepatnya, $2\frac{1}{3}$, tidak dapat direpresentasikan secara eksak ketika Python memformat bilangan riil (*floating-point numbers*) yang representasinya dalam komputer memiliki presisi terbatas (sekitar 1 dalam 10^{16}). Nilai `float` terdekat dengan jawaban yang dapat direpresentasikanlah yang dikembalikan.

Pustaka Matematika (`math` dan `numpy`)

Selain operator aljabar dasar, terdapat banyak fungsi matematika serta konstanta seperti π dan e yang disediakan oleh pustaka `math` dalam Python. Pustaka ini adalah modul bawaan yang disediakan di setiap instalasi Python (tidak perlu menginstal paket tambahan), tetapi harus diimpor dengan perintah: `import math`.

Fungsi-fungsi tersebut (sebagian tercantum dalam Tabel 3) kemudian dapat digunakan dengan menambahkan awalan `math.` Contohnya:

```
import math
math.sin(math.pi/4)
```

Output:

```
0.7071067811865475
```

Cara seperti di atas adalah contoh pemanggilan fungsi. Fungsi `math.sin` diberikan sebuah argumen (di sini, angka $\pi/4$) di dalam tanda kurung. Ekspresi tersebut kemudian mengembalikan hasil perhitungannya.

Paket **NumPy**, yang awalnya bukan bawaan Python (perlu diinstal secara terpisah), menyediakan semua fungsionalitas `math` dan berbagai tambahan fitur numerik. Oleh karena itu, kita akan lebih sering menggunakannya daripada `math`. Biasanya NumPy diimpor dengan alias `np`, seperti pada contoh berikut:

```
import numpy as np
1 / np.sqrt(2)
```

Output:

```
0.7071067811865475
```

Tabel 3: Beberapa fungsi dan konstanta yang disediakan oleh modul `math`. Sudut diasumsikan dalam radian.

Fungsi/Konstanta	**Deskripsi/Nilai Matematika**
`math.pi`	π
`math.e`	e
`math.sqrt(x)`	$\sqrt{x}$
`math.exp(x)`	e^x
`math.log(x)`	$\ln x$
`math.log10(x)`	$\log_{10} x$
`math.sin(x)`	$\sin(x)$
`math.cos(x)`	$\cos(x)$
`math.tan(x)`	$\tan(x)$
`math.asin(x)`	$\arcsin(x)$
`math.acos(x)`	$\arccos(x)$
`math.atan(x)`	$\arctan(x)$
`math.hypot(x, y)`	jarak Euklides (*Euclidean norm*), $\sqrt{x^2 + y^2}$
`math.comb(n, r)`	Koefisien binomial, $\binom{n}{r} \equiv {}^nC_r$
`math.degrees(x)`	Konversi x dari radian ke derajat
`math.radians(x)`	Konversi x dari derajat ke radian

Meskipun fungsi trigonometri dalam `math` dan NumPy menggunakan radian alih-alih derajat, ada beberapa metode praktis untuk saling konversi antara keduanya:

```
np.degrees(np.pi/2)
```

Output:

```
90.0
```

```
np.sin(np.radians(30))
```

Output:

```
0.49999999999999994
```

Perhatikan kembali presisi yang terbatas pada contoh tersebut. Kita tahu nilai eksak dari $\sin(30°)$ adalah 0.5, tetapi representasi oleh komputer dapat berupa angka 0.49999999999999994 seperti pada contoh.

Fungsi `math.log` dan `np.log` memberikan logaritma natural (basis e). Ada pula varian `math.log10` dan `np.log10` yang terpisah:

```
np.log(10)
```

Output:

```
2.302585092994046
```

```
1 / np.log10(np.e)
```

Output:

```
2.302585092994046
```

Fungsi Bawaan Lainnya

Ada beberapa fungsi bawaan yang berguna (yaitu, yang tidak memerlukan paket seperti `math` atau NumPy untuk diimpor):

- `abs` mengembalikan nilai absolut dari argumennya.
- `round` membulatkan angka ke presisi tertentu dalam digit desimal (atau ke bilangan bulat terdekat jika tidak ada presisi yang ditentukan).

3 Pendefinisian Bilangan

Berbeda dengan beberapa bahasa pemrograman lain, Python tidak mengharuskan pengguna/pemrogram untuk mendeklarasikan tipe bilangan sebelum digunakan. Angka-angka yang terlihat oleh *interpreter* sebagai bilangan bulat akan diperlakukan sebagai objek `int`, sementara yang tampak seperti bilangan riil akan menjadi objek `float`. Namun, tipe data ini adalah bilangan tanpa dimensi. Jika ada besaran fisis yang diwakili tipe-tipe data tersebut, pemrogram bertanggung jawab untuk menjaga sendiri satuannya.

Bilangan bulat dalam Python bisa sebesar apapun yang diizinkan memori komputer. Untuk mendefinisikan bilangan bulat yang sangat besar, akan sangat mudah memisahkan kelompok digit dengan karakter garis bawah, '_'.

```
# Konstanta Avogadro (mol-1): nilai eksak berdasarkan definisi.
602_214_076_000_000_000_000_000
```

Output:

```
602214076000000000000000
```

Bilangan titik kambang (*floating point*) dapat ditulis dengan titik desimal '.', dan boleh disertai pengelompokan digit opsional untuk memperjelas:

```
# Konstanta Gas (J.K-1.mol-1): nilai eksak berdasarkan definisi.
8.31_446_261_815_324
```

Output:

```
8.31446261815324
```

Selain itu, dalam notasi ilmiah, kita bisa menggunakan karakter `e` (atau `E`) yang memisahkan mantisa (digit signifikan) dan eksponen:

```
# Konstanta Boltzmann (J.K-1): nilai eksak berdasarkan definisi.
1.380649e-23
```

Output:

```
1.380649e-23
```

Bilangan kompleks dapat ditulis sebagai jumlah dari bagian riil dan imajiner. Bagian imajiner dalam Python ditandai dengan akhiran `j` (bukan `i`) mengikuti konvensi dalam *engineering*:

```
1 + 4j
```

Output:

```
(1+4j)
```

Cara lain, kita bisa secara eksplisit memberikan sepasang nilai pada fungsi `complex`:

```
complex(-2, 3)
```

Output:

```
(-2+3j)
```

Dalam bilangan kompleks, bagian riil dan imajiner direpresentasikan dalam bentuk titik kambang *floating point* dan dapat diperoleh secara terpisah menggunakan atribut `.real` dan `.imag`. Fungsi bawaan `abs` mengembalikan besaran (magnitudo) bilangan kompleks:

```
(3 + 4j).real
```

Output:

```
3.0
```

```
(3 + 4j).imag
```

Output:

```
4.0
```

```
abs(3 + 4j)
```

Output:

```
5.0
```

4 Variabel

Ketika memecahkan suatu masalah numerik tertentu, kita perlu menyimpan angka-angka dalam program sehingga dapat digunakan berulang kali dan dirujuk dengan nama yang mudah. Untuk mencapai tujuan tersebut, kita perlu mendefinisikan "variabel". Dalam Python, variabel dapat dianggap sebagai label yang disematkan pada objek (misalnya suatu bilangan `int` atau `float`). Ada beberapa aturan penting terkait penamaan variabel:

- Nama variabel boleh mengandung huruf, angka, dan karakter garis bawah (sering digunakan untuk mengindikasikan subskrip).
- Nama variabel tidak boleh diawali dengan angka.
- Variabel tidak boleh memiliki nama yang sama dengan salah satu dari kata-kata kunci tercadang (*reserved keywords*) yang dikhususkan dalam bahasa Python, seperti dapat dilihat pada Tabel 4.

Sebagian besar editor kode modern sudah memiliki fitur sorotan sintaksis yang akan memberikan peringatan saat ada kata-kata kunci khusus yang digunakan. Perhatikan perbedaan antara pemberian nilai atau penugasan (*assignment*) yang valid:

```
Konstanta Avogadro (mol-1): nilai eksak berdasarkan definisi.
N_A = 602_214_076_000_000_000_000_000
```

dan yang invalid:

```
import = 0
```

Output:

```
  File "/tmp/ipython-input-2269274265.py", line 1
    import = 0
           ^
SyntaxError: invalid syntax
```

Upaya pemberian nilai ke variabel bernama `import` gagal dilakukan (muncul `SyntaxError`) karena `import` adalah salah satu kata kunci tercadang. Kata kunci ini merupakan bagian dari sintaksis Python yang digunakan untuk mengimpor modul, seperti yang sudah pernah kita lakukan.

Pada praktiknya, kata kunci tercadang jarang menjadi nama variabel yang mungkin dipilih, dengan pengecualian `lambda` (belakangan akan kita lihat), yang bisa dipilih

untuk merepresentasikan panjang gelombang (*wavelength*). Namun, kita sarankan menggunakan nama lain seperti `lam` dalam kasus ini. Tabel 4 juga memuat tiga kata kunci khusus yang tidak dapat diubah: `True` dan `False`, yang mewakili konsep logika Boolean, serta `None`, yang digunakan untuk menyatakan nilai kosong atau ketiadaan.

Tabel 4: Kata kunci tercadang (*reserved keywords*) dalam Python 3.

`and`	`as`	`assert`	`async`	`await`
`break`	`class`	`continue`	`def`	`del`
`elif`	`else`	`except`	`finally`	`for`
`from`	`global`	`if`	`import`	`in`
`is`	`lambda`	`nonlocal`	`not`	`or`
`pass`	`raise`	`return`	`try`	`while`
`with`	`yield`	`False`	`True`	`None`

Nama variabel yang dipilih dengan baik dapat membuat kode Python sangat jelas dan ekspresif:

```
# Konstanta Boltzmann (J.K-1): nilai eksak berdasarkan definisi.
k_B = 1.380649e-23
R = N_A * k_B    # konstanta gas (J.K-1.mol-1)
```

Pada pernyataan terakhir, sisi kanan tanda `=`, yaitu ekspresi $N_A * k_B$, dievaluasi terlebih dahulu dan nama variabel `R` disematkan pada hasil perhitungan ini.

Kita bisa juga memodifikasi nilai yang terkait dengan nama variabel:

```
n = 1000
n = n + 1
```

Ekspresi $n = n + 1$ tidaklah merepresentasikan persamaan matematika analitik yang valid karena mustahil untuk diselesaikan. Namun, dalam "cara pikir" komputer, pernyataan ini merupakan instruksi untuk mengambil nilai n yang sebelumnya telah ditetapkan, kemudian tambahkan angka satu padanya, dan akhirnya ditetapkan ulang dengan memberi nama n kembali pada hasilnya. Nilai sebelumnya yang sebesar 1.000 langsung "dilupakan". Memori komputer yang digunakan untuk penyimpanannya dibebaskan dan dapat digunakan untuk keperluan lain. Ekspresi seperti pada contoh sangat umum dalam pemrograman apapun sehingga beberapa bahasa pemrograman, termasuk Python, menyediakan jalan pintas (disebut "penugasan yang diperluas" atau *augmented assignment*):

```
n = 1000
n += 1
```

Cara penulisan di atas akan memberikan hasil yang sama dengan sebelumnya. Ada sintaksis serupa untuk operator lain, seperti pengurangan (-=) dan perkalian (*=).

Jalan pintas lain yang berguna dan difasilitasi Python adalah penggunaan nilai yang dipisahkan koma untuk menetapkan beberapa variabel sekaligus, misalnya:

```
a, b, c = 42, -1, 0.5
```

Untuk selanjutnya, kita akan sebaik-baiknya memberikan nama variabel yang bermakna pada objek agar kode lebih bersifat "*self-documenting*" dan kita bisa meminimalkan penggunaan komentar penjelas.

5 Batasan dan Perangkap

Pembagian dengan nol selalu berbahaya dalam operasi matematika apapun. Peringatan *Exception* (kesalahan Python) dapat "dimunculkan" dalam kasus ini, dan eksekusi kode dihentikan. Bentuk pesan kesalahan dapat bervariasi, tergantung perhitungan mana yang memunculkan *Exception* tersebut. Sebagai contoh:

```
1/0
```

Output:

```
ZeroDivisionError         Traceback (most recent call last)
/tmp/ipython-input-2354412189.py in <cell line: 0>()
----> 1 1/0
ZeroDivisionError: division by zero
```

NumPy sedikit lebih toleran terhadap pembagian dengan nol, dan hanya mengeluarkan peringatan:

```
x = 0
y = np.sin(x) / x
```

Output:

```
/tmp/ipython-input-1162324160.py:4: RuntimeWarning: invalid value
↪  encountered in scalar divide
  y = np.sin(x) / x
```

Coba eksekusi lagi variabel y:

```
y
```

Output:

```
nan
```

Pada kasus ini, `y` telah ditetapkan ke nilai *floating-point* khusus berlabel `nan`, yang merupakan singkatan dari "*Not a Number*" (bukan angka). Label ini menandakan bahwa hasil perhitungan tidak terdefinisi. Apa yang selanjutnya bisa kita lakukan dengan informasi ini akan terserah kita. Namun, perlu diketahui bahwa tidak ada jalan kembali dari `NaN`. Operasi apa pun yang dilakukan pada nilai ini hanya akan mengembalikan `NaN` lain dan tidak ada yang sama dengan `NaN` (bahkan tidak sama dengan dirinya sendiri).

Beberapa perilaku, sebagian besar yang dihasilkan dari aritmetika *floating-point*, bisa mengejutkan bagi pengguna baru:

```
np.tan(np.pi/2)
```

Output:

```
1.633123935319537e+16
```

Nilai eksak secara matematis untuk $\tan(\pi/2)$ adalah $+\infty$, tetapi karena `np.pi` tidak merepresentasikan π secara eksak, nilai yang dikembalikan hanyalah angka yang sangat besar (orde `e+16` pada kasus ini), bukan tak terhingga.

Bilangan *floating-point*, karena disimpan dalam jumlah memori tetap (8 byte untuk *double-precision floating point*), memiliki magnitudo maksimum (dan minimum). Upaya evaluasi ekspresi yang menghasilkan angka terlalu besar untuk direpresentasikan (sekitar 1.8×10^{308}) akan memberikan "overflow":

```
np.exp(1000)
```

Output:

```
/tmp/ipython-input-2006519253.py:1: RuntimeWarning: overflow encountered in
↪ exp
  np.exp(1000)
np.float64(inf)
```

Pada contoh di atas, kondisi *overflow* telah mengembalikan nilai khusus `inf`, yang berarti tak terhingga (meskipun e^{1000} tidak tak terhingga, hanya sangat besar).

Hal yang sebalknya, yakni *underflow*, dapat terjadi ketika sebuah angka memiliki nilai absolut yang terlalu kecil untuk direpresentasikan dalam bentuk *double-precision floating point* (kurang dari sekitar 2.2×10^{-308}). Pada kasus ini, kita umumnya tidak akan diperingatkan:

```
np.exp(-1000)
```

Output:

```
0.0
```

Studi Kasus

Pada bagian ini, kita berikan beberapa contoh kasus dan solusinya untuk menunjukkan penggunaan Python sebagai kalkulator dari masalah tertentu.

Kasus 1 Stoikiometri Air

Berapa banyak molekul air yang ada dalam segelas air 250 mL? Gunakan rapat massa, $\rho(\mathrm{H_2O}(l)) = 1\ \mathrm{g\ cm^{-3}}$ dan massa molar, $M(\mathrm{H_2O}) = 18\ \mathrm{g\ mol^{-1}}$.

Solusi

1 mL adalah volume yang sama dengan $1\ \mathrm{cm^3}$ sehingga kita bisa tetap menggunakan satuan tersebut. Kita definisikan beberapa variabel terlebih dahulu:

```
# Konstanta Avogadro, dalam mol-1 dinyatakan hingga 4 angka penting.
N_A = 6.022e23
# Volume air yang sedang ditinjau, dalam cm3.
V = 250
# Rapat massa air, dalam g.cm-3.
rho = 1
# Massa molar H2O, dalam g.mol-1.
M_H2O = 18
```

Air di gelas kita bermassa $m = \rho V$ dan mengandung $n = m/M(\mathrm{H_2O})$ mol air.

```
# Massa air, dalam g.
m = rho * V
# Jumlah zat air, dalam mol.
n = m / M_H2O
n
```

Output:

```
13.88888888888889
```

Jumlah molekul air kemudian adalah $N = nN_A$.

```
n * N_A
```

Output:

```
8.363888888888889e+24
```

Hasil terakhir ini memberikan jawaban $N = 8.364 \times 10^{24}$ molekul.

Kasus 2 Laju Suara dalam Medium

Laju suara c dalam gas dengan massa molar M pada suhu T adalah

$$c = \sqrt{\frac{\gamma RT}{M}},$$

dengan R adalah konstanta gas. Untuk udara, diketahui indeks adiabatik $\gamma = 7/5$. Perkirakan laju suara di udara pada (a) 25°C dan (b) −20°C. Gunakan $M = 29\ \mathrm{g\ mol^{-1}}$.

Solusi

Kita memiliki dua suhu untuk dihitung. Untuk menghindari pengulangan, definisikan faktor f terlebih dahulu:

$$c = f\sqrt{T} \quad \text{dengan} \quad f = \sqrt{\frac{\gamma R}{M}}.$$

Jika kita tetap menggunakan satuan SI (perhatikan bahwa ini berarti menyatakan M dalam $\mathrm{kg\ mol^{-1}}$), kita harus membuat c keluar dalam satuan $\mathrm{m\ s^{-1}}$. Secara eksplisit:

$$[c] = \sqrt{\frac{[\cdot][\mathrm{J\ K^{-1}mol^{-1}}][\mathrm{K}]}{[\mathrm{kg\ mol^{-1}}]}} = \sqrt{\frac{[\mathrm{J}]}{[\mathrm{kg}]}} = \sqrt{\frac{[\mathrm{kg\ m^2 s^{-2}}]}{[\mathrm{kg}]}} = \mathrm{m\ s^{-1}}.$$

```
import numpy as np
# Konstanta gas dalam J.K-1.mol-1 (4 a.p.).
R = 8.314
# Massa molar rata-rata udara, dalam kg.mol-1.
M = 29 / 1000
# Rasio kapasitas panas C_p / C_V (indeks adiabatik) gas diatomik.
gamma = 7 / 5

# Faktor f dalam m.s-1.K-1/2.
f = np.sqrt(gamma * R / M)

# Konversi suhu pertama dari degC ke K.
T = 25 + 273
f * np.sqrt(T)
```

Output:

```
345.84234000181505
```

```
# Konversi suhu kedua dari degC ke K.
T = -20 + 273
f * np.sqrt(-20 + 273)
```

Output:

```
318.66200881509076
```

Jadi, pada 25°C laju suara adalah 346 $\mathrm{m\,s^{-1}}$, sedangkan pada -20°C, lajunya 319 $\mathrm{m\,s^{-1}}$.

Kasus 3 Gerak Jatuh dengan Kecepatan Awal

Sebuah batu dilemparkan vertikal ke bawah dari puncak tebing setinggi $h = 200$ m dengan besar kecepatan awal $v_0 = 5$ m/s. Berapa lama waktu (t) yang dibutuhkan batu tersebut untuk mencapai tanah? Gunakan percepatan gravitasi $g = 9.81$ m/s^2.

Solusi

Posisi batu y pada waktu t dapat ditelusuri dengan persamaan kinematika:

$$y(t) = h - v_0 t - \frac{1}{2} g t^2$$

Batu mencapai tanah ketika $y(t) = 0$, sehingga kita mendapatkan persamaan:

$$h = v_0 t + \frac{1}{2} g t^2 \quad \Rightarrow \quad \frac{1}{2} g t^2 + v_0 t - h = 0,$$

yang merupakan persamaan kuadrat dalam bentuk $at^2 + bt + c = 0$, dengan $a = \frac{1}{2} g$, $b = v_0$, dan $c = -h$.
Kita selesaikan t dengan rumus abc (rumus kuadratik):

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Karena waktu t harus positif ($t > 0$), kita ambil akar positifnya:

$$t = \frac{-v_0 + \sqrt{v_0^2 - 4(\frac{1}{2} g)(-h)}}{2(\frac{1}{2} g)} = \frac{-v_0 + \sqrt{v_0^2 + 2gh}}{g}$$

```python
# Koefisien persamaan kuadrat: (1/2*g)t^2 + (v0)t - h = 0
a = 0.5 * g
b = v0
c = -h

# Menghitung t menggunakan rumus abc (akar positif)
t_exact = (-b + np.sqrt(b**2 - 4 * a * c)) / (2 * a)
t_exact
```

Output:

```
5.895662701892837
```

Jadi, waktu yang dibutuhkan batu untuk mencapai tanah adalah sekitar 5.90 detik.

Kasus 4 Statistik Maxwell-Boltzmann

Dalam atmosfer matahari ($T \approx 5800$ K), atom hidrogen dapat berada pada keadaan dasar ($n = 1$) maupun keadaan tereksitasi. Hitunglah rasio populasi atom hidrogen pada tingkat energi eksitasi pertama ($n = 2$) dibandingkan dengan keadaan dasar ($n = 1$). Diketahui:

- Perbedaan energi antara $n = 1$ dan $n = 2$ adalah $\Delta E = 10.2$ eV.
- Konstanta Boltzmann $k_B = 8.617 \times 10^{-5}$ eV/K.
- Faktor degenerasi untuk tingkat n adalah $g_n = 2n^2$.

Rumus distribusi Boltzmann:

$$\frac{N_2}{N_1} = \frac{g_2}{g_1} e^{-\Delta E/(k_B T)}$$

Solusi

Pertama, kita tetapkan variabel untuk konstanta fisika dan parameter yang diketahui. Perhatikan bahwa energi diberikan dalam satuan elektron-volt (eV) sehingga sebaiknya kita menggunakan konstanta Boltzmann dalam satuan yang sesuai (eV/K) agar satuan saling menghilangkan di eksponen.

```python
import numpy as np
# Konstanta Boltzmann dalam eV/K
kB = 8.617e-5
# Parameter Fisika
T = 5800            # Suhu permukaan matahari (K)
dE = 10.2           # Perbedaan energi (eV)
# Faktor degenerasi g_n = 2n^2
g1 = 2 * (1**2)     # g1 = 2
g2 = 2 * (2**2)     # g2 = 8
```

Rasio populasi ditentukan oleh faktor statistik (rasio degenerasi) dikalikan dengan faktor Boltzmann (eksponensial).

$$\text{Rasio} = \frac{g_2}{g_1} \times \exp\left(-\frac{\Delta E}{k_B T}\right)$$

```python
# Menghitung rasio N2/N1
# Perhatikan tanda kurung pada penyebut (kB * T)
rasio = (g2 / g1) * np.exp(-dE / (kB * T))
rasio
```

Output:

```
5.50763970425776e-09
```

Hasilnya sekitar 5.5×10^{-9}, yang berarti hanya satu dari setiap ∼200 juta atom hidrogen yang berada dalam keadaan tereksitasi $n = 2$ di permukaan matahari.

Hati-hati perangkap matematis! Ada kesalahan umum yang sering terjadi saat menerjemahkan rumus fisika seperti $e^{-E/kT}$ ke dalam kode Python satu baris, yaitu masalah urutan prioritas operator.

```python
# SALAH!
# Python akan mengevaluasi (-dE / kB) terlebih dahulu, lalu hasilnya
↪  DIKALIKAN dengan T
rasio_salah = (g2 / g1) * np.exp(-dE / kB * T)
rasio_salah
```

Output:

```
0.0
```

Kode di atas salah karena secara efektif menghitung $e^{(-\Delta E/k_B) \times T}$, menghasilkan pangkat negatif yang sangat besar (sehingga hasilnya *underflow* menjadi 0). Pastikan kita selalu menggunakan tanda kurung untuk penyebut yang terdiri dari perkalian: `np.exp(-dE / (kB * T))`.

Serial Python Minimalis:
Tutorial #02-String

Ahmad R. T. Nugraha

ver. 21 April 2026

1 Definisi String

"String" (dengan tipe `str` dalam Python) adalah urutan karakter, biasanya mewakili data teks. Contoh string adalah pesan yang dapat dicetak di layar. String didefinisikan dengan menggunakan tanda kutip tunggal maupun ganda, dan dapat dialokasikan ke nama variabel seperti halnya angka. Jika kita tuliskan format string dengan tanda kutip begitu dan langsung dieksekusi, bagian output pada mode interaktif Python akan langsung memberikan hasil yang sama dengan yang ditulis pada input.

```
'sodium'
```

Output:

```
'sodium'
```

```
gas = "Carbon dioxide"
print(gas)
```

Output:

```
Carbon dioxide
```

String dapat digabungkan (*concatenated*) dengan operator + dan diulang dengan operator *:

```
'CH3' + 'CH2'*3 + 'CH3'
```

Output:

```
'CH3CH2CH2CH2CH3'
```

```
prefix = 'trans-'
prefix + 'but-2-ene'
```

Output:

```
'trans-but-2-ene'
```

String yang berada di antara tanda kutip disebut *string literal*. String semacam ini yang didefinisikan bersebelahan satu sama lain akan digabungkan secara otomatis (tanpa memerlukan +).

```
'Na' 'Cl'
```

Output:

```
'NaCl'
```

Perhatikan bahwa tidak ada spasi yang ditambahkan ke dalam string gabungan tersebut dan biasanya akan berguna ketika membuat string yang panjang, yang dapat dipecah menjadi beberapa bagian pada baris terpisah asalkan keseluruhan ekspresi tersebut diapit dalam tanda kurung.

```
quote = ('For me physics represented an indefinite cloud of future '
         'potentialities which enveloped my life to come in silent '
         'symmetries and vast cosmic laws')
```

Karakter `escape`

Pemisah baris (*line breaks*) tidak disertakan dalam string. Untuk merepresentasikan baris baru dan karakter khusus lainnya seperti tab, kita perlu menggunakan karakter "penyelamatan" (*escape*), yang diawali dengan garis miring terbalik (*backslash*, '\'). Tabel 1 menunjukkan urutan *escape* (*escape sequences*) yang umum digunakan.

Tabel 1: Urutan *escape* Python yang umum digunakan.

Urutan *escape*	Makna
`\'`	Tanda kutip tunggal (')
`\"`	Tanda kutip ganda (")
`\n`	*Linefeed* (LF) / Baris baru
`\r`	*Carriage return* (CR)
`\t`	Tab horizontal
`\\`	Karakter *backslash* itu sendiri
`\u`, `\U`, `\N{}`	Karakter Unicode berdasarkan *code point*-nya

Karakter-karakter `escape` ini tidak akan diproses dalam string literal yang ditampilkan oleh *shell* interaktif Python atau sel keluaran Jupyter. Agar dapat diproses, kita perlu memberikan string tersebut ke fungsi `print`. Contoh berikut ini menunjukkan input-output dari karakter `escape` yang tidak diproses menjadi makna yang diharapkan:

```
'Li\tNa\t\tRb\n3\t11\t19\t37'
```

Output:

```
'Li\tNa\t\tRb\n3\t11\t19\t37'
```

Sementara itu, jika kita menggunakan fungsi `print`, kita akan memperoleh apa yang kita kehendaki dari karakter `escape`:

```
print('Li\tNa\t\tRb\n3\t11\t19\t37')
```

Output:

```
Li      Na      Rb
3       11      19      37
```

Pada contoh di atas, karakter *escape* `\t` telah dicetak sebagai tab dan `\n` mengakhiri baris keluaran dan memulai baris baru.

Ada kalanya (misalnya, saat mendefinisikan string kode sumber LaTeX), kita tidak ingin *backslash* dianggap sebagai karakter *escape*. Untuk kasus ini, kita perlu melakukan *escape* pada *backslash* itu sendiri (`'\\'`) atau definisikan *raw string* (`r'...'`) sebagai berikut:

```
print('\tan(\pi)')
# Ups: kita tidak ingin \t diubah menjadi tab
```

Output:

```
        an(\pi)
```

```
# Lakukan escape pada backslash
print('\\tan(\pi)')
```

Output:

```
\tan(\pi)
```

```
# Atau gunakan raw string
print(r'\tan(\pi)')
```

Output:

```
\tan(\pi)
```

Fungsi `print` sangatlah berguna dan dapat menerima urutan objek yang dipisahkan koma (termasuk angka, yang akan dikonversi menjadi string) lalu mencetaknya. Secara *default*, fungsi ini memisahkan objek-objek tersebut dengan satu spasi, dan memberikan satu baris baru (*newline*) di akhir keluarannya.

```
print('Standard atomic weight of Na:', 22.99)
```

Output:

```
Standard atomic weight of Na: 22.99
```

```
print('2 + 3 =', 2 + 3)
```

Output:

```
2 + 3 = 5
```

Perbedaan antara `\n` dan `\r` adalah bahwa `\r` mengembalikan "kursor" cetak ke awal baris tetapi tidak membuat baris baru, sehingga keluaran berikutnya menimpa apa pun yang ada di sana sebelumnya:

```
print("Fluorine\rChl")
print("Fluorine\nChl")
```

Output:

```
Chlorine
Fluorine
Chl
```

Karakter Unicode dapat diidentifikasi berdasarkan *code point*-nya: setiap karakter di hampir semua bahasa tulis utama di dunia diberikan bilangan bulat dalam standar Unicode, dan kode bilangan bulat ini dapat digunakan dalam string Python menggunakan kode *escape* `\u`:

```
print("\u212b") # Angstrom Swedia
```

Output:

```
Å
```

Pengindeksan dan Pemotongan String (*Slicing*)

Setiap karakter individual dalam suatu string dapat diakses menggunakan notasi pengindeksan `s[i]`, dengan indeks `i` adalah posisi karakter dalam string dimulai dari nol. Misalnya:

```
s = 'Plutonium'
s[0]
```

Output:

```
'P'
```

```
s[4]
```

Output:

```
'o'
```

Jika indeks `i` bernilai negatif, pengindeksan dihitung mundur dari akhir string (karakter terakhir berada di "posisi" -1):

```
s[-1]
```

Output:

```
'm'
```

```
s[-3]
```

Output:

```
'i'
```

Berikut adalah ilustrasi indeks pada string "Plutonium":

Karakter	P	l	u	t	o	n	i	u	m
Indeks	0	1	2	3	4	5	6	7	8
Indeks (-)	-9	-8	-7	-6	-5	-4	-3	-2	-1

Substring didapatkan dengan memotong (*slicing*) string menggunakan notasi `s[i:j]`, dengan `i` adalah indeks karakter pertama yang digunakan, dan `j` adalah satu posisi setelah karakter terakhir yang diambil. Artinya, substring mencakup karakter awal tapi mengecualikan karakter akhir:

```
s[2:4]
# Substring adalah s[2] + s[3], karakter ketiga dan keempat
```

Output:

```
'ut'
```

Untuk sepenuhnya memahami substring, kita perlu pembiasaan, tetapi sebenarnya notasinya cukup berguna. String yang dikembalikan memiliki $j - i$ karakter di dalamnya, dan potongan berurutan menggunakan indeks yang dimulai tepat di tempat potongan sebelumnya berakhir:

```
s[0:4] + s[4:7] + s[7:9]
```

Output:

```
'Plutonium'
```

String dapat dipotong bahkan dihitung mundur atau dicacah dengan langkah (*stride*) yang berbeda menggunakan nilai ketiga dalam format `s[i:j:stride]`. Jika salah satu dari `i`, `j`, atau `stride` dihilangkan (dengan tetap menjaga format tanda titik dua), interpreter Python akan mengasumsikan nilai berturut-turut adalah awal string ($i = 0$), akhir string, dan stride $= 1$ (karakter berurutan). Perhatikan beberapa contoh berikut.

```
s[:5]
# Lima karakter pertama
```

Output:

```
'Pluto'
```

```
s[1:6:2]
# s[1], s[3], s[5]
```

Output:

```
'ltn'
```

```
s[::-1]
# Seluruh string, dibalik
```

Output:

```
'muinotulp'
```

String dalam Python bersifat *immutable* (tidak dapat diubah). Kita tidak dapat mengubahnya seketika dalam proses interpretasi sehingga upaya perubahan akan memberikan pesan *error*.

```
s = 'alkane'
s[3] = 'e'
```

Output:

```
TypeError                                 Traceback (most recent call last)
Cell In[4], line 2
      1 s = 'alkane'
----> 2 s[3] = 'e'

TypeError: 'str' object does not support item assignment
```

Sebagai gantinya, string baru dapat dibuat dari potongan string yang lama:

```
# 'alk' + 'e' + 'ne'
s[:3] + 'e' + s[4:]
```

Output:

```
'alkene'
```

String baru tersebut tentu saja dapat ditetapkan ke nama variabel, termasuk variabel yang sama, jika kita kehendaki.

```
# 'alk' + 'y' + 'ne'
s = s[:3] + 'y' + s[4:]
s
```

Output:

```
'alkyne'
```

2 Metode String

Python dilengkapi dengan sejumlah besar metode untuk memanipulasi string. Daftar lengkapnya tersedia dalam dokumentasi resmi Python. Di sini kita akan bahas beberapa metode yang paling berguna.

```
# definisikan string yang akan dijadikan contoh:
molekul = 'CH3CH2CH2CH3'

# panjang urutan, fungsi bawaan (built-in) Python:
len(molekul)
```

Output:

```
12
```

```
# jumlah substring
molekul.count('CH2')
```

Output:

```
2
```

```
# mengubah semua karakter menjadi huruf kecil
molekul.lower() # selain ini, tersedia juga molekul.upper()
```

Output:

```
'ch3ch2ch2ch3'
```

```
# menentukan indeks pertama dari substring
molekul.index('CH2')
```

Output:

```
3
```

```
# hapus karakter 'C', 'H', '3' mana pun dari kedua ujung f
molekul.strip('CH3')
```

Output:

```
'2CH2'
```

```
molekul.removeprefix('CH3') # selain itu, ada juga molekul.removesuffix()
↪   sejak Python 3.9
```

Output:

```
'CH2CH2CH3'
```

```
molekul.replace('CH3', 'NH2')
```

Output:

```
'NH2CH2CH2NH2'
```

Sebagai penegasan, string `f` bersifat *immutable* dan tidak berubah oleh operasi-operasi ini. Apa yang dilakukan operasi-operasi tersebut adalah mengembalikan objek Python baru. Dengan kata lain, metode-metode tersebut dapat dirangkai (*chained*) bersama, misalnya:

```
molekul.removeprefix('CH3').removesuffix('CH3').lower()
```

Output:

```
'ch2ch2'
```

3 Pemformatan String

Ketika sebuah angka dicetak ke layar, Python mengubahnya menjadi string dan berusaha sebaik mungkin untuk memilih representasi yang sesuai. Bilangan bulat selalu ditampilkan dengan semua digitnya, sementara untuk bilangan *floating point* yang sangat besar atau sangat kecil, Python mengadopsi notasi ilmiah.

```
N_A = 602_214_076_000_000_000_000_000
print(N_A)
```

Output:

```
602214076000000000000000
```

```
print(float(N_A))
```

Output:

```
6.02214076e+23
```

Cara angka diformat dapat dikontrol menggunakan metode string `format`. Dalam penggunaan paling sederhana, nilai-nilai disisipkan begitu saja ke dalam *template* string pada lokasi yang ditunjukkan oleh kurung kurawal (`'{}'`):

```
c = 299792458
units = 'm.s-1'
'The speed of light is c={} {}'.format(c, units)
```

Output:

```
'The speed of light is c=299792458 m.s-1'
```

Argumen yang diteruskan ke metode `format` juga dapat dirujuk berdasarkan nama atau indeks (berbasis nol) di dalam kurung kurawal:

```
'{0}={1} {cgs_units}'.format('c', c*100, cgs_units='cm.s-1')
```

Output:

```
'c=29979245800 cm.s-1'
```

Ada sintaksis khusus untuk lebih memperindah pemformatan string hasil interpolasi ini, di antaranya adalah lebar bidang (*field width*), *padding*, jumlah tempat desimal, dan sebagainya. Sebagai contoh, untuk bilangan bulat, lebar spasi yang dialokasikan untuk angka (yakni `w`) ditentukan dengan `:wd`.

```
'{:6d}'.format(42)
```

Output:

```
'    42'
```

Angka 42 dikeluarkan sebagai string sejumlah 6 karakter, diawali dari sebelah kiri dengan beberapa spasi hingga akhirnya angka 2 merupakan karakter ke-6 (indeks 5) dalam string tersebut.

Untuk bilangan riil *floating-point*, kita dapat menentukan presisi, `p`, serta lebar, `w`, yakni `:w.pf` dan `:w.pe`, masing-masing untuk notasi posisi dan notasi ilmiah.

```
'{:.28f}'.format(k_B) # notasi posisi, 28 karakter
```

Output:

```
'0.0000000000000000000000138065'
```

```
k_B = 1.380649e-23
'{:12.3e}'.format(k_B) # notasi ilmiah 12 karakter, 3 tempat desimal
```

Output:

```
'   1.381e-23'
```

```
'{:.3e}'.format(k_B) # karakter sebanyak yang dibutuhkan, 3 desimal
```

Output:

```
'1.381e-23'
```

Perhatikan bahwa jika lebar dihilangkan, Python membuat string sepanjang yang diperlukan untuk memberikan presisi yang diminta.

Ada beberapa jenis penentu format (*format specifier*) lainnya yang berguna. Di antaranya, `:w.pg` dapat beralih antara notasi posisi dan ilmiah tergantung pada besarnya angka dan presisi yang diminta. Kita bisa juga menggunakan `:ws` untuk menginterpolasi string ke dalam string lain (yang terkadang diperlukan untuk menyelaraskan output).

Sejak Python versi 3.6, kita dapat menginterpolasi nilai objek secara langsung menggunakan *f-strings*. Nama variabel objek ditempatkan sebelum titik dua di dalam string yang didefinisikan dalam tanda kutip yang didahului oleh huruf `f` sebagai berikut (tanda petik ganda akan digunakan sebagai awalan dan penutup karena ada satu petik tunggal sebagai apostrof):

```
f"Boltzmann's constant is approximately {k_B:.3e} J.K-1"
```

Output:

```
"Boltzmann's constant is approximately 1.381e-23 J.K-1"
```

Kita bahkan bisa juga melakukan perhitungan di dalam kurung kurawal interpolasi. Artinya, objek yang diinterpolasi dapat berasal dari sebuah ekspresi. Untuk menjaga kode tetap jelas, lebih baik tidak menyertakan ekspresi kompleks dalam *f-string*, tetapi boleh saja dilakukan misalnya ketika melakukan konversi satuan.

```
print(f'The speed of light is approximately {c/1000:.3f} km.s-1')
```

Output:

```
The speed of light is approximately 299792.458 km.s-1
```

Studi Kasus

Kasus 1 Massa Alkana

Buatlah string terformat yang memberikan massa rantai alkana linear C_nH_{2n+2} jika diberikan n. Gunakan massa molar C dan H berturut-turut sebesar 12.0107 g.mol^{-1} dan 1.00784 g.mol^{-1}.

Solusi

Kita definisikan terlebih dahulu massa molarnya dan *template* string yang akan digunakan:

```
# Massa molar atom karbon dan hidrogen (g.mol-1).
mC, mH = 12.0107, 1.00784

# Template pemformatan string.
fmt = 'Mass of {:5s} : {:6.2f} g.mol-1'
```

Selanjutnya kita hitung untuk $n = 2$ dan $n = 8$:

```
n = 2
nH = 2*n + 2
formula = 'C{}H{}'.format(n, nH) # atau f'C{n}H{nH}'
print(fmt.format(formula, n*mC + nH*mH))

n = 8
nH = 2*n + 2
formula = 'C{}H{}'.format(n, nH)
print(fmt.format(formula, n*mC + nH*mH))
```

Output:

```
Mass of C2H6  :  30.07 g.mol-1
Mass of C8H18 : 114.23 g.mol-1
```

Kasus 2 Beberapa Konstanta Fisis

Buatlah daftar nilai konstanta fisis h, c, k_B, R, dan N_A yang diformat dengan rapi hingga empat angka penting, beserta satuannya.

Solusi

Pertama, definisikan variabel untuk nilai dan satuan konstanta fisis tersebut.

```
h, h_units = 6.62607015e-34, 'J.s'
c, c_units = 299792458, 'm.s-1'
kB, kB_units = 1.380649e-23, 'J.K-1'
R, R_units = 8.314462618, 'J.K.mol-1'
N_A, N_A_units = 6.02214076e+23, 'mol-1'
```

Kita dapat mendefinisikan daftar konstanta dalam satu string melalui penggabungan (*concatenation*). Kita bisa juga memecah penetapan (*assignment*) konstanta ke dalam beberapa baris input, tetapi sisi kanan baris terakhir harus dibungkus dalam tanda kurung agar Python menafsirkannya sebagai ekspresi tunggal.

```
s = (f'h   = {h:9.3e} {h_units}\n'
     f'c   = {c:9.3e} {c_units}\n'
     f'kB  = {kB:9.3e} {kB_units}\n'
     f'R   = {R:9.3f} {R_units}\n'
     f'N_A = {N_A:9.3e} {N_A_units}')
print(s)
```

Output:

```
h   = 6.626e-34 J.s
c   = 3.000e+08 m.s-1
kB  = 1.381e-23 J.K-1
R   =     8.314 J.K.mol-1
N_A = 6.022e+23 mol-1
```

Kasus 3 Pemformatan Data Termodinamika

Variabel-variabel berikut mendefinisikan beberapa sifat termodinamika dari CO_2 dan H_2O

```
# Titik tripel CO2 (K, Pa).
T3_CO2, p3_CO2 = 216.58, 5.185e5
# Entalpi peleburan CO2 (kJ.mol-1).
DfusH_CO2 = 9.019
# Entropi peleburan CO2 (J.K-1.mol-1).
DfusS_CO2 = 40
# Entalpi penguapan CO2 (kJ.mol-1).
DvapH_CO2 = 15.326
# Entropi penguapan CO2 (J.K-1.mol-1).
DvapS_CO2 = 70.8

# Titik tripel H2O (K, Pa).
T3_H2O, p3_H2O = 273.16, 611.73
# Entalpi peleburan H2O (kJ.mol-1).
DfusH_H2O = 6.01
# Entropi peleburan H2O (J.K-1.mol-1).
DfusS_H2O = 22.0
# Entalpi penguapan H2O (kJ.mol-1).
DvapH_H2O = 40.68
# Entropi penguapan H2O (J.K-1.mol-1).
DvapS_H2O = 118.89
```

Gunakan serangkaian perintah `print` dengan *f-strings* untuk menghasilkan tabel terformat semacam ini:

		CO2	H2O
p3	/Pa	518500	611.73
T3	/K	216.58	273.16
DfusH	/kJ.mol-1	9.019	6.010
DfusS	/J.K-1.mol-1	40.0	22.0
DvapH	/kJ.mol-1	15.326	40.680
DvapS	/J.K-1.mol-1	70.8	118.9

Solusi

Kita perlu mengatur lebar kolom dan presisi desimal agar sesuai dengan tabel target. Sebagai contoh, perhatikan bahwa nilai input `118.89` untuk `DvapS_H2O` dibulatkan menjadi `118.9` dalam tampilan tabel, yang menunjukkan penggunaan format `.1f`. Format string lainnya dapat kita sesuaikan.

```python
# Header Tabel
print(f"{'':<6} {'':<14} {'CO2':>10} {'H2O':>10}")

# Baris p3 (Pa): CO2 integer, H2O 2 desimal
print(f"{'p3':<6} {'/Pa':<14} {p3_CO2:10.0f} {p3_H2O:10.2f}")

# Baris T3 (K): 2 desimal
print(f"{'T3':<6} {'/K':<14} {T3_CO2:10.2f} {T3_H2O:10.2f}")

# Baris Entalpi Peleburan (kJ.mol-1): 3 desimal
print(f"{'DfusH':<6} {'/kJ.mol-1':<14} {DfusH_CO2:10.3f}
↪ {DfusH_H2O:10.3f}")

# Baris Entropi Peleburan (J.K-1.mol-1): 1 desimal
print(f"{'DfusS':<6} {'/J.K-1.mol-1':<14} {DfusS_CO2:10.1f}
↪ {DfusS_H2O:10.1f}")

# Baris Entalpi Penguapan (kJ.mol-1): 3 desimal
print(f"{'DvapH':<6} {'/kJ.mol-1':<14} {DvapH_CO2:10.3f}
↪ {DvapH_H2O:10.3f}")

# Baris Entropi Penguapan (J.K-1.mol-1): 1 desimal
print(f"{'DvapS':<6} {'/J.K-1.mol-1':<14} {DvapS_CO2:10.1f}
↪ {DvapS_H2O:10.1f}")
```

Output:

```
                             CO2        H2O
p3     /Pa                518500     611.73
T3     /K                 216.58     273.16
DfusH  /kJ.mol-1           9.019      6.010
DfusS  /J.K-1.mol-1         40.0       22.0
DvapH  /kJ.mol-1          15.326     40.680
DvapS  /J.K-1.mol-1         70.8      118.9
```

Serial Python Minimalis:
Tutorial #03-List dan Loop

Ahmad R. T. Nugraha

ver. 21 April 2026

1 Definisi, Sintaksis, dan Penggunaan

Untuk memahami konsep `list` dan `loop`, kita kembali pada konsep string yang dipelajari dalam tutorial sebelumnya. Pada dasarnya, string adalah salah satu contoh barisan (*sequence*) atau kumpulan objek (dalam hal ini, karakter) yang dapat diambil satu per satu (alias di-"iterasi"). Sintaksis untuk mengambil objek dari sebuah barisan adalah:

```
for objek in barisan:
    # Lakukan sesuatu dengan objek
```

Bentuk ini merupakan perulangan `for` (*for loop*). Dalam bahasa Python, blok kode yang diindentasi dengan empat spasi dieksekusi satu kali untuk setiap objek dalam barisan tersebut. Contohnya:

```
name = 'tin'
for letter in name:
    print(letter)
```

Output:

```
t
i
n
```

Bentuk barisan Python penting lainnya adalah `list`, yaitu kumpulan objek terurut yang dapat berupa tipe apa pun. `list` didefinisikan sebagai barisan objek yang dipisahkan koma di antara tanda kurung siku:

```
mylist = [4, 'Sn', 'sodium', 3.14159]
```

`list` dapat diindeks, dipotong (*sliced*), dan diiterasi sama seperti string:

```
print(mylist[2])
```

Output:

```
sodium
```

```
# Semua item mulai dari indeks ketiga dan seterusnya
print(mylist[2:])
```

Output:

```
['sodium', 3.14159]
```

```
# Mengiterasi semua item dalam mylist
for item in mylist:
    print(item)
```

Output:

```
4
Sn
sodium
3.14159
```

Tidak seperti string, `list` bersifat *mutable*, yakni dapat diubah di tempat (*in place*) dengan berbagai cara, misalnya melalui penetapan langsung:

```
mylist[2] = 'potassium'
mylist
```

Output:

```
[4, 'Sn', 'potassium', 3.14159]
```

Ada beberapa metode penting lainnya untuk memanipulasi `list`. Metode `append` menambahkan sebuah item ke akhir list (menambah panjangnya sebanyak satu):

```
mylist.append(-42)
mylist
```

Output:

```
[4, 'Sn', 'sodium', 3.14159, -42]
```

Metode `extend` dapat digunakan untuk menggabungkan dua `list`:

```
list1 = ['H', 'He', 'Li']
list2 = ['Be', 'B', 'C']
list1.extend(list2)
list1
```

Output:

```
['H', 'He', 'Li', 'Be', 'B', 'C']
```

`list` baru dapat dibuat dengan melewatkan objek teriterasi (*iterable*) apa pun ke dalam konstruktor `list()`:

```
list('helium')
```

Output:

```
['h', 'e', 'l', 'i', 'u', 'm']
```

Angka individual tidak *iterable*, sehingga `list(3.14159)` akan memberikan pesan *error*:

```
list(3.14159)
```

Output:

```
TypeError                                 Traceback (most recent call last)
<ipython-input-32-8870af92a665> in <module>
----> 1 list(3.14159)

TypeError: 'float' object is not iterable
```

Namun, fungsi `str(3.14159)` yang merupakan representasi string dari angka ini, "3.14159", bersifat *iterable* sehingga kita dapat membuat `list` dari karakter-karakternya:

```
list(str(3.14159))
```

Output:

```
['3', '.', '1', '4', '1', '5', '9']
```

2 Fungsi `range` dan `enumerate`

Untuk menghasilkan barisan bilangan bulat dengan jarak yang teratur (sebuah deret aritmetika), terdapat fungsi bawaan yang disebut `range`. Dengan argumen tunggal, `range(n)` menghasilkan bilangan bulat 0, 1, 2, ... n-1:

```
for i in range(4):
    print(i, i**2, i**3)
```

Output:

```
0 0 0
1 1 1
2 4 8
3 9 27
```

Perhatikan bahwa barisan ini mengikuti aturan umum dimulai dari 0 dan berakhir pada $n - 1$ sehingga cocok digunakan untuk menghasilkan indeks pada urutan lain seperti `list` dan string. Ingat kembali contoh iterasi karakter dalam sebuah string:

```
for letter in 'tin':
    print(letter)
```

Output:

```
t
i
n
```

Contoh di sini menunjukkan ekspresifnya Python sebagai bahasa pemrograman. Dalam bahasa pemrograman lain, kita perlu melacak indeks ke dalam barisan dan menggunakannya untuk mengambil item satu per satu, sesuatu yang dapat langsung dicapai dengan fungsi `range`:

```
name = 'tin'
for i in range(len(name)):
    # yaitu, i = 0, 1, 2
    print(name[i])
```

Output:

```
t
i
n
```

Meski begitu, penggunaan `range` semacam ini tidak dianjurkan di Python karena ada sintaksis `for letter in name:` yang lebih sederhana dan jelas lebih mudah dipahami dan dikelola. Jika ada alasan tertentu yang membuat kita menginginkan indeks dari setiap item serta item itu sendiri, terdapat fungsi bawaan lainnya untuk keperluan tersebut, yaitu `enumerate`, yang mengembalikan keduanya dan dapat diekstrak (*unpacked*) menjadi dua variabel di dalam perulangan:

```
for i, letter in enumerate('zinc'):
    print(i, ':', letter)
```

Output:

```
0 : z
1 : i
2 : n
3 : c
```

3 Membuat `list`

Karena `list` dapat dikembangkan di tempat, salah satu cara untuk membuatnya adalah dengan menambahkan (`append`) elemen-elemennya satu per satu:

```
my_list = []
for i in range(5):
    my_list.append((i + 1)**2)
my_list
```

Output:

```
[1, 4, 9, 16, 25]
```

Namun, terdapat sintaksis yang lebih ringkas dan umum digunakan dalam kasus ini yang disebut *list comprehension*:

```
my_list = [(i + 1)**2 for i in range(5)]
my_list
```

Output:

```
[1, 4, 9, 16, 25]
```

Ekspresi yang cukup kompleks dapat digunakan dalam *list comprehension*, tetapi sebaiknya pastikan bahwa kode tersebut tetap mudah dipahami. Untungnya, Python cukup ekspresif sehingga beberapa fungsi dapat dirangkai tanpa kehilangan keterbacaan. Misalnya, untuk menghitung $\sin(\theta)$ pada beberapa sudut, θ:

```
import math
angles = [0, 30, 60, 90, 120, 150, 180]
sines = [round(math.sin(math.radians(theta)), 3) for theta in angles]
sines
```

Output:

```
[0.0, 0.5, 0.866, 1.0, 0.866, 0.5, 0.0]
```

4 Metode `split` dan `join`

Dua metode berguna ini bekerja pada string. Untuk memecah string menjadi `list` yang berisi substring berdasarkan pemisah (*separator*) tertentu, panggil `split()`:

```
'a,b,c'.split(',')
```

Output:

```
['a', 'b', 'c']
```

```
'a::b::c'.split(':')
```

Output:

```
['a', '', 'b', '', 'c']
```

```
'a::b::c'.split('::')
```

Output:

```
['a', 'b', 'c']
```

Jika kita ingin memecahnya berdasarkan spasi kosong dalam jumlah berapa pun (spasi, tab), tidak perlu memberikan argumen pada `split()`:

```
'a b   c'.split()
```

Output:

```
['a', 'b', 'c']
```

Metode `join` melakukan hal sebaliknya, yakni menggabungkan urutan string dengan pemisah (*separator*) yang ditentukan:

```
'-'.join(['a', 'b', 'c'])
```

Output:

```
'a-b-c'
```

Urutan tersebut tidak harus berupa `list`. String adalah urutan karakter (yang dengan sendirinya adalah string), sehingga hal berikut juga berlaku:

```
'-'.join('abc')
```

Output:

```
'a-b-c'
```

5 Fungsi `zip`

Terkadang kita perlu mengiterasi dua urutan atau lebih pada saat yang bersamaan, yang dapat dilakukan dengan pengindeksan langsung ke dalamnya. Sebagai contoh:

```
list1 = [1, 2, 3]
list2 = ['a', 'b', 'c']
for i in range(len(list1)):
    print(list1[i], list2[i])
```

Output:

```
1 a
2 b
3 c
```

Fungsi bawaan `zip` memungkinkan iterasi tanpa pengindeksan eksplisit. Saat diulang, fungsi ini akan mengeluarkan item-item yang cocok dari setiap urutan secara bergantian:

```
for number, letter in zip(list1, list2):
    print(number, letter)
```

Output:

```
1 a
2 b
3 c
```

Jika urutan-urutan tersebut memiliki panjang yang berbeda, `zip` akan mengembalikan elemen-elemen yang cocok sampai urutan yang paling pendek habis:

```
seq1 = 'abcdefghijk'
seq2 = range(1000)
seq3 = ['Be', 'Mg', 'Ca', 'Sr']
for letter, number, symbol in zip(seq1, seq2, seq3):
    print(letter, number, symbol)
```

Output:

```
a 0 Be
b 1 Mg
c 2 Ca
d 3 Sr
```

Studi Kasus

Kasus 1 Simbol dan Nomor Atom

Buatlah tabel yang berisi 10 simbol unsur pertama beserta nomor atomnya, Z.

Solusi

Karena indeks Python dimulai dari 0, kita harus menambahkan 1 pada indeks urutan `list` simbol unsur untuk mendapatkan Z:

```python
symbols = ['H', 'He', 'Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'Ne']

print('Element | Z')
print('--------|---')
for Z, symbol in enumerate(symbols):
    print('{:7s} | {:2d}'.format(symbol, Z + 1))
```

Output:

```
Element | Z
--------|---
H       |  1
He      |  2
Li      |  3
Be      |  4
B       |  5
C       |  6
N       |  7
O       |  8
F       |  9
Ne      | 10
```

Sebagai alternatif, `enumerate` dapat menerima argumen opsional, `start`, yang menetapkan bilangan bulat pertama untuk dikembalikan sebagai hitungan awal:

```python
for Z, symbol in enumerate(symbols, start=1):
    print('{:7s} | {:2d}'.format(symbol, Z))
```

Kasus 2 Massa Molar Udara Kering

Hitunglah rata-rata massa molar udara kering, berdasarkan komposisi dan massa yang tercantum pada Tabel 4.1.

Molekul	Massa / Da	Komposisi / %
N_2	28.0134	78.084
O_2	31.9898	20.946
Ar	39.948	0.9290
CO_2	44.0095	0.041

Solusi

Kita mendefinisikan data ke dalam `list`. Kemudian, kita perlu mengiterasi dua `list` secara bersamaan menggunakan `zip` untuk menghitung rata-ratanya:

$$\overline{M}_{air} = \sum_i x_i M_i$$

dengan x_i adalah fraksi jumlah volume dari konstituen udara ke-i, dengan massa molar M_i.

```
# Konstituen udara kering:
# molekul, massa (g.mol-1), persentase komposisi volume
molecule = ['N2', 'O2', 'Ar', 'CO2']
mass = [28.0134, 31.9898, 39.948, 44.0095]
composition = [78.084, 20.946, 0.9290, 0.041]

mean_air_mass = 0
for m, x_perc in zip(mass, composition):
    mean_air_mass += m * x_perc / 100

print('Mean mass of dry air: {:.2f} g.mol-1'.format(mean_air_mass))
```

Output:

```
Mean mass of dry air: 28.96 g.mol-1
```

Kasus 3 Gaya Apung Balon

Sebuah balon, dengan massa 0.8 g saat belum ditiup, diisi dengan gas helium hingga bervolume 2 L pada tekanan lingkungan 1 atm. Berapakah gaya netto (*net force*) pada balon pada suhu-suhu berikut: $-20, 0, 25, 40°C$?
Ambil $M(\text{He}) = 4$ g.mol^{-1}, massa molar rata-rata udara $\overline{M}_{air} = 29$ g.mol^{-1}, dan percepatan gravitasi $g = 9.8$ m.s^{-2}.

Solusi

Memperlakukan helium sebagai gas ideal, jumlah molnya adalah

$$n = pV/RT$$

dengan tekanan, p, dapat dianggap sama dengan tekanan lingkungan (mengabaikan efek kelengkungan permukaan balon yang mengembang).
Ada dua gaya pada balon: (1) beratnya, $F_g = -mg$, dengan $m = m_{balloon} + m_{He}$, dan (2) gaya apung karena udara yang dipindahkan, $F_b = m_{air}g$. Gaya nettonya adalah $F = F_g + F_b$ dengan nilai negatif F menunjukkan balon tenggelam (gaya

diarahkan ke bawah), dan nilai positif menunjukkan balon naik.
Pertama, definisikan variabel untuk konstanta dan parameter masalah ini:

```
# Konstanta gas (J.K-1.mol-1) dan percepatan gravitasi (m.s-2).
R = 8.314
g = 9.8
# Tekanan (Pa).
p = 101325
# Volume balon mengembang (m3).
V = 2.e-3
# Massa balon belum ditiup (kg).
m_balloon = 0.8 / 1000
# Massa molar helium dan rata-rata massa molar udara (g.mol-1).
M_He = 4
M_air = 29

# Suhu dalam degC
temperatures = [-20, 0, 25, 40]
```

Sekarang perhitungan gaya netto dapat dilakukan di dalam perulangan untuk berbagai suhu:

```
for T in temperatures:
    # Jumlah (mol) dan massa (kg) He dalam balon.
    n_He = p * V / R / (T + 273)
    m_He = M_He / 1000 * n_He

    # Udara yang dipindahkan memiliki jumlah mol yang sama, tapi lebih
    ↪ berat.
    m_air = M_air / 1000 * n_He

    # Gaya netto
    F = (m_air - m_He - m_balloon) * g
    print('{:3d} deg C : {:.4f} N'.format(T, F))
```

Output:

```
-20 deg C : 0.0158 N
  0 deg C : 0.0140 N
 25 deg C : 0.0122 N
 40 deg C : 0.0112 N
```

Perulangan bisa dibuat bersarang (*nested*). Misalnya, kita dapat mengulangi perhitungan tersebut untuk gas Ne dan Ar juga:

```python
# Gas dan massa molarnya
gases = ['He', 'Ne', 'Ar']
gas_masses = [4, 20, 40]

for T in temperatures:
    n = p * V / R / (T + 273)
    m_air = M_air / 1000 * n

    for gas, M_gas in zip(gases, gas_masses):
        m_gas = M_gas / 1000 * n
        F = (m_air - m_gas - m_balloon) * g
        print('{} at {:3d} deg C: Force = {:8.5f} N'.format(gas, T, F))
```

Output:

```
He at -20 deg C: Force =  0.01576 N
Ne at -20 deg C: Force =  0.00066 N
Ar at -20 deg C: Force = -0.01823 N
He at   0 deg C: Force =  0.01403 N
Ne at   0 deg C: Force =  0.00003 N
Ar at   0 deg C: Force = -0.01746 N
He at  25 deg C: Force =  0.01220 N
Ne at  25 deg C: Force = -0.00063 N
Ar at  25 deg C: Force = -0.01666 N
He at  40 deg C: Force =  0.01124 N
Ne at  40 deg C: Force = -0.00097 N
Ar at  40 deg C: Force = -0.01623 N
```

Dengan sedikit modifikasi, kita dapat menggunakan `print` untuk menghasilkan tabel dari hasil tersebut. Gunakan parameter `end=`'' untuk menekan perilaku *default* `print` yang selalu menambahkan baris baru di akhir.

```python
print('Net force on balloon (in newtons; +ve is up)')
print('  T      He          Ne          Ar')

for T in temperatures:
    n = p * V / R / (T + 273)
    m_air = M_air / 1000 * n
    print('{:3d} °C'.format(T), end='')

    for gas, M_gas in zip(gases, gas_masses):
        m_gas = M_gas / 1000 * n
        F = (m_air - m_gas - m_balloon) * g
        print('{:12.5f}'.format(F), end='')

    # Mencetak baris baru untuk suhu berikutnya
    print()
```

Output:

```
Net force on balloon (in newtons; +ve is up)
  T      He          Ne          Ar
-20 C     0.01576     0.00066    -0.01823
  0 C     0.01403     0.00003    -0.01746
 25 C     0.01220    -0.00063    -0.01666
 40 C     0.01124    -0.00097    -0.01623
```

Serial Python Minimalis:
Tutorial #04-Kontrol Alur

Ahmad R. T. Nugraha

ver. 21 April 2026

Sejauh ini, kode yang telah kita tulis dieksekusi dalam urutan ("alur") yang telah ditentukan sebelumnya, yakni satu pernyataan dieksekusi setelah pernyataan lainnya, atau diulang dalam jumlah yang tetap dalam kasus perulangan `for`. Namun, untuk program-program yang lebih kompleks, sering kali kita memerlukan kode tersebut dapat bercabang, yakni mengeksekusi pernyataan kode yang berbeda berdasarkan satu atau lebih kondisi. Oleh karena itu, kita harus mengetahui beberapa teknik kontrol alur program dalam Python.

1 Perbandingan dan Logika

Dalam Python, kontrol alur di antaranya dapat dicapai menggunakan sintaksis `if...elif...else` dengan perbandingan antarobjek dievaluasi sebagai `True` (Benar) atau `False` (Salah). Daftar operator perbandingan yang berguna disajikan pada Tabel 1. Perhatikan bahwa untuk membandingkan kesetaraan dua objek (misalnya angka), operatornya adalah ==: dua tanda sama dengan. Operator ini jangan sampai disamakan dengan satu tanda sama dengan (texttt=) yang spesifik digunakan untuk penetapan atau penugasan (*assignment*).

Tabel 1: Operator perbandingan Python

Operator	Arti
==	Sama dengan
!=	Tidak sama dengan
>	Lebih besar dari
<	Lebih kecil dari
>=	Lebih besar dari atau sama dengan
<=	Lebih kecil dari atau sama dengan

Mari kita lihat beberapa contoh:

```
a = 4
a == 2 + 2
```

Output:

```
True
```

```
a >= 5
```

Output:

```
False
```

Perbandingan dapat digabungkan dengan kata kunci operator logika `and`, `not`, dan `or`:

```
a, b, c = 1, 2, 3
a > b or c > b
```

Output:

```
True
```

```
a > b and c > b
```

Output:

```
False
```

```
not a > b
```

Output:

```
True
```

Dalam kontrol alur menggunakan operator perbandingan dan logika, ada prioritas urutan (*precedence*) yang perlu diperhatikan. Aturan dasarnya, operasi aritmetika (+, ∗, dll.) selalu memiliki presedensi lebih tinggi (dievaluasi lebih dulu) daripada perbandingan (==, <, dll.), dan perbandingan selalu memiliki presedensi lebih tinggi daripada operator logika. Di antara operator logika, presedensinya dari `not` (tertinggi) hingga `or` (terendah). Seperti halnya aritmetika, ekspresi di dalam tanda kurung memiliki presedensi tertinggi dari semuanya.

```
not a == 1 and b > c
```

Output:

```
False
```

```
not (a == 1 and b > c)
# (not True) and False => False
```

```
# not (True and False) => (not False) => True
```

Output:

```
True
```

Ada beberapa "operator keanggotaan" (*membership operators*) yang kerap berguna untuk keseharian pemrograman kita, di antaranya `in` dan `not in`, masing-masing menguji keanggotaan dan ketiadaan anggota dalam sebuah barisan:

```
metals = 'Sc Ti V Cr Mn Fe Co Ni Cu Zn'.split()
metals
```

Output:

```
['Sc', 'Ti', 'V', 'Cr', 'Mn', 'Fe', 'Co', 'Ni', 'Cu', 'Zn']
```

```
'Fe' in metals
```

Output:

```
True
```

```
'V' not in metals
```

Output:

```
False
```

Dua fungsi bawaan lainnya berguna untuk perbandingan yang melibatkan semua item dalam sebuah barisan: `any` mengembalikan `True` jika salah satu item setara dengan `True`; `all` mengembalikan `True` jika semuanya setara dengan `True`:

```
all(['a', True, 2])
```

Output:

```
True
```

```
any([0, [], '', False])
```

Output:

```
False
```

2 Konstruksi if ... elif ... else

Kode di dalam blok kode if (diindentasi 4 spasi, seperti perulangan for) hanya dieksekusi jika ekspresi setelah kata kunci if terevaluasi sebagai True:

```
a = 3.14159
if a > 3:
    print('a is greater than 3')
```

Output:

```
a is greater than 3
```

```
if a > 4:
    print('a is greater than 4')
```

Kata kunci else menyediakan jalur alternatif jika ekspresinya tidak True:

```
if a < 3:
    print('a is less than 3')
else:
    print('a is greater than 3')
```

Output:

```
a is greater than 3
```

Kata kunci elif kemudian memungkinkan beberapa kondisi untuk diperiksa. Blok kode yang mengikuti perbandingan True pertama akan dieksekusi:

```
if a < 3:
    print('a is less than 3')
elif a < 4:
    print('a is between 3 and 4')
else:
    print('a is greater than 4')
```

Output:

```
a is between 3 and 4
```

True dan False adalah nilai boolean konstan bawaan Python (bertipe bool). Nilai-nilai ini tidak dapat diubah atau ditetapkan sebagai nama variabel. Untuk tujuan perbandingan, Python juga memiliki konsep *truthiness* (kebenaran), yakni bahwa objek non-boolean diperlakukan sebagai True atau False berdasarkan nilainya. Secara khusus, float atau int dengan nilai 0 setara dengan False (semua nilai lainnya adalah True); string kosong ('') dan list kosong ([]) bernilai False (string atau list apa pun yang memiliki isi setara dengan True). Konsep inimemungkinkan sintaksis seperti:

```
b = 10
if b % 2:
    print('b is odd')
else:
    print('b is even')
```

Output:

```
b is even
```

Di sini, jika `b` ganjil, `b % 2` adalah 1 (yaitu, `b` mod 2 = 1, sisa bagi dari `b/2`) yang setara dengan `True`. Jika `b` genap, maka `b % 2` adalah 0, yang setara dengan `False`.

3 Perulangan `while`

Cara alternatif untuk membuat perulangan adalah dengan kata kunci `while`. Dari pelajaran sebelumnya, kita tahu bahwa perulangan `for` dieksekusi untuk jumlah iterasi yang tetap. Di sini kita akan lihat bahwa perulangan `while` hanya akan mengeksekusi blok kode selama kondisinya masih terpenuhi (`True`). Misalnya:

```
i = 0
while i < 3:
    i += 1
    print(i)
```

Output:

```
1
2
3
```

Pencacah di sini (*counter*) `i` diinisialisasi ke 0, yang tentunya lebih kecil dari 3, sehingga bagian badan perulangan `while` dieksekusi. Pertama-tama `i` bertambah dan nilainya (1) dicetak ke layar. Eksekusi kemudian kembali ke pengujian perulangan `while`. Semua ini berulang sampai `i` mencapai nilai 3, yang pada situasi ini nilainya dicetak, tetapi sekarang pengujian perulangan `while`, `i < 3`, memberikan hasil `False` sehingga eksekusi kode keluar atau selesai dari bagian badan perulangan tersebut.

4 Kontrol Alur Lainnya: `break`, `continue`, dan `pass`

Ketiga kata kunci ini memberikan kontrol lebih lanjut terhadap eksekusi perulangan. Saat pernyataan `break` tercapai, perulangan akan segera dihentikan:

```
for i in range(10):
    print(i)
    if i == 3:
```

```
        break
```

Output:

```
0
1
2
3
```

Objek `range(10)` akan menghasilkan angka 0, 1, 2, ..., 9 dan mengumpankannya ke `i` secara bergantian, tetapi ketika `i` mencapai 3, maka perulangan diakhiri. Penggunaan `break` yang umum adalah untuk keluar dari perulangan tak terbatas (*infinite loop*):

```
n = 0
while True:
    n += 1
    if not (n % 12 or n % 42):
        break
print(n)
```

Output:

```
84
```

Kode di atas mencetak angka terkecil, yakni kelipatan persekutuan terkecil (KPK), yang habis dibagi oleh 12 dan 42.

Ketika pernyataan `continue` ditemui, iterasi perulangan saat ini akan ditinggalkan dan iterasi berikutnya (jika ada) dimulai. Seperti `break`, kata kunci `continue` dapat digunakan di dalam perulangan `for` maupun `while`:

```
for n in range(10):
    if n % 3:
        continue
    print(n)
```

Output:

```
0
3
6
9
```

Di sini, pernyataan `print` hanya dieksekusi jika `n % 3` bernilai 0 (setara dengan `False`), yaitu jika 3 membagi `n`. Jika tidak, `n % 3` setara dengan `True` dan pernyataan `continue` dieksekusi, segera mengembalikan alur kode ke bagian atas perulangan `for`, yang kemudian menambah nilai `n`.

5 Eksepsi/Pengecualian (*Exceptions*)

Kondisi kesalahan yang ditemui dalam kode Python dilaporkan sebagai pengecualian (*exceptions*), yang biasanya disertai laporan pelacakan balik (*traceback*) yang dapat membantu melacak penyebab kesalahan. Misalnya:

```
print(float("2.3D-2"))
```

Output:

```
ValueError                                Traceback (most recent call last)
Input In [x], in <module>
----> 1 print(float("2.3D-2"))

ValueError: could not convert string to float: '2.3D-2'
```

Exception yang dilaporkan di sini adalah `ValueError`: objek yang diteruskan ke fungsi `float` adalah sebuah string, `"2.3D-2"`, yang merupakan tipe yang benar, tetapi nilainya mengandung karakter yang tidak valid (`'D'`), sehingga `float` tidak dapat mengubahnya menjadi bilangan riil titik kambang (*floating-point*). Beberapa *Exception* yang umum ditemui dapat dibaca pada Tabel 2.

Tabel 2: Pengecualian (Exceptions) Python yang umum

Exception	Penyebab dan deskripsi
`IndexError`	Mengindeks sebuah urutan (seperti list atau string) dengan subskrip yang berada di luar rentang.
`KeyError`	Mengindeks sebuah kamus (*dictionary*) dengan kunci yang tidak ada di dalam tersebut. Struktur kamus akan kita pelajari belakangan.
`NameError`	Merujuk ke nama variabel yang belum didefinisikan.
`TypeError`	Mencoba menggunakan objek dengan tipe yang tidak sesuai sebagai argumen untuk operasi atau fungsi bawaan.
`ValueError`	Mencoba menggunakan objek dengan tipe yang benar tetapi dengan nilai yang tidak kompatibel sebagai argumen untuk operasi atau fungsi bawaan.
`ZeroDivisionError`	Mencoba membagi dengan nol (baik secara eksplisit menggunakan "/" atau "//", maupun sebagai bagian dari operasi modulo "%").
`SyntaxError`	Menggunakan sintaksis Python yang keliru, misalnya mencoba menetapkan kata kunci tercadang (*reserved*) sebagai nama variabel, menghilangkan tanda kurung tutup, atau menggunakan satu tanda sama dengan (=) untuk perbandingan alih-alih ==.

Python yang idomatik menganut filosofi yang diringkas dengan singkatan EAFP: *"Easier to Ask Forgiveness than to seek Permission"* ("Lebih Mudah Meminta Maaf daripada Meminta Izin"). Daripada mencoba menghindari *Exceptions*, pada praktiknya sangat umum untuk tetap menjalankan kode tersebut, tetapi di dalam sebuah blok `try`, yang diikuti oleh blok `except` yang menangani setiap kesalahan yang diantisipasi. Misalnya:

```
x = 0
try:
    print(4/x)
except ZeroDivisionError:
    print("4/0 is not defined!")
```

Output:

```
4/0 is not defined!
```

Eksekusi kode kemudian berlanjut secara normal setelah blok `try...except` ini, terlepas dari apakah nilai $4/x$ ataupun pesan peringatannya yang dicetak. Kesalahan yang tidak terantisipasi diharapkan untuk menghentikan eksekusi kode dan dilaporkan kepada pemrogram/pengguna.

6 Studi Kasus

Kasus 1 Wujud Zat Logam

Diketahui data berikut mengenai titik leleh (*melting points*) dan titik didih (*boiling points*) standar dari berbagai logam.

```
metals = ['Fe', 'Al', 'Hg', 'Ti', 'Pb']
# Suhu perubahan wujud standar untuk setiap logam, dalam Kelvin.
melting_points = [1811, 933, 234, 1941, 601]
boiling_points = [3134, 2743, 630, 3560, 2022]
```

Tentukan wujudnya (padat, cair, atau gas) pada suhu 500°C.

Solusi

Pertama, konversikan suhu rujukan, `T`, ke Kelvin, lalu iterasi ketiga `list` tersebut secara bersamaan. Untuk setiap logam, bandingkan `T` dengan titik didih (`Tb`) dan titik lelehnya (`Tm`).

```python
# Suhu (K)
T = 500 + 273

for metal, Tm, Tb in zip(metals, melting_points, boiling_points):
    if T > Tb:
        print(f'{metal} is a gas at {T} K')
    elif T > Tm:
        print(f'{metal} is a liquid at {T} K')
    else:
        print(f'{metal} is a solid at {T} K')
```

Output:

```
Fe is a solid at 773 K
Al is a solid at 773 K
Hg is a gas at 773 K
Ti is a solid at 773 K
Pb is a liquid at 773 K
```

Untuk menghasilkan `list` yang berisi nilai boolean, `True` atau `False`, yang menunjukkan apakah `T` lebih kecil dari titik didih, kita dapat menggunakan metode append:

```python
is_gas = []
for Tb in boiling_points:
    if T > Tb:
        is_gas.append(True)
    else:
        is_gas.append(False)
is_gas
```

Output:

```
[False, False, True, False, False]
```

Namun, kita dapat sepenuhnya menghilangkan blok `if...else` tersebut. Karena kita menambahkan hasil dari perbandingan `T > Tb` (`True` atau `False`), cara yang lebih baik adalah:

```python
is_gas = []
for Tb in boiling_points:
    is_gas.append(T > Tb)
is_gas
```

Output:

```
[False, False, True, False, False]
```

Bahkan lebih baik lagi, kita dapat membuat `list` ini tanpa harus menggunakan metode `append`: gunakan *list comprehension* untuk menghasilkan list tersebut:

```
is_gas = [T > Tb for Tb in boiling_points]
is_gas
```

Output:

```
[False, False, True, False, False]
```

Fungsi bawaan `any` dan `all` masing-masing dapat digunakan untuk memastikan apakah ada salah satu atau seluruh logam yang berwujud gas pada suhu `T`:

```
any(is_gas), all(is_gas)
```

Output:

```
(True, False)
```

Kasus 2 Volume Molar Gas van der Waals

Tentukan volume molar, V_m, dari uap heksana pada $T = 25°\text{C}$ dan $p = 10\,\text{kPa}$. Gunakan persamaan keadaan van der Waals dengan parameter $a = 2.471\,\text{m}^6\,\text{Pa}\,\text{mol}^{-2}$ dan $b = 1.735 \times 10^{-4}\,\text{m}^3\,\text{mol}^{-1}$.
Persamaan keadaan van der Waals adalah

$$p = \frac{RT}{V_m - b} - \frac{a}{V_m^2}$$

Solusi

Pikiran pertama kita mungkin langsung mengatur ulang persamaan ini untuk V_m:

$$pV_m^3 - (bp + RT)V_m^2 + aV_m - ab = 0$$

Sayangnya, ini adalah persamaan kubik dalam V_m, dan tidak ada cara mudah untuk menemukan akar (yang relevan) secara langsung.
Namun, ada pendekatan numerik yang didasarkan pada iterasi. Pengaturan ulang yang berbeda memberikan:

$$V_m = b + \frac{RT}{p + \frac{a}{V_m^2}}$$

dengan suku interaksi antarmolekul, $\frac{a}{V_m^2} \ll p$. Jika kita mulai dari tebakan awal, $V_m^{(0)}$, kita dapat mengharapkan pendekatan yang secara progresif menjadi lebih

baik ke nilai V_m sebenarnya dari proses iteratif:

$$V_m^{(n+1)} = b + \frac{RT}{p + \frac{a}{(V_m^{(n)})^2}}$$

Tebakan awal yang wajar adalah $V_m^{(0)} = \frac{RT}{p}$, yaitu volume molar yang diprediksi oleh persamaan gas ideal.
Definisikan konstanta dan parameter yang kita butuhkan:

```
# Konstanta gas, dalam J.K-1.mol-1.
R = 8.314
# Parameter Van der Waals untuk heksana: m6.Pa.mol-2 dan m3.mol-1.
a, b = 2.471, 1.735e-4
# Suhu (K) dan tekanan (Pa) untuk soal ini.
T, p = 25 + 273.15, 10 * 1000

# Volume molar gas ideal, dalam m3:
Vm = R * T / p
Vm
```

Output:

```
0.0024788191
```

Kita akan mengiterasi sampai nilai V_m berubah kurang dari jumlah yang sangat kecil, `TOL`.

```
TOL = 1.e-12
Vold = R * T / p

while True:
    Vm = b + R * T / (p + a / Vold**2)
    print(Vm)
    if abs(Vm - Vold) < TOL:
        break
    Vold = Vm

print()
print('Vm = {:.4f} L.mol-1'.format(Vm * 1000))
```

Output:

```
0.00023364429019125273
0.00017404750299408598
0.0001738038466444478
0.00017380299660870434
0.00017380299364529655
0.00017380299363496549

Vm = 0.1738 L.mol-1
```

Terakhir, kita harus memeriksa bahwa prosedur ini benar-benar konvergen ke jawaban yang tepat dengan memasukkan nilai V_m ini kembali ke persamaan gas van der Waals:

```
R * T / (Vm - b) - a / Vm**2
```

Output:

```
9999.990276411176
```

Hasil ini sangat dekat dengan nilai yang diharapkan, yaitu $p = 10$ kPa.

Kasus 3 Verifikasi Konfigurasi Elektron

Diberikan sebuah string yang mewakili konfigurasi elektron suatu atom dalam bentuk '1s2.2s2.2p4'. Tuliskan sebuah program untuk memverifikasi bahwa konfigurasi tersebut valid. Pengujian berikut harus dilakukan:

1. Orbital harus benar-benar ada (contoh: '1s2.2d3' tidak valid).
2. Orbital tidak boleh diduplikasi (contoh: '1s2.2s1.2s1' tidak valid).
3. Orbital tidak boleh berisi elektron lebih dari jumlah maksimum yang diizinkan untuk orbital tersebut.

Solusi

Konfigurasi orbital atom ditulis sebagai nl^N dengan $n = 1, 2, 3, \ldots$ adalah bilangan kuantum utama, $l = 0, 1, \ldots, n-1$ adalah bilangan kuantum momentum sudut orbital yang ditulis sebagai huruf (s, p, d, dll.), dan N adalah keterisian orbital. Degenerasi spasial orbital adalah $2l + 1$, dan masing-masing bilangan kuantum proyeksi spin elektron dapat bernilai $m_s = \pm\frac{1}{2}$ sehingga nilai maksimum N adalah $2(2l + 1)$.
Kita akan menyusun kodenya langkah demi langkah, dan mengujinya dengan

string konfigurasi yang (sangat) tidak valid. Karena setiap orbital yang terisi dipisahkan oleh karakter '.', kita bisa mengubahnya menjadi list dengan metode `split`:

```
# Sebuah konfigurasi atom yang sengaja disalahkan.
s = '1p2.2s2.2p6.2s3.4d10'
s.split('.')
```

Output:

```
['1p2', '2s2', '2p6', '2s3', '4d10']
```

Selanjutnya, urai (*parse*) string orbital yang terisi untuk mengekstrak n, l, dan N. Karena huruf `l` terlihat mirip dengan angka `1` di beberapa font, kita akan menamai variabelnya `ell`; untuk lebih menghindari kebingungan, bilangan kuantum utama dinamai `n`, dan jumlah elektron N dinamai `n_electrons`.

```
# Pisahkan string untuk setiap orbital; dan lakukan loop
for occ_orbital in s.split('.'):
    # Orbital adalah dua karakter pertama dari substring
    # Catatan: ini berarti kita tidak bisa memiliki n > 9
    orbital = occ_orbital[:2]
    n = int(orbital[0])
    ell_letter = orbital[1]

    # Cara termudah mengonversi huruf ke integer ell: temukan
    ↪ indeksnya
    ell = 'spdf'.index(ell_letter)
    n_electrons = int(occ_orbital[2:])
    print(f'n={n}, l={ell} ("{ell_letter}"), N={n_electrons}')
```

Output:

```
n=1, l=1 ("p"), N=2
n=2, l=0 ("s"), N=2
n=2, l=1 ("p"), N=6
n=2, l=0 ("s"), N=3
n=4, l=2 ("d"), N=10
```

Hasilnya terlihat bagus. Sekarang periksa apakah $l < n$ dan $N \leq 2(2l+1)$, serta simpan bagian orbital dari konfigurasi (tanpa jumlah keterisiannya) agar kita bisa memastikan bahwa orbital yang sama tidak muncul dua kali:

```python
s = '1p2.2s2.2p6.2s3.4d10'
orbitals = []

for occ_orbital in s.split('.'):
    orbital = occ_orbital[:2]
    n = int(orbital[0])
    ell_letter = orbital[1]
    ell = 'spdf'.index(ell_letter)
    n_electrons = int(occ_orbital[2:])

    # Periksa keberadaan orbital (l harus lebih kecil dari n)
    if ell >= n:
        print(f'No such orbital: {orbital}')

    # Periksa batas maksimum elektron
    if n_electrons > 2 * (2 * ell + 1):
        print(f'Too many electrons in orbital: {occ_orbital}')

    # Periksa duplikasi orbital
    if orbital in orbitals:
        print(f'Repeated orbital: {orbital}')

    orbitals.append(orbital)
```

Output:

```
No such orbital: 1p
Too many electrons in orbital: 2s3
Repeated orbital: 2s
```

Serial Python Minimalis:
Tutorial #05-Fungsi (Subprogram) dalam Python

Ahmad R. T. Nugraha

ver. 21 April 2026

Hingga saat ini, kode yang telah kita tulis belum banyak yang bisa digunakan ulang (*reusable*). Dalam pemrograman profesional, kita biasanya akan perlu mengelompokkan kode ke dalam blok-blok yang dapat diberi nama dan dieksekusi berulang kali (mungkin dengan parameter yang berbeda). Dengan kata lain, kita ingin menulis "subprogram" kita sendiri. Dalam bahasa Python, bentuk subprogram pada dasarnya adalah suatu "fungsi".

Penulisan fungsi tidak hanya berguna untuk meningkatkan portabilitas dan menghindari pengulangan baris kode, tetapi juga membantu memecah tugas-tugas kompleks menjadi tugas yang lebih sederhana. Dengan begitu, pada akhirnya kita dapat menghasilkan kode yang lebih mudah dipahami dan dikelola.

1 Pendefinisian Fungsi

Sebuah fungsi didefinisikan oleh ekspresi `def` yang memberinya nama dan menentukan "argumen" atau parameter apa saja yang diterimanya dalam tanda kurung, diikuti dengan titik dua (`:`). Badan fungsi diindentasi dengan 4 spasi. Saat dipanggil, segala pernyataan di dalam fungsi akan dieksekusi. Jika kata kunci `return` ditemui di titik mana pun, nilai yang ditentukan akan dikembalikan ke pemanggil (*caller*). Misalnya:

```
def cube(x):
    x_cubed = x**3
    return x_cubed

cube(3)
```

Output:

```
27
```

```
cube(4)
```

Output:

```
64
```

Fungsi di sini, `cube`, mengambil satu argumen, `x` (angka yang akan dipangkatkan tiga), dan mengembalikan satu nilai. Untuk mengembalikan dua nilai atau lebih dari sebuah fungsi, pisahkan dengan koma.

Program di bawah ini mendefinisikan sebuah fungsi untuk mengembalikan kedua akar dari persamaan kuadrat $ax^2 + bx + c$ (dengan asumsi persamaannya memiliki dua akar real). Parameter yang tepat yang diambil oleh fungsi ini adalah koefisien `a`, `b`, dan `c` yang mendefinisikan polinomial tersebut:

```
import math

def roots(a, b, c):
    d = b**2 - 4 * a * c
    r1 = (-b + math.sqrt(d)) / (2 * a)
    r2 = (-b - math.sqrt(d)) / (2 * a)
    return r1, r2

print(roots(1, 3, -10))
```

Output:

```
(2.0, -5.0)
```

2 Argumen *Keyword* dan *Default*

Sejauh ini, kita telah menggunakan argumen posisional dalam memanggil fungsi. Nilai yang akan ditetapkan ke setiap argumen fungsi ditentukan oleh urutan diberikannya nilai tersebut. Sebagai contoh, `roots(1, 3, -10)` menghasilkan pemanggilan fungsi `roots` dengan argumen `a=1`, `b=3`, dan `c=-10`. Kita juga dapat mengoper nilai ke sebuah fungsi dalam urutan sembarang menggunakan argumen *keyword*:

```
roots(b=3, c=-10, a=1)
```

Output:

```
(2.0, -5.0)
```

Argumen posisional dan *keyword* dapat digunakan dalam pemanggilan fungsi yang sama, tetapi argumen posisional harus didahulukan:

```
roots(1, c=-10, b=3)
```

Output:

```
(2.0, -5.0)
```

```
roots(1, b=3, -10)
```

Output:

```
  File "<ipython-input-10-b31133d01219>", line 1
    roots(1, b=3, -10)
                 ^
SyntaxError: positional argument follows keyword argument
```

Terkadang ada gunanya membuat argumen ke sebuah fungsi menjadi opsional. Dengan kata lain, kode fungsi akan menggunakan nilai bawaan (*default*) untuk argumen tersebut jika tidak ada nilai yang diberikan saat fungsi dipanggil. Hal ini dapat dilakukan di dalam definisi fungsi:

```
def Vm_ideal(T=298.15, p=101325):
    R = 8.31446 # konstanta gas, J.K-1.mol-1
    return R * T / p
```

Sekarang, fungsi `Vm_ideal` dapat dipanggil tanpa argumen sama sekali, dengan salah satu dari `T` atau `p`, atau dengan keduanya:

```
# Tanpa argumen: gunakan nilai default
Vm_ideal()
```

Output:

```
0.024465395993091537
```

```
# Gunakan T yang diberikan dan p default
Vm_ideal(273.15)
```

Output:

```
0.022413962487046633
```

```
# Gunakan T default dan p yang diberikan
Vm_ideal(p=1.e5)
```

Output:

```
0.02478956249
```

```
# Gunakan nilai yang diberikan untuk p dan T
Vm_ideal(600, 1.e3)
```

Output:

```
4.988676000000001
```

Kadang kita perlu juga mengoper isi dari sebuah `list` ke fungsi sebagai argumen-argumennya (daripada meneruskan `list` tersebut sebagai satu objek tunggal). "Sintaksis bintang" (*star syntax*) membongkar (*unpack*) suatu barisan dengan cara ini:

```
state = [298.15, 101325]
# Membongkar state menjadi argumen T, p
Vm_ideal(*state)
```

Output:

```
0.024465395993091537
```

3 Docstring

Docstring adalah literal string semacam komentar yang ditempatkan setelah *signature* (baris definisi) fungsi yang menjelaskan apa yang dilakukan fungsi tersebut, nilai apa yang dikembalikannya, dan argumen apa saja yang diterimanya. Jika string tersebut ditulis dengan tanda kutip tiga (`"""..."""` atau `'''...'''`), maka kita dapat memasukkan baris baru (*line breaks*).

```
def Vm_ideal(T=298.15, p=101325):
    """Return the molar volume of an ideal gas, in m^3.
    The temperature, T (in K) and pressure, p (in Pa) can be provided.
    """
    R = 8.31446 # gas constant, J.K-1.mol-1
    return R * T / p
```

Untuk memeriksa *docstring* sebuah fungsi dalam sesi Python interaktif (misalnya, IPython atau Jupyter notebook), ketik `help(func)` atau `func?`:

```
help(Vm_ideal)
```

Output:

```
Signature: Vm_ideal(T=298.15, p=101325)
Docstring:
Return the molar volume of an ideal gas, in m^3.
The temperature, T (in K) and pressure, p (in Pa) can be provided.
File:      <ipython-input...>
Type:      function
```

Teknik ini juga berlaku untuk fungsi bawaan dan fungsi yang diimpor:

```
help(math.asin)
```

Output:

```
Signature: math.asin(x, /)
Docstring:
Return the arc sine (measured in radians) of x.

The result is between -pi/2 and pi/2.
Type:      builtin_function_or_method
```

4 Cakupan (*Scope*)

Istilah cakupan (*scope*) mengacu pada cara nama variabel diselesaikan (*resolved*) dalam program Python, yaitu bagaimana interpreter Python mengetahui objek mana yang dirujuk oleh variabel tersebut. Pemahaman tentang cakupan sangat penting dalam pemrograman karena dimungkinkan untuk menggunakan nama variabel yang sama untuk merujuk pada objek yang berbeda di dalam dan di luar sebuah fungsi. Misalnya:

```
x, y = 1, 3

def add_y(z):
    x = y + z
    return x

add_y(x)
```

Output:

```
4
```

```
x
```

Output:

```
1
```

Minimalnya kita perlu memahami tiga jenis cakupan: (1) cakupan lokal (*local scope*), (2) cakupan global (*global scope*), dan (3) cakupan bawaan (*built-in scope*). Definisi `x, y = 1, 3` di atas membuat dua nama variabel dalam cakupan global, yakni sesuai namanya, variabel tersebut dapat diakses dari mana saja di dalam kode kita.

Definisi fungsi `add_y` mengambil satu argumen tunggal, yang diikat ke variabel baru bernama `z`. Badan fungsi memiliki akses ke objek ini serta objek `x` dan `y`. Namun, ketika penugasan `x = y + z` dibuat, bukan `x` global yang diperbarui, melainkan nama variabel baru yang juga disebut `x` yang telah dibuat dan bersifat lokal terhadap fungsi tersebut. Perubahan pada `x` di dalam fungsi ini tidak memengaruhi `x` global. Oleh karena itu, nilai yang dikembalikan dari fungsi adalah `4`, tetapi pemeriksaan nilai `x` setelah fungsi

mengembalikan nilainya akan memberikan nilai dari x global (yang tidak berubah) yaitu 1.

Ada banyak hal yang terjadi di sini sehingga perlu dijelaskan sedikit lebih mendetail. Di dalam fungsi add_y, terdapat pernyataan penetapan atau penugasan, x = y + z. Penugasan dalam Python selalu dievaluasi dari kanan ke kiri (interpreter perlu mengetahui objek apa yang akan ditugaskan sebelum menyematkan nama variabel kepadanya). Oleh karena itu, ekspresi y + z harus dievaluasi, dan nilai y harus diperoleh (z dilewatkan sebagai argumen ke fungsi).

Aturan cakupan menentukan bagaimana nilai y diambil. Pertama, interpreter Python mencari di cakupan lokal fungsi, tetapi karena tidak ada variabel bernama y yang ada di cakupan ini, interpreter tidak menemukan apa pun. Selanjutnya, interpreter mencari di cakupan global kode (atau sesi Python interaktif) dan menemukan variabel bernama y yang tersemat pada bilangan bulat 3. Jadi, sisi kanan dari pernyataan tersebut dievaluasi menjadi bilangan bulat 4, dan Python diinstruksikan untuk menetapkan objek ini ke nama variabel x. Karena ini terjadi di dalam fungsi, nama variabel baru x dibuat dalam cakupan yang bersifat lokal terhadap fungsi tersebut. x inilah yang dirujuk dalam pernyataan return, dan karenanya objek 4 dikembalikan. Di luar fungsi, satu-satunya x adalah yang berada dalam cakupan global, dan itu tidak berubah.

Kita mungkin saja ingin benar-benar menaikkan (*increment*) bilangan bulat x dalam cakupan global, yang dengannya kita dapat menugaskan kembali objek yang dikembalikan oleh add_y ke x:

```
x = add_y(x)
x
```

Output:

```
4
```

Fungsi bukanlah satu-satunya tempat keberadaan cakupan lokal. Misalnya, *list comprehension* juga menggunakan cakupan lokal:

```
x = 1
[x**2 for x in range(6)]
```

Output:

```
[0, 1, 4, 9, 16, 25]
```

```
x
```

Output:

```
1
```

Cakupan bawaan (*built-in scope*) adalah tempat Python meletakkan kata kunci, fungsi bawaan, *exceptions*, dan atribut lain yang menyusun sintaksis dan fungsionalitas intinya.

Cakupan bawaaan adalah tempat terakhir yang dicari oleh *interpreter* saat mencoba menyelesaikan (*resolve*) sebuah nama variabel. Misalnya, jika kita (secara tidak bijak) mengikat nama `len` ke sebuah objek, nama ini tidak akan lagi merujuk pada fungsi bawaan `len()` (untuk mencari panjang suatu urutan):

```
len('abc')
```

Output:

```
3
```

```
len = 100 # diperbolehkan; tapi bukan ide yang bagus.
len('abcdef')
```

Output:

```
TypeError                                 Traceback (most recent call last)
<ipython-input-3-4f6544af4f86> in <module>
----> 1 len('abcdef')

TypeError: 'int' object is not callable
```

Pada penggunaan pertamanya, nama `len` hanya ditemukan di cakupan bawaan sebagai fungsi panjang (*length function*), dan begitulah cara penggunaannya. Setelah `len` ditetapkan ke bilangan bulat `100`, pendefinisian ini eksis dalam cakupan global dan ditemukan di sana sebelum dicari di cakupan bawaan, sehingga penggunaannya sebagai sebuah fungsi memunculkan *exception*, yakni `len('abcdef')` mencoba memanggil objek bilangan bulat `100` dengan sebuah argumen, `'abcdef'`, yang merupakan suatu kesalahan (*error*).

5 Fungsi `lambda`

Ada kalanya berguna juga bagi kita untuk mendefinisikan dan menggunakan fungsi sederhana, tetapi tanpa membungkusnya dalam blok `def` atau memberinya nama. Pada kasus ini, fungsi `lambda` dapat memainkan peranan penting. Sintaksis untuk mendefinisikan fungsi `lambda` adalah:

```
lambda arguments: expression
```

dengan `arguments` adalah urutan argumen yang dipisahkan koma, seperti fungsi biasa, dan `expression` adalah ekspresi Python sederhana (tidak diperbolehkan adanya blok perulangan atau pengondisian). Misalnya, dua hal berikut ini ekuivalen:

```
def adder(a, b):
    return a + b
```

```
adder(5, 6)
```

Output:

```
11
```

```
adder = lambda a, b: a + b
adder(5, 6)
```

Output:

```
11
```

Pada praktiknya, fungsi `lambda` bisa dibilang agak terlalu sering disalahgunakan, dan biasanya jika fungsi `lambda` diikat ke sebuah nama variabel seperti ini (yang tidak lagi menjadi anonim), fungsi `def` reguler lebih disarankan. Namun, fungsi `lambda` sangat berguna dalam kasus-kasus sekali pakai (*one-off cases*) ketika objek fungsi itu sendiri diteruskan ke fungsi lain. Contoh yang umum adalah dalam mengurutkan sebuah `list` dari `tuple`. Tinjau `list` berikut yang berisi tiga `tuple`, masing-masing menyimpan simbol unsur dan nomor massanya:

```
masses = [('H', 1), ('Ar', 40), ('C', 12)]
```

Pengurutan `list` ini dengan metode `sort` akan menghasilkan `list tuple` yang terurut berdasarkan urutan abjad dari item pertama masing-masing:

```
masses.sort()
masses
```

Output:

```
[('Ar', 40), ('C', 12), ('H', 1)]
```

Misalkan kita ingin mengurutkan `list` tersebut berdasarkan item kedua dari setiap `tuple`, yaitu nomor massanya. Metode `sort` menerima argumen opsional, `key`, yang mengharapkan untuk menerima sebuah fungsi. Fungsi ini diteruskan ke setiap item di dalam `list`, dan `list` tersebut diurutkan berdasarkan apa pun yang dikembalikan oleh fungsi itu untuk setiap item. Kita tinjau contoh berikut ini.

```
def get_mass(e):
    return e[1]

masses.sort(key=get_mass)
masses
```

Output:

```
[('H', 1), ('C', 12), ('Ar', 40)]
```

Perhatikan sintaksisnya: `key` harus berupa sebuah fungsi, argumen tunggalnya adalah sebuah item dari `list` (di sini, dipanggil `e`, tetapi nama apa pun boleh). Fungsi bernama `get_mass` ini memiliki tugas sederhana untuk mengembalikan item kedua dari `tuple` yang diteruskan kepadanya. Namun, karena ini mungkin satu-satunya tempat fungsi semacam itu digunakan, lebih mudah untuk menggunakan fungsi `lambda` secara sejajar (*in-line*):

```
masses = [('H', 1), ('Ar', 40), ('C', 12)]
masses.sort(key=lambda e: e[1])
masses
```

Output:

```
[('H', 1), ('C', 12), ('Ar', 40)]
```

Demikian pula, pengurutan string dengan mengabaikan spasi di awal (*leading whitespace*) dapat dicapai dengan menggunakan `strip()` di dalam fungsi `lambda`:

```
sorted([' alpha', 'beta', ' gamma'])
```

Output:

```
[' gamma', ' alpha', 'beta']
# spasi diikutkan dalam pengurutan
```

```
# Mengurutkan berdasarkan string tanpa spasi di awal.
sorted([' alpha', 'beta', ' gamma'], key=lambda s: s.strip())
```

Output:

```
['alpha', 'beta', ' gamma']
```

Sebenarnya, dalam kasus ini, kita tidak memerlukan fungsi `lambda` karena metode string `strip` tersedia langsung dari objek tipe `str`:

```
sorted([' alpha', 'beta', ' gamma'], key=str.strip)
```

Output:

```
['alpha', 'beta', ' gamma']
```

Studi Kasus

Kasus 1 Kapasitas Panas Einstein

Model Einstein untuk kapasitas panas dari sebuah kristal adalah

$$C_{V,m} = 3R\left(\frac{\Theta_E}{T}\right)^2 \frac{\exp\left(\frac{\Theta_E}{T}\right)}{\left[\exp\left(\frac{\Theta_E}{T}\right) - 1\right]^2}$$

dengan suhu Einstein, Θ_E, adalah konstanta untuk material tertentu.
Tuliskan sebuah fungsi Python untuk menghitung $C_{V,m}$ pada 300 K untuk (a) natrium ($\Theta_E = 192$ K) dan (b) intan ($\Theta_E = 1450$ K).

Solusi

Fungsi `CV_Einstein` mengambil dua argumen: `T`, suhu tempat menghitung kapasitas panas; dan `ThetaE`, suhu karakteristik Einstein untuk material tersebut.

```
import numpy as np

def CV_Einstein(T, ThetaE):
    """Return the Einstein molar heat capacity of a material, C_Vm.
    T is the temperature (K) and ThetaE the Einstein temperature for
    ↪  the
    material being considered. C_Vm is returned in J.K-1.mol-1.
    """
    # Konstanta gas, J.K-1.mol-1
    R = 8.314462618
    x = ThetaE / T
    f = np.exp(x)
    return 3 * R * f * (x / (f - 1))**2

T = 300
ThetaE_Na, ThetaE_dia = 192, 1450

CV_Na = CV_Einstein(T, ThetaE_Na)
CV_dia = CV_Einstein(T, ThetaE_dia)

print(f'Material CVm({T} K)')
print('{:7s} {:8.2f} J.K-1.mol-1'.format('Na', CV_Na))
print('{:7s} {:8.2f} J.K-1.mol-1'.format('diamond', CV_dia))
```

Output:

```
Material CVm(300 K)
Na          24.11 J.K-1.mol-1
diamond      4.71 J.K-1.mol-1
```

Kasus 2 Persamaan Keadaan Gas

Untuk persamaan-persamaan keadaan berikut, definisikan fungsi Python yang memberikan tekanan gas, p, jika diberikan volume V, jumlah mol n, dan suhu T.
Persamaan keadaan gas ideal:

$$p = \frac{nRT}{V}$$

Persamaan keadaan van der Waals:

$$p = \frac{nRT}{V - nb} - \frac{n^2 a}{V^2}$$

Persamaan keadaan Dieterici:

$$p = \frac{nRT \exp\left(-\frac{na}{RTV}\right)}{V - nb}$$

Bandingkan tekanan yang diprediksi untuk 1 mol CO_2 pada $T = 273.15\,\mathrm{K}$ yang dibatasi dalam volume 20 L. Ambil parameter van der Waals $a = 3.640\,\mathrm{L^2\,bar\,mol^{-2}}$ dan $b = 0.04267\,\mathrm{L\,mol^{-1}}$; dan parameter Dieterici $a = 4.692\,\mathrm{L^2\,bar\,mol^{-2}}$ dan $b = 0.04639\,\mathrm{L\,mol^{-1}}$.
Ulangi perhitungan tersebut untuk jumlah 0.01 mol.

Solusi

Pertama, definisikan fungsi Python untuk masing-masing persamaan keadaan.

```python
import numpy as np

# Konstanta gas, J.K-1.mol-1
R = 8.314462618

def p_ideal(n, T, V):
    """Ideal gas equation of state, SI units."""
    return n * R * T / V

def p_vdw(n, T, V, a, b):
    """Van der Waals equation of state, SI units."""
    return n * R * T / (V - n * b) - n**2 * a / V**2

def p_dieterici(n, T, V, a, b):
    """Dieterici equation of state, SI units."""
    return n * R * T * np.exp(-n * a / (R * T * V)) / (V - n * b)
```

Sekarang panggil fungsi-fungsi tersebut dengan variabel keadaan yang sesuai, `n`, `T`, dan `V` serta parameter `a` dan `b`. Berhati-hatilah untuk mengonversi semuanya ke dalam satuan SI.

```python
# Jumlah gas (mol), suhu (K), dan volume (m3).
n, T, V = 1, 273.15, 20 * 1.e-3

print(f'n = {n} mol, T = {T} K, V = {V} m3')

# Persamaan keadaan gas ideal.
print(f'Ideal:         p = {p_ideal(n, T, V):.3f} Pa')

# Persamaan keadaan Van der Waals (a dikonversi ke m6.Pa.mol-2, b ke
↪ m3.mol-1).
a_vdw, b_vdw = 3.640 / 10, 0.04267 / 1.e3
print(f'Van der Waals: p = {p_vdw(n, T, V, a_vdw, b_vdw):.3f} Pa')

# Persamaan keadaan Dieterici (a dikonversi ke m6.Pa.mol-2, b ke
↪ m3.mol-1).
a_diet, b_diet = 4.692 / 10, 0.04639 / 1.e3
print(f'Dieterici:     p = {p_dieterici(n, T, V, a_diet, b_diet):.3f}
↪ Pa')
```

Output:

```
n = 1 mol, T = 273.15 K, V = 0.02 m3
Ideal:         p = 113554.773 Pa
Van der Waals: p = 112887.560 Pa
Dieterici:     p = 112649.100 Pa
```

Sekarang ulangi perhitungan tersebut untuk 0.01 mol:

```
n = 0.01
print(f'n = {n} mol, T = {T} K, V = {V} m3')
print(f'Ideal:          p = {p_ideal(n, T, V):.3f} Pa')
print(f'Van der Waals: p = {p_vdw(n, T, V, a_vdw, b_vdw):.3f} Pa')
print(f'Dieterici:      p = {p_dieterici(n, T, V, a_diet, b_diet):.3f}
 ↪  Pa')
```

Output:

```
n = 0.01 mol, T = 273.15 K, V = 0.02 m3
Ideal:          p = 1135.548 Pa
Van der Waals: p = 1135.481 Pa
Dieterici:      p = 1135.457 Pa
```

Pada tekanan yang lebih rendah (dicapai dengan membatasi jumlah gas yang lebih sedikit pada volume dan suhu yang sama dengan sebelumnya), ketiga persamaan keadaan tersebut memberikan nilai-nilai yang hampir sama.

Serial Python Minimalis: Tutorial #06-Struktur Data Dasar

Ahmad R. T. Nugraha

ver. 21 April 2026

Selain tipe data dasar seperti `int`, `float`, `bool`, `str`, dan sebagainya, Python menyediakan berbagai tipe data majemuk (*compound data types*), yaitu tipe data yang digunakan untuk mengelompokkan objek-objek. Struktur `list`, yang telah diperkenalkan sebelumnya, merupakan contoh paling sederhana dari tipe data majemuk. Struktur `list` akan dibahas lebih jauh pada sesi kali ini bersama dengan beberapa struktur data lain yang sangat berguna untuk menyimpan objek. Kita akan bahas juga perbedaan penting antara objek yang dapat diubah (*mutable*) dan tidak dapat diubah (*immutable*).

1 List

Kita sudah pernah mengeksplorasi penggunaan dasar `list` dalam Python, seperti proses kreasinya, pengindeksan, pemotongan (*slicing*), penambahan (*appending*), dan perluasan (*extending*). Secara umum, `list` adalah urutan objek dari tipe apa pun yang terurut dan bersifat dapat diubah (*mutable*). Struktur `list` dikatakan terurut karena setiap elemennya dapat dirujuk menggunakan indeks bilangan bulat yang unik dan memiliki panjang yang dapat diketahui menggunakan fungsi bawaan `len`:

```
# Sebuah list dengan elemen yang diindeks pada lst[0], lst[1], lst[2],
↪  lst[3].
lst = ['A', 2.0, 'C', 9]
lst[2] # elemen ketiga dari list
```

Output:

```
'C'
```

```
len(lst)
```

Output:

```
4
```

`list` bersifat *mutable* yang berarti elemennya dapat diubah langsung di tempat (berbeda dengan string):

```
lst[2] = -1
lst
```

Output:

```
['A', 2.0, -1, 9]
```

Perubahan ini tidak menjadi masalah karena `lst[2]` sebelumnya merupakan referensi ke objek string `'C'`. Kemudian, kita cukup menetapkannya kembali ke objek baru, yaitu bilangan bulat `-1`. Namun, string itu sendiri bersifat tidak dapat diubah *immutable*:

```
my_str = "hello"
my_str
```

Output:

```
'hello'
```

```
my_str[1] = "a"
```

Output:

```
TypeError                                 Traceback (most recent call last)
Input In [x], in <module>
----> 1 my_str[1] = "a"

TypeError: 'str' object does not support item assignment
```

Dalam situasi di atas, satu-satunya solusi adalah mendefinisikan objek `str` yang sepenuhnya baru, contohnya:

```
my_str = my_str[0] + "a" + my_str[2:]
my_str
```

Output:

```
'hallo'
```

Meskipun objek *immutable* tampak kurang fleksibel, ada keunggulan dari sisi kecepatan dan keamanan. Objek ini juga dapat diteruskan ke berbagai fungsi maupun objek lain tanpa perlu disalin, karena isinya dijamin tidak akan berubah di masa mendatang.

Fakta bahwa objek *mutable* dapat berubah setelah dibuat akan membawa konsekuensi penting ketika objek yang sama ditetapkan pada lebih dari satu nama variabel. Contoh berikut akan memperjelas situasinya.

```
lst1 = ['a', 'b', 'c']
lst2 = lst1
```

```
lst1.append('d') # Menambahkan elemen ke lst1

lst1 # Sesuai perkiraan
```

Output:

```
['a', 'b', 'c', 'd']
```

```
lst2 # lst2 ternyata ikut berubah!
```

Output:

```
['a', 'b', 'c', 'd']
```

Apa yang sebenarnya terjadi di sini? Untuk memahaminya, cara terbaik adalah membayangkan nama variabel (seperti `lst1`) sebagai sebuah label yang menempel pada objek `list` yang mendasarinya, bukan sebagai objek itu sendiri. Pada kode di atas, objek list `['a', 'b', 'c']` dibuat di suatu tempat di dalam memori komputer. Kemudian, nama variabel `lst1` ditugaskan untuk merujuk pada objek tersebut. Penetapan `lst2 = lst1` membuat nama `lst2` menunjuk ke objek yang persis sama. Dengan kata lain, `lst1` dan `lst2` adalah referensi ke satu `list` yang sama. Pengaksesan `list` tersebut melalui salah satu nama variabel akan mengubah isinya.

Objek *immutable* seperti string dan angka tentu saja dapat ditugaskan ke banyak nama variabel, tetapi karena nilainya sama sekali tidak dapat diubah, tidak akan ada perilaku tak terduga yang terjadi. Variabel apa pun dapat ditugaskan ke objek yang baru; perhatikan contoh berikut:

```
n = 1
m = n
m = m + 1

n
```

Output:

```
1
```

```
m
```

Output:

```
2
```

Selama operasi ini berlangsung, `n` tetap menjadi referensi ke objek *immutable* `1`. Variabel `m` awalnya juga merujuk ke objek ini, tetapi penetapan `m = m + 1` menghasilkan objek baru, yaitu 2, dan menugaskan kembali nama `m` kepadanya (dan sama sekali tidak mengubah angka `1` yang dirujuk oleh `n`). Python memiliki metode bawaan, `id()`, untuk

melacaknya. Biasanya, fungsi ini akan mengembalikan alamat objek di dalam memori komputer:

```
n = 1
id(n)
```

Output:

```
4400245040
```

```
m = n
id(m) # Objek yang sama, id yang sama
```

Output:

```
4400245040
```

```
m = m + 1
id(m) # Sekarang m adalah objek int baru dengan id yang berbeda
```

Output:

```
4400245072
```

```
id(n) # n tidak berubah
```

Output:

```
4400245040
```

Jika kita memang menginginkan salinan `list` yang saling bebas (*independent*), panggil metode `copy()`:

```
lst1 = ['a', 'b', 'c']
lst2 = lst1.copy() # alternatif lain, potong seutuhnya: lst1[:]

lst1.append('d')
lst1
```

Output:

```
['a', 'b', 'c', 'd']
```

```
lst2 # lst2 tidak berubah
```

Output:

```
['a', 'b', 'c']
```

```
id(lst1), id(lst2) # karena ia adalah objek yang berbeda dari lst1
```

Output:

```
(4456958976, 4445977280)
```

Sementara itu, untuk mendapatkan indeks suatu elemen di dalam `list`, panggil metode `index`:

```
lst2.index('c')
```

Output:

```
2
```

Struktur `list` dapat diurutkan langsung di tempat (*in-place*) menggunakan metode `sort` (yang tidak mengembalikan nilai apa pun):

```
masses = [44, 28, 16, 32, 4]
masses.sort()
masses
```

Output:

```
[4, 16, 28, 32, 44]
```

Sebagai alternatif, fungsi bawaan `sorted` akan mengembalikan objek `list` baru dengan urutan yang sudah disesuaikan, sementara `list` aslinya tetap tidak berubah:

```
masses = [44, 28, 16, 32, 4]
sorted(masses) # mengembalikan list terurut yang baru
```

Output:

```
[4, 16, 28, 32, 44]
```

```
masses # tidak berubah
```

Output:

```
[44, 28, 16, 32, 4]
```

Python akan menolak untuk mengurutkan sebuah `list` apabila elemen-elemennya tidak dapat dibandingkan nilainya secara logis:

```
lst1 = [1, 0, "a"]
lst1.sort()
```

Output:

```
TypeError                                  Traceback (most recent call last)
Input In [x], in <module>
----> 1 lst1.sort()
```

```
TypeError: '<' not supported between instances of 'str' and 'int'
```

Karena tidak ada cara yang masuk akal untuk membandingkan nilai antara objek string dan bilangan bulat, Python memunculkan pengecualian `TypeError`.

2 Tuple

Struktur `tuple` adalah urutan objek terurut yang bersifat *immutable*. Seperti halnya `list`, `tuple` dapat menampung objek dari berbagai tipe. Struktur `tuple` didefinisikan dengan memberikan urutan elemen di dalam tanda kurung biasa atau dengan meneruskan objek yang sesuai (yaitu yang bersifat *iterable*) ke dalam konstruktor `tuple`:

```
tpl1 = (0, 'a', 2.3)
tpl1
```

Output:

```
(0, 'a', 2.3)
```

```
tpl2 = tuple('abcdef')
tpl2
```

Output:

```
('a', 'b', 'c', 'd', 'e', 'f')
```

Pada dasarnya, dalam banyak kasus, tanda kurung biasa tidak diperlukan dan urutan yang dipisahkan oleh koma sudah cukup:

```
tpl1 = 0, 'a', 2.3
tpl1
```

Output:

```
(0, 'a', 2.3)
```

Untuk membuat `tuple` dengan elemen tunggal, tanda koma di akhir sangat diperlukan:

```
a = (3)
a # ini adalah bilangan bulat 3
```

Output:

```
3
```

```
a = (3,)
a # sebuah tuple dengan panjang 1 (sebuah "singleton")
```

Output:

```
(3,)
```

Sebagai struktur data yang *immutable*, penetapan nilai baru ke dalam elemen `tuple` tidak diperbolehkan, begitu pula penambahan maupun penghapusan elemen tidak didukung oleh struktur ini:

```
tpl2[0] = "z"
```

Output:

```
TypeError                                 Traceback (most recent call last)
Input In [x], in <module>
----> 1 tpl2[0] = "z"

TypeError: 'tuple' object does not support item assignment
```

3 Himpunan (`set`)

Himpunan atau `set` dalam Python adalah sekumpulan objek tak terurut (*unordered*) yang bersifat *mutable* dan tidak memiliki elemen duplikat:

```
s = set([1, 1, 'a', 'b', 1, 2, 'a', 1.0])
s
```

Output:

```
{1, 2, 'a', 'b'}
```

Perhatikan bahwa `1` dan `1.0` dianggap sebagai duplikat meskipun keduanya merupakan objek dengan tipe yang berbeda (yakni `int` dan `float`) karena secara matematis nilainya setara. Sebuah `set` yang tidak kosong juga dapat didefinisikan dengan meletakkan elemen-elemennya secara langsung di dalam tanda kurung kurawal:

```
s = {1, 1, 'a', 'b', 1, 2, 'a', 1.0}
```

Sebagai catatan, `s = {}` mendefinisikan sebuah *dictionary* (akan kita bahas belakangan), bukan sebuah `set` kosong. Karena tidak memiliki urutan yang baku, `set` tidak dapat diakses melalui indeks:

```
s[0]
```

Output:

```
TypeError                                 Traceback (most recent call last)
Input In [x], in <module>
----> 1 s[0]
```

```
TypeError: 'set' object is not subscriptable
```

Sebagai objek yang bersifat *mutable*, `set` memiliki berbagai metode untuk menambah dan menghapus elemen:

```
s.add(3) # menambah satu elemen
s
```

Output:

```
{1, 2, 3, 'a', 'b'}
```

```
s.update([2, 3, 4]) # menambah banyak elemen sekaligus dari sebuah koleksi
s
```

Output:

```
{1, 2, 3, 4, 'a', 'b'}
```

```
s.remove(4) # menghapus elemen tertentu
s
```

Output:

```
{1, 2, 3, 'a', 'b'}
```

```
s.pop() # menghapus dan mengembalikan sembarang elemen
```

Output:

```
1
```

```
s
```

Output:

```
{2, 3, 'a', 'b'}
```

Struktur `set` juga mendukung penggunaan operator matematis berikut: `-` (selisih / *difference*), `^` (selisih simetris / *symmetric difference*), `&` (irisan / *intersection*), dan `|` (gabungan / *union*):

```
s1 = {"A", "B", "C", "D"}
s2 = {"C", "D", "X", "Y", "Z"}

s1 - s2 # elemen yang ada di s1 tetapi tidak ada di s2
```

Output:

```
{'A', 'B'}
```

```
s2 - s1 # elemen yang ada di s2 tetapi tidak ada di s1
```

Output:

```
{'X', 'Y', 'Z'}
```

```
s1 ^ s2 # selisih simetris: elemen yang tidak dimiliki bersama oleh kedua set
```

Output:

```
{'A', 'B', 'X', 'Y', 'Z'}
```

```
s1 & s2 # irisan: elemen yang sama-sama dimiliki oleh kedua set
```

Output:

```
{'C', 'D'}
```

```
s1 | s2 # gabungan: seluruh elemen unik dari kedua set
```

Output:

```
{'A', 'B', 'C', 'D', 'X', 'Y', 'Z'}
```

4 Kamus (*dictionary*)

Kamus atau kita sebut *dictionary* dalam Python merupakan salah satu jenis *associative array*, yaitu sebuah struktur data yang memetakan objek kunci (*key*) ke objek nilai (*value*). Alih-alih menggunakan indeks berupa bilangan bulat, setiap nilai diasosiasikan dengan kunci unik (kuncinya wajib berupa objek yang *immutable*).

Cara paling sederhana untuk mendefinisikan *dictionary* adalah dengan menempatkan pasangan kunci-nilai (*key-value pairs*) sebagai urutan yang dipisahkan koma di dalam tanda kurung kurawal:

```
densities = {'Fe': 7.9, 'Al': 2.73, 'Pt': 21.4}
densities['Pt'] # dalam satuan g/cm3
```

Output:

```
21.4
```

Percobaan pengambilan nilai menggunakan kunci yang tidak ada di dalam *dictionary* akan memicu pengecualian `KeyError`:

```
densities["Ni"]
```

Output:

```
KeyError                                  Traceback (most recent call last)
Input In [x], in <module>
----> 1 densities["Ni"]

KeyError: 'Ni'
```

Metode get memungkinkan kita untuk memberikan nilai bawaan (*default*) untuk berjaga-jaga apabila kunci yang dicari ternyata tidak terdapat di dalam *dictionary*:

```
print(densities.get("Ni")) # mengembalikan None jika Ni tidak ada
```

Output:

```
None
```

```
print(densities.get("Ni", "No data available"))
```

Output:

```
No data available
```

Dictionary bersifat *mutable*, sehingga nilai yang sudah ada dapat diubah dan pasangan kunci-nilai baru dapat disisipkan sewaktu-waktu:

```
densities['Fe'] = 7.85
densities['Cu'] = 8.94
densities.update({'Hg': 13.6, 'Au': 19.3})

densities
```

Output:

```
{'Fe': 7.85, 'Al': 2.73, 'Pt': 21.4, 'Cu': 8.94, 'Hg': 13.6, 'Au': 19.3}
```

Iterasi suatu *dictionary* akan mengembalikan kunci-kuncinya secara berurutan sesuai dengan waktu saat kunci tersebut dimasukkan:

```
for element in densities:
    print(element)
```

Output:

```
Fe
Al
Pt
Cu
Hg
Au
```

Kunci-kunci *dictionary* juga dapat diekstrak secara lebih eksplisit melalui pemanggilan `densities.keys()`. Sementara itu, untuk mengakses nilainya saja, gunakan `densities.values()`:

```
for density in densities.values():
    print(density)
```

Output:

```
7.85
2.73
21.4
8.94
13.6
19.3
```

Sering kali kita ingin mengiterasi kunci sekaligus nilainya bersamaan, yang dapat diakomodasi oleh metode `densities.items()`:

```
for element, density in densities.items():
    print(element, density)
```

Output:

```
Fe 7.85
Al 2.73
Pt 21.4
Cu 8.94
Hg 13.6
Au 19.3
```

Terakhir, perlu diperhatikan bahwa sebuah *dictionary* kosong dapat dibuat cukup dengan sepasang tanda kurung kurawal tanpa isi:

```
my_dict = {}
```

Sekali lagi, ingat bahwa struktur ini bukanlah representasi dari `set` kosong.

Studi Kasus

Kasus 1 *Dictionary* sebagai Basis Data Sederhana

Salah satu penerapan *dictionary* adalah sebagai basis data sederhana. Nilai-nilai dapat disimpan berpasangan dengan kunci (*keys*) daripada ditugaskan ke nama variabel secara individu. Cara ini jauh lebih mudah untuk mengelola data dengan keuntungan kuncinya dapat berupa string sembarang (atau tipe data *immutable* lainnya), sedangkan nama variabel dibatasi oleh aturan sintaksis Python.
Tunjukkan bagaimana cara menggunakan *dictionary* bersarang (*nested dictionary*) untuk menyimpan, mengakses, memperbarui, dan menggabungkan data sifat-sifat fisik dari beberapa unsur Golongan IV (seperti C, Si, Ge, Sn, dan Pb).

Solusi

Untuk menyimpan beberapa sifat dari unsur C, Si, dan Ge, kita dapat menyusun sebuah *dictionary* bersarang dengan cara berikut:

```
element_properties = {
    'C': {'mass /u': 12.0, 'Tm /K': 3823, 'Tb /K': 5100,
          'rho /g.cm-3': 3.51, 'IE /eV': 11.26, 'atomic radius /pm':
          ↪ 77.2},
    'Si': {'mass /u': 28.1, 'Tm /K': 1683, 'Tb /K': 2628,
           'rho /g.cm-3': 2.33, 'IE /eV': 8.15, 'atomic radius /pm':
           ↪ 117},
    'Ge': {'mass /u': 72.6, 'Tm /K': 1211, 'Tb /K': 3103,
           'rho /g.cm-3': 5.32, 'IE /eV': 7.90, 'atomic radius /pm':
           ↪ 122.5}
}

element_properties['C']
```

Output:

```
{'mass /u': 12.0, 'Tm /K': 3823, 'Tb /K': 5100, 'rho /g.cm-3': 3.51,
 ↪ 'IE /eV': 11.26, 'atomic radius /pm': 77.2}
```

Kita dapat mengakses data spesifik dengan memanggil kuncinya secara beruntun:

```
IE = element_properties['Ge']['IE /eV']
print(f'The ionization energy of Ge is {IE} eV.')
```

Output:

```
The ionization energy of Ge is 7.9 eV.
```

Kita juga bisa mengiterasi *dictionary* tersebut untuk mencetak massa jenis dari seluruh unsur yang tersimpan:

```
print('Densities of Group IV elements (g.cm-3)')
for symbol, properties in element_properties.items():
    density = properties['rho /g.cm-3']
    print(f'{symbol:>2s}: {density:5.2f}')
```

Output:

```
Densities of Group IV elements (g.cm-3)
 C:  3.51
Si:  2.33
Ge:  5.32
```

Sebagai struktur data yang *mutable*, *dictionary* dapat diperbarui. Misalnya, kita dapat menambahkan entri baru yang bersesuaian dengan titik leleh dalam derajat Celcius untuk setiap unsur:

```
for symbol, properties in element_properties.items():
    Tm_K = properties['Tm /K']
    element_properties[symbol]['Tm /degC'] = Tm_K - 273

print(f"Melting point of silicon: {element_properties['Si']['Tm
↪ /degC']} degC")
```

Output:

```
Melting point of silicon: 1410 degC
```

Sejak versi Python 3.9, dua buah *dictionary* dapat digabungkan secara langsung menggunakan operator `|`:

```
more_element_properties = {
    'Sn': {'mass /u': 118.7, 'Tm /K': 505, 'Tb /K': 2543,
           'rho /g.cm-3': 7.29, 'IE /eV': 7.34, 'atomic radius /pm':
            ↪ 140.5},
    'Pb': {'mass /u': 207.2, 'Tm /K': 601, 'Tb /K': 2013,
           'rho /g.cm-3': 11.3, 'IE /eV': 7.42, 'atomic radius /pm':
            ↪ 175}
}

# Tambahkan sifat unsur Sn dan Pb ke dalam dictionary awal.
element_properties = element_properties | more_element_properties
print(element_properties.keys())
```

Output:

```
dict_keys(['C', 'Si', 'Ge', 'Sn', 'Pb'])
```

Sebagai catatan, pada praktiknya, data dengan struktur seperti ini mungkin akan lebih baik jika disimpan ke dalam sebuah DataFrame menggunakan modul `pandas`, salah satu pustaka yang cukup banyak digunakan praktisi sains data.

Kasus 2 Penguraian Rumus Kimia

Tuliskan sebuah fungsi untuk mengurai (*parse*) rumus kimia dengan format `'CH3CH2Cl'` menjadi suatu `list` yang berisi simbol-simbol unsur di dalamnya sesuai dengan urutan kemunculannya (contohnya, `list` berupa `['C', 'H', 'C', 'H', 'Cl']`).
Gunakan fungsi bawaan `set` di Python untuk menentukan simbol unsur unik yang ada (dalam hal ini, `{'C', 'H', 'Cl'}`) dan kemudian kategorikan apakah rumus tersebut merupakan senyawa hidrokarbon (hanya mengandung karbon dan hidrogen) atau bukan.

Solusi

Fungsi berikut ini bertugas memeriksa simbol unsur di dalam string rumus dan mengembalikannya dalam bentuk `list`.

```
def get_element_symbols(formula):
    """Mengembalikan semua simbol unsur di dalam string formula."""
    n = len(formula)
    i = 0
    symbols = []

    # Iterasi melalui string, perhatikan huruf kapital yang
    # menandakan awal dari sebuah simbol unsur.
    while i < n:
        if formula[i].isupper():
            # Sebuah simbol unsur baru
            symbols.append(formula[i])
        elif formula[i].islower():
            # Jika menemui huruf kecil, ia adalah karakter kedua
            # dari sebuah simbol unsur.
            symbols[-1] = symbols[-1] + formula[i]
        i += 1

    return symbols

# Uji fungsi tersebut.
formula = 'CH3CH2Cl'
symbols = get_element_symbols(formula)
print(formula, '->', symbols)
```

Output:

```
CH3CH2Cl -> ['C', 'H', 'C', 'H', 'Cl']
```

Untuk menentukan simbol unsur unik, kita tinggal membuat `set` dari `list` tersebut:

```
set(symbols)
```

Output:

```
{'C', 'Cl', 'H'}
```

```
# Pengujian lain.
formula = 'CH3CBr2CO2H'
symbols = get_element_symbols(formula)

print(formula, '->', symbols)
print('Unique symbols:', set(symbols))
```

Output:

```
CH3CBr2CO2H -> ['C', 'H', 'C', 'Br', 'C', 'O', 'H']
Unique symbols: {'Br', 'C', 'H', 'O'}
```

Dengan pendekatan ini, kita dapat mendefinisikan sebuah fungsi khusus untuk memastikan apakah suatu rumus kimia tergolong senyawa hidrokarbon atau tidak:

```
def is_hydrocarbon(formula):
    symbol_set = set(get_element_symbols(formula))
    hydrocarbon_set = {'C', 'H'}
    return hydrocarbon_set == symbol_set

is_hydrocarbon('CH3CH2Cl')
```

Output:

```
False
```

```
is_hydrocarbon('CH3CH3')
```

Output:

```
True
```

Kasus 3 Struktur Kisi Kristal

Mari kita tinjau struktur kisi kristal dari unsur-unsur logam transisi baris pertama:

- fcc: Cu, Co, Fe, Mn, Ni, Sc
- bcc: Cr, Fe, Mn, Ti, V
- hcp: Co, Ni, Sc, Ti, Zn

Beberapa unsur memiliki struktur kristal yang berbeda pada kondisi suhu dan tekanan yang berbeda. Gunakan `set` pada Python untuk mengelompokkannya dan tentukan logam mana saja yang:

(a) hanya eksis dalam struktur *face-centered cubic* (fcc), *body-centered cubic* (bcc), atau *hexagonal close-packed* (hcp);

(b) eksis dalam dua dari struktur-struktur ini;

(c) tidak membentuk struktur hcp.

Apakah ada di antara logam-logam tersebut yang eksis dalam ketiga struktur itu?

Solusi

Kita dapat menggunakan operasi himpunan (`set`) pada Python seperti gabungan (`|`), irisan (`&`), dan selisih (`-`) untuk menjawab pertanyaan-pertanyaan ini.

```python
# Definisikan himpunan untuk setiap struktur kristal
fcc = {'Cu', 'Co', 'Fe', 'Mn', 'Ni', 'Sc'}
bcc = {'Cr', 'Fe', 'Mn', 'Ti', 'V'}
hcp = {'Co', 'Ni', 'Sc', 'Ti', 'Zn'}

# Kumpulkan semua logam transisi baris pertama yang ada
all_metals = fcc | bcc | hcp

# (a) Logam yang HANYA eksis di satu struktur saja
only_fcc = fcc - bcc - hcp
only_bcc = bcc - fcc - hcp
only_hcp = hcp - fcc - bcc
only_one = only_fcc | only_bcc | only_hcp
print("(a) Hanya di satu struktur :", only_one)

# (b) Logam yang eksis tepat di dua struktur
fcc_bcc = (fcc & bcc) - hcp
fcc_hcp = (fcc & hcp) - bcc
bcc_hcp = (bcc & hcp) - fcc
in_two = fcc_bcc | fcc_hcp | bcc_hcp
print("(b) Eksis di dua struktur  :", in_two)

# (c) Logam yang tidak membentuk struktur hcp
not_hcp = all_metals - hcp
print("(c) Tidak membentuk hcp    :", not_hcp)

# Apakah ada yang eksis di ketiga struktur?
all_three = fcc & bcc & hcp
print("Eksis di ketiga struktur   :", all_three)
```

Output:

```
(a) Hanya di satu struktur : {'V', 'Cr', 'Zn', 'Cu'}
(b) Eksis di dua struktur  : {'Mn', 'Co', 'Sc', 'Ni', 'Ti', 'Fe'}
(c) Tidak membentuk hcp    : {'V', 'Fe', 'Mn', 'Cr', 'Cu'}
Eksis di ketiga struktur   : set()
```

Berdasarkan *output* di atas, kita dapat menyimpulkan bahwa tidak ada satu pun logam transisi baris pertama dari daftar tersebut yang memiliki ketiga struktur kisi kristal sekaligus karena hasilnya adalah `set()` yang berarti himpunan kosong.

Serial Python Minimalis:
Tutorial #07-Masukan/Keluaran Berkas (*File IO*)

Ahmad R. T. Nugraha

ver. 21 April 2026

Sering kali kita perlu membaca data ke dalam program Python, atau menuliskan data yang dihasilkannya ke sebuah berkas (*file*). Untuk keperluan ini, kode Python dapat berinteraksi dengan file data eksternal melalui sebuah objek file (*file object*), yang dibuat menggunakan fungsi bawaan `open`.

1 Menulis File

Untuk membuat objek file yang akan ditulis, kita dapat memberikan fungsi `open` nama filenya, dan menetapkan `'w'` (*write*) sebagai modenya. Di sini, kita akan menetapkan variabel `f` ke objek file yang diberikan:

```
f = open('output_file.txt', 'w')
```

Output dari fungsi `print` yang biasanya dikirim ke layar kemudian dapat dialihkan ke objek file yang terbuka ini dengan menambahkan argumen `file=f`:

```
print('a line of data', file=f)
```

Setelah selesai menuliskan semua output, objek file harus ditutup dengan memanggil fungsi `close`-nya:

```
f.close()
```

Sekarang, jika kita periksa file teks `output_file.txt`, akan didapati bahwa file tersebut berisi satu baris bertuliskan `a line of data`.

Pada praktiknya dalam Python, lebih disarankan untuk menggunakan sintaksis yang sedikit berbeda untuk memasukkan dan mengeluarkan data ke sebuah file:

```
with open('output_file.txt', 'w') as f:
    print('a line of data', file=f)
    print('a second line of data', file=f)
```

Kata kunci `with` menciptakan sebuah konteks (*context*) agar objek file `f` dapat digunakan. Konteks ini memastikan bahwa file tersebut secara otomatis ditutup setelah blok kodenya (yang diindentasi) selesai dieksekusi sehingga tidak perlu lagi memanggil `f.close()`. Selain itu, semua kesalahan apa pun yang mungkin terjadi selama proses penulisan file dapat ditangani guna meminimalkan risiko kehilangan data.

Untuk menulis isi yang sama persis dari sebuah string ke file (tanpa tambahan karakter baris baru (*newline*) atau sejenisnya), kita panggil metode `f.write`:

```
with open("data.txt", "w") as f:
    f.write("text1")
    f.write("text2")
```

Hasilnya adalah file teks dengan isi `text1text2` (tanpa karakter baris baru atau spasi apa pun).

2 Membaca File

Untuk membaca file data teks, buka dengan mode `'r'` (*read*). Mode ini adalah bawaan (*default*) fungsi `open` sehingga sebenarnya tidak perlu dituliskan secara eksplisit.

Ada beberapa metode objek file untuk membaca dari file yang sedang terbuka:

- `f.read()` (untuk membaca keseluruhan file),
- `f.readline()` (membaca satu baris tunggal dari file),
- atau `f.readlines()` (membaca semua baris dan menjadikannya sebuah `list` berisi string).

```
with open('output_file.txt') as f:
    data = f.read()

data
```

Output:

```
'a line of data\na second line of data\n'
```

Perhatikan adanya karakter penanda akhir baris (*line-ending*) yang di-*escape*, yaitu `\n`. Untuk melihat string data tersebut sebagaimana tampilannya di penyunting teks (*text editor*), kita dapat mencetaknya:

```
print(data)
```

Output:

```
a line of data
a second line of data
```

Sebagai alternatif:

```
with open('output_file.txt') as f:
    data = f.readlines()

data
```

Output:

```
['a line of data\n', 'a second line of data\n']
```

Di sini, `data` adalah sebuah `list` Python yang berisi dua baris dari file tersebut (termasuk karakter akhir barisnya).

Tabel 1: Mode buka file pada Python

Karakter	Deskripsi
'r'	buka untuk membaca (*default*)
'w'	buka untuk menulis, menimpa file jika sudah ada
'x'	buka untuk menulis, akan gagal jika file sudah ada
'a'	buka untuk menulis, menambahkan ke akhir file (*append*) jika file sudah ada
't'	baca/tulis data teks (*default*)
'b'	baca/tulis data biner
'+'	buka untuk pembaruan (*updating* – membaca dan menulis)

Python juga dapat membaca dan menulis file biner, serta menambahkan isi baru ke file yang sudah ada dengan menyertakan tanda (*flag*) tambahan ke dalam argumen mode (lihat tabel 1). Misalnya, `open('file.dat', 'xb')` membuka file bernama `'file.dat'` untuk menulis data biner, tetapi akan gagal dieksekusi jika file tersebut ternyata sudah ada sebelumnya.

3 Pengodean Karakter (Character Encoding)

Komputer merepresentasikan karakter teks yang dapat dibaca manusia sebagai sekumpulan angka. Pemetaan yang mengaitkan setiap angka (titik kode / *code point*) ke karakter tertentu disebut dengan pengodean karakter (*character encoding*).

Banyak komputer di era awal menggunakan pengodean ASCII (*American Standard Code for Information Interchange*) yang merepresentasikan himpunan 95 karakter sebagai angka antara 32 dan 126 (misalnya, 65 = "A"). Sayangnya, hingga saat ini, belum ada standar yang disepakati secara luas untuk pengodean perluasan (*expanded encoding*) yang diperlukan untuk merepresentasikan ribuan karakter yang digunakan dalam ratusan bahasa manusia yang berbeda-beda. Selain itu, sistem operasi dan platform yang berbeda sering kali menggunakan sistem pengodean yang berbeda pula.

Jika kita pernah melihat teks yang ditampilkan berantakan seperti "caf¿" atau "cafÃ©" padahal seharusnya "café", penyebabnya kemungkinan besar adalah ketidakcocokan antara pengodean string yang diasumsikan oleh program dan yang sebenarnya digunakan oleh si penulis aslinya.

Pengodean yang digunakan oleh Python dalam membaca dan menulis file teks bergantung pada platform sistem operasi. File yang ditulis serta dibaca pada platform yang sama seharusnya tidak akan mengalami perubahan apa-apa. Namun, terkadang kita perlu menentukan pengodean secara eksplisit saat membuka suatu file, terutama jika file tersebut berasal dari tempat atau sistem operasi lain. Pengodean secara eksplisit dapat dilakukan menggunakan argumen `encoding`. Karena pengodean karakter yang paling banyak diadopsi penggunaannya di dunia saat ini adalah UTF-8, kita bisa mengadopsinya sebagai titik awal:

```
with open("data.txt", encoding="utf8") as f:
    print(f.read())
```

Pengecualian utama dari tren ini adalah (tentu saja) Microsoft Windows, yang secara historis lebih menyukai pengodean `cp1252`. Sebenarnya terdapat sedikit irisan (*overlap*) di antara sistem-sistem pengodean karakter (misalnya, 95 karakter cetak pertama yang dikodekan oleh UTF8 sama persis dengan yang ada di set karakter ASCII), tetapi jika proses membaca file gagal atau memberikan hasil yang kacau karena alasan tertentu, kita perlu mendeduksi atau menebak pengodeannya dengan cara tertentu.

Studi Kasus

Kasus 1 Menulis dan Membaca File Teks

Tuliskan sebuah file teks bernama `atomic_numbers.txt` yang berisi daftar simbol dari sepuluh unsur pertama pada tabel periodik beserta nomor atomnya, Z.

Solusi

Pertama-tama, buatlah sebuah `list` terurut yang berisi simbol-simbol unsur tersebut. Nomor atom dari setiap unsur adalah indeksnya di dalam `list` ditambah satu (karena indeks pada Python dimulai dari nol).

```
elements = 'H He Li Be B C N O F Ne'
elements = elements.split()
elements
```

Output:

```
['H', 'He', 'Li', 'Be', 'B', 'C', 'N', 'O', 'F', 'Ne']
```

Untuk menulis ke dalam file teks, buatlah sebuah objek file, `fo`, dalam mode tulis (`'w'`) lalu iterasi `list` elemen tersebut. Kita dapat menggunakan fungsi `enumerate` untuk mendapatkan nilai indeks dan simbol unsur secara bersamaan pada setiap iterasinya:

```
with open('atomic_numbers.txt', 'w') as fo:
    for i, symbol in enumerate(elements):
        print(f'{i+1:2d} {symbol}', file=fo)
```

Untuk membaca file teks tersebut, kita dapat membuat objek file yang lain, kali ini dalam mode baca (`'r'`) dan membaca seluruh barisnya menjadi sebuah `list` yang berisi sekumpulan string:

```
with open('atomic_numbers.txt', 'r') as fi:
    lines = fi.readlines()

lines
```

Output:

```
[' 1 H\n', ' 2 He\n', ' 3 Li\n', ' 4 Be\n', ' 5 B\n', ' 6 C\n', ' 7
↪  N\n', ' 8 O\n', ' 9 F\n', '10 Ne\n']
```

Setiap baris di atas mencakup karakter baris baru (*newline*), yaitu `'\n'`, tepat di bagian akhirnya. Untuk membuat outputnya terlihat lebih rapi, kita bisa mengiterasi `list` ini dan mengurai (*parse*) kolom-kolomnya secara terpisah. Terdapat dua kolom pada setiap baris yang dipisahkan oleh spasi kosong (*whitespace*):

```
with open('atomic_numbers.txt', 'r') as fi:
    for line in fi.readlines():
        fields = line.split()
        Z = int(fields[0])
        symbol = fields[1]
        print(f'Z({symbol:2s}) = {Z:2d}')
```

Output:

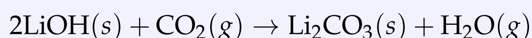
```
Z(H ) =  1
Z(He) =  2
Z(Li) =  3
Z(Be) =  4
Z(B ) =  5
Z(C ) =  6
Z(N ) =  7
Z(O ) =  8
Z(F ) =  9
Z(Ne) = 10
```

Kasus 2 Termodinamika Reaksi

File `thermo-data.csv` (yang dapat diunduh dari folder `dat` pada repositori `https://github.com.com/BRIN-Q/Python-minimalis`), berisi data termodinamika untuk beberapa senyawa pada kondisi tekanan standar ($p = 1$ bar) dan pada $T = 298$ K dengan format:

```
Formula, DfHm/kJ.mol-1, Sm/J.K-1.mol-1
H2O(g), -241.83, 188.84
CO2 (g), -393.52, 213.79
LiOH(s), -484.93, 42.81
Li2CO3(s), -1216.04, 90.31
```

Bacalah data ini ke dalam *dictionary* yang sesuai untuk entalpi pembentukan molar standar dan entropi molar standar. Gunakan data tersebut untuk menghitung konstanta kesetimbangan dari reaksi berikut, yang menjadi dasar operasi pembuangan karbon dioksida dari kabin awak pada beberapa wahana antariksa:



$$2\text{LiOH}(s) + \text{CO}_2(g) \rightarrow \text{Li}_2\text{CO}_3(s) + \text{H}_2\text{O}(g)$$

Solusi

Konstanta kesetimbangan (K) dapat dihitung dari perubahan energi bebas Gibbs standar ($\Delta_r G^\ominus$), dengan $\Delta_r G^\ominus = \Delta_r H^\ominus - T\Delta_r S^\ominus$.

```python
import math

DfH = {}
S = {}

with open('dat/thermo-data.csv', 'r') as f:
    header = f.readline()
    for line in f:
        fields = line.split(',')
        formula = fields[0].strip() # Menghapus spasi ekstra jika ada
        DfH[formula] = float(fields[1])
        S[formula] = float(fields[2])

# Reaksi: 2 LiOH(s) + CO2(g) -> Li2CO3(s) + H2O(g)
# Stoikiometri (produk bernilai positif, reaktan bernilai negatif)
nu = {'LiOH(s)': -2, 'CO2 (g)': -1, 'Li2CO3(s)': 1, 'H2O(g)': 1}

# Hitung Delta H dan Delta S reaksi
DrH = sum(nu[comp] * DfH[comp] for comp in nu) # dalam kJ/mol
DrS = sum(nu[comp] * S[comp] for comp in nu)   # dalam J/(K.mol)

# Hitung Delta G reaksi dan Konstanta Kesetimbangan
T = 298.15
DrG = DrH * 1000 - T * DrS # konversi DrH ke J/mol
R = 8.31446

K = math.exp(-DrG / (R * T))

print(f"Entalpi Reaksi (DrH)   = {DrH:.2f} kJ/mol")
print(f"Entropi Reaksi (DrS)   = {DrS:.2f} J/(K.mol)")
print(f"Energi Gibbs (DrG)     = {DrG/1000:.2f} kJ/mol")
print(f"Konstanta Kesetimbangan= {K:.2e}")
```

Output:

```
Entalpi Reaksi (DrH)   = -94.49 kJ/mol
Entropi Reaksi (DrS)   = -20.26 J/(K.mol)
Energi Gibbs (DrG)     = -88.45 kJ/mol
Konstanta Kesetimbangan= 3.12e+15
```

Reaksi ini sangat disukai secara termodinamika (spontan) ke arah produk, dibuktikan dengan nilai tetapan kesetimbangan K yang sangat besar ($\sim 10^{15}$).

Serial Python Minimalis:
Tutorial #08-Larik 1D dengan NumPy

Ahmad R. T. Nugraha

ver. 21 April 2026

Kita telah mengetahui bahwa `list` pada Python merupakan urutan item yang dapat diindeks, diiterasi, ditambahkan elemen baru ke dalamnya (*append*), dan lain sebagainya:

```
lst = [1, 'a', 'pi', -0.25]
lst[2]
```

Output:

```
'pi'
```

```
lst.append(99.)
for item in lst:
    print(item)
```

Output:

```
1
a
pi
-0.25
99.0
```

Penggunaan `list` memang praktis untuk menyimpan sejumlah kecil data yang heterogen, tetapi kurang cocok untuk komputasi numerik berskala besar. Kreasi dan iterasi `list` bawaan Python bisa jadi lambat dan merepotkan. Sebagai contoh, misalkan kita ingin menghitung fungsi $y = \sin(x)$ untuk sebuah `list` berisi 10 nilai x yang berjarak sama dari 0 hingga 2π:

```
import math

xvals, yvals = [], []
n = 10
dx = 2 * math.pi / (n - 1)
```

```
for i in range(n):
    x = i * dx
    xvals.append(x)
    yvals.append(math.sin(x))
```

```
xvals
```

Output:

```
[0.0, 0.6981317007977318, 1.3962634015954636, 2.0943951023931953,
↪ 2.792526803190927, 3.490658503988659, 4.1887902047863905,
↪ 4.886921905584122, 5.585053606381854, 6.283185307179586]
```

```
yvals
```

Output:

```
[0.0, 0.6427876096865393, 0.984807753012208, 0.8660254037844388,
↪ 0.3420201433256689, -0.34202014332566866, -0.8660254037844384,
↪ -0.9848077530122081, -0.6427876096865396, -2.4492935982947064e-16]
```

Proses di atas cukup merepotkan. Kita harus menghitung jarak spasi antarnilai x, yaitu $\Delta x = \frac{2\pi}{n-1}$ untuk n titik, lalu perlahan-lahan mengiterasinya menggunakan sebuah pencacah (*counter*), i, yang bergerak dari 0 hingga $n - 1$ guna membangun `list` `xvals` dan `yvals` satu demi satu menggunakan metode `append`.

Pustaka *Numerical Python* (NumPy) menyediakan cara yang jauh lebih efisien untuk mencapai hasil yang sama tanpa memerlukan perulangan (*loops*) secara eksplisit.

```
import numpy as np

x = np.linspace(0, 2 * np.pi, 10)
y = np.sin(x)
```

```
x
```

Output:

```
array([0.        , 0.6981317 , 1.3962634 , 2.0943951 , 2.7925268 ,
       3.4906585 , 4.1887902 , 4.88692191, 5.58505361, 6.28318531])
```

```
y
```

Output:

```
array([ 0.00000000e+00,  6.42787610e-01,  9.84807753e-01,  8.66025404e-01,
        3.42020143e-01, -3.42020143e-01, -8.66025404e-01, -9.84807753e-01,
       -6.42787610e-01, -2.44929360e-16])
```

Lebih jauh lagi, aplikasi aljabar linear secara umum sangat bergantung pada penggunaan entitas yang bersifat homogen dan berukuran tetap, seperti matriks matematika. Pada Python standar, matriks sebenarnya dapat direpresentasikan menggunakan `list` di dalam `list` (*lists of lists*). Akan tetapi, pendekatan ini tergolong kaku dan merepotkan karena bentuk sintaksisnya jauh berbeda dari notasi matematika yang selama ini kita kenal. Dalam konteks inilah, larik NumPy (NumPy *arrays*) menjadi struktur data andalan untuk komputasi numerik.

Sebelum membahas kasus multidimensi (khususnya dua dimensi), kita akan mempelajari larik (*array*) NumPy satu dimensi (1D) terlebih dahulu. Larik 1D ini dapat digunakan sebagai pengganti langsung untuk `list` pada Python. Ke depannya, kita tidak akan terus-menerus menyebut wadah data baru ini dengan nama panjang "NumPy *array*". Kita akan menyederhanakannya menjadi "larik" atau "*array*" saja, sama halnya seperti `list` bawaan inti Python cukup disebut "*list*".

1 Pembuatan Larik (*Array*) pada NumPy

Kita mulai dengan cara yang biasa, yaitu menggunakan `list` Python:

```
L = [11, 7, 19, 22, 55]
```

Langkah selanjutnya adalah mengimpor fungsionalitas NumPy yang dapat dilakukan dengan berbagai cara. Di sini kita menggunakan konvensi standar, yaitu mengimpor NumPy dengan nama singkatan:

```
import numpy as np
```

Cara termudah untuk membuat larik adalah dengan menggunakan fungsi `array()`, yang menerima entitas berbentuk urutan (*sequence-like entity*) dan mengembalikan sebuah larik yang berisi data yang diteruskan itu. Pada contoh kita, dengan menggunakan `list` `L`, kita mendapatkan:

```
xs = np.array(L)
xs
```

Output:

```
array([11,  7, 19, 22, 55])
```

Perhatikan bahwa fungsi `array()` merupakan bagian dari NumPy, sehingga kita harus menuliskan `np.array()` untuk mengaksesnya. Contoh ini dengan jelas menunjukkan bahwa *interpreter* Python memahami bahwa `xs` bukanlah sebuah `list`, melainkan sebuah objek larik. Perlu dicatat bahwa di sini terdapat fungsi `array()` dan objek larik yang saling berkaitan, dengan kata lain, fungsi `array()` itulah yang menciptakan objeknya. Perlu ditekankan juga bahwa ketika kita mencetak (*print*) `list` dan larik, outputnya akan terlihat sedikit berbeda:

```
print(xs)
print(L)
```

Output:

```
[11  7 19 22 55]
[11, 7, 19, 22, 55]
```

Tampilan larik dari NumPy di atas dapat kita lihat tidak menunjukkan tanda koma sehingga sangat membantu ketika kita perlu berfokus pada angka-angkanya itu sendiri, yang memang menjadi tujuan utama dirancangnya larik. Jika kita ingin melihat berapa banyak elemen yang terdapat di dalam larik `xs`, kita mungkin tergoda untuk mengambil fungsi bawaan inti Python `len()`. Namun, teknik ini dapat menyebabkan masalah di kemudian hari, sehingga jauh lebih baik menggunakan atribut `size` dari array `xs`:

```
xs.size
```

Output:

```
5
```

Ingat bahwa larik NumPy memiliki panjang yang tetap (*fixed-length*), sehingga jumlah total elemennya tidak dapat berubah. Dengan kata lain, tidak ada metode `append()` untuk larik sehingga ukurannya akan selalu tetap. Perilaku ini berbeda dengan `list`, yang jika dipanggil oleh `len(L)`, akan mengembalikan nilai yang berbeda sebelum dan sesudah kita menambahkan/menghapus elemen.

Atribut lain yang sangat berguna yang dimiliki larik NumPy adalah `dtype`, yakni tipe data dari elemen-elemen di dalamnya:

```
xs.dtype
```

Output:

```
dtype('int64')
```

Kita dapat langsung melihat bahwa meskipun kita tidak mendefinisikan tipe data secara eksplisit, elemen-elemen lariknya dapat menyimpulkan sendiri harus bertipe data apa. Ingat setiap larik NumPy bersifat homogen, yang berarti semua elemennya wajib memiliki tipe yang sama. Kita dapat melihat tipe yang dikembalikan bukanlah integer Python standar, melainkan `int64`. Angka ini memberi tahu kita berapa banyak bit yang digunakan untuk menyimpan integer tersebut dalam representasi mesin yang spesifik (64 bit membentuk apa yang oleh sebagian besar bahasa pemrograman disebut sebagai "*long integer*"). Kita juga dapat menggunakan `type()` bawaan Python untuk memeriksa tipe-tipe yang berbeda ini:

```
type(L)
```

Output:

```
list
```

```
type(L[0])
```

Output:

```
int
```

```
type(xs)
```

Output:

```
numpy.ndarray
```

```
type(xs[0])
```

Output:

```
numpy.int64
```

Output ini dengan jelas menunjukkan bahwa kita sedang berhadapan dengan tipe-tipe data yang berbeda. Tipe data NumPy dipetakan secara langsung ke representasi mesin.

Kita juga boleh memiliki larik yang berisi bilangan pecahan (*floats*):

```
ys = np.array([1.1, 2.2, 3.3, 4.4])
ys.dtype
```

Output:

```
dtype('float64')
```

Di sini kita menempatkan `list` yang digunakan untuk membuat array secara langsung sebagai argumen pada fungsi `array()`. Angka 64 di atas memberi tahu kita bahwa kita sedang berhadapan dengan bilangan pecahan presisi ganda (*double-precision float*). Pada contoh di atas, yakni `xs` dan `ys`, kita membiarkan NumPy menyimpulkan sendiri tipe argumennya. Namun, kita juga dapat memberikan tipe data tersebut sebagai argumen (opsional) eksplisit ke dalam fungsi `array()`:

```
zs = np.array([5, 6, 7, 8], dtype=np.float32)
zs
zs.dtype
```

Output:

```
dtype('float32')
```

Kita sengaja menggunakan `list` berisi bilangan bulat (integer) dan meminta NumPy untuk mempromosikannya menjadi bilangan pecahan (presisi tunggal). Dengan demikian, NumPy memberi kita kendali penuh (*fine-grained control*) atas representasi mesin mana yang ingin kita gunakan, khususnya untuk *floats*, yang dapat memiliki konsekuensi praktis yang besar dan merupakan salah satu nilai jual utama dari larik NumPy.

Sejauh ini, kita baru melihat cara membuat larik NumPy menggunakan bantuan `list` biasa (sebagai alat sementara). Sebagai contoh, jika kita menginginkan sebuah larik yang berisi sejumlah angka nol, mungkin kita akan melakukan sesuatu seperti:

```
np.array([0.]*5)
```

Output:

```
array([0., 0., 0., 0., 0.])
```

sedangkan untuk menghasilkan larik yang berisi sejumlah angka satu kita mungkin akan menuliskan:

```
np.array([1.]*4)
```

Output:

```
array([1., 1., 1., 1.])
```

Kasus-kasus penggunaan ini sangatlah umum sehingga NumPy memiliki fungsi bawaan untuk menghasilkan larik semacam itu secara langsung:

```
np.zeros(5)
```

Output:

```
array([0., 0., 0., 0., 0.])
```

```
np.ones(4)
```

Output:

```
array([1., 1., 1., 1.])
```

Seperti yang bisa kita lihat, kita meneruskan jumlah total elemen sebagai argumen pada setiap pemanggilan.

Fungsi NumPy lain yang sangat berguna untuk menghasilkan larik adalah `arange()`, yang merupakan generalisasi dari fungsi `range()` bawaan inti Python:

```
np.arange(10,16)
```

Output:

```
array([10, 11, 12, 13, 14, 15])
```

Secara alamiah kita dapat memberikan langkah (*stride*) sebagai argumen ketiga pada `arange()`, sama seperti yang kita lakukan pada `range()`. Pada titik ini, mereka yang baru mengenal NumPy sering kali baru menyadari bahwa `arange()` ternyata dapat menerima *floats* sebagai argumen:

```
np.arange(10.,16.)
```

Output:

```
array([10., 11., 12., 13., 14., 15.])
```

Namun, kita akan segera lihat bahwa teknik ini dapat mengundang masalah. Berikut adalah dua contoh yang menunjukkan kemungkinan kekacauan yang dihasilkan.

```
np.arange(1.5,1.75,0.05)
```

Output:

```
array([1.5 , 1.55, 1.6 , 1.65, 1.7 ])
```

```
np.arange(1.5,1.8,0.05)
```

Output:

```
array([1.5 , 1.55, 1.6 , 1.65, 1.7 , 1.75, 1.8 ])
```

Ingat kembali bahwa bilangan pecahan (dan operasi *floating-point*) tidak disimpan (atau dieksekusi) menggunakan presisi tak terbatas. Akibatnya, pada kasus pertama nilai akhirnya tidak diikutsertakan, sedangkan pada kasus kedua nilai akhirnya malah ikut masuk. Perbedaan ini bisa berdampak signifikan seperti pada kasus integrasi numerik. Saat melakukan perhitungan dengan rumus integral numerik tertentu, sangat penting agar menerapkannya secara tepat, misalnya menggunakan metode tertutup (yaitu, mengikutsertakan kedua titik akhir). Dokumentasi NumPy sangat eksplisit mengenai hal ini, "...apabila menggunakan langkah non-integer, seperti 0.1, hasilnya sering kali tidak konsisten...". Kita mungkin tergoda juga untuk menyelesaikan masalah ini dengan solusi akal-akalan seperti:

```
np.arange(1.5,1.75001,0.05)
```

Output:

```
array([1.5 , 1.55, 1.6 , 1.65, 1.7 , 1.75])
```

Namun, cara ini lagi-lagi adalah ide yang buruk. Sebagai gantinya, jauh lebih disarankan untuk menggunakan fungsi `linspace()` dari NumPy, yang mengembalikan sejumlah elemen dengan jarak yang persis sama sepanjang interval yang ditentukan:

```
np.linspace(1.5,1.75,6)
```

Output:

```
array([1.5 , 1.55, 1.6 , 1.65, 1.7 , 1.75])
```

```
np.linspace(1.5,1.8,7)
```

Output:

```
array([1.5 , 1.55, 1.6 , 1.65, 1.7 , 1.75, 1.8 ])
```

Perhatikan contoh di atas memberikan perilaku yang konsisten, tetapi dengan kompensasi bahwa kita harus memberikan sendiri jumlah elemen yang diinginkan. Kita mungkin bertanya-tanya mengapa fungsi `linspace()` diberi nama demikian. Jawabannya akan menjadi jelas setelah ditunjukkan bahwa NumPy memiliki fungsi yang disebut `logspace()`. Fungsi ini menghasilkan kisi-kisi titik (*grid of points*) dengan jarak logaritmik (perilaku bawaannya adalah menggunakan basis 10). Misalnya, jika kita menginginkan 5 titik yang didistribusikan secara logaritmik dari 10^{-1} hingga 10^{-5}, kita akan menuliskan:

```
np.logspace(-1, -5, 5)
```

Output:

```
array([1.e-01, 1.e-02, 1.e-03, 1.e-04, 1.e-05])
```

Argumen pertama adalah pangkat dari nilai awal, argumen kedua adalah pangkat dari nilai akhir, dan argumen ketiga adalah jumlah sampel yang akan dihasilkan. Tentunya jelas, rentang titik yang dihasilkan oleh `linspace()` bersifat linier. Ingat: baik `linspace()` maupun `logspace()` sama-sama menyertakan kedua titik akhirnya.

Dua perbedaan antara `list` dan larik yang kita temui sejauh ini (yakni, larik memiliki ukuran tetap dan bersifat homogen) mungkin tampak kurang begitu penting bagi kita. Toh, kita bisa saja membuat `list` yang tidak pernah bertambah/berkurang dan terdiri dari elemen dengan tipe yang sama. Sekarang kita akan melihat dua perbedaan besar lainnya, yang pertama berkaitan dengan cara kerja *slicing* (pemotongan) dan yang kedua berkaitan dengan bagaimana operasi perhitungan dipropagasikan ke berbagai elemen berbeda. Kita akan membahas masing-masing di dalam bagian tersendiri.

2 Pengindeksan dan Pemotongan (*Slicing*)

Pengindeksan untuk larik bekerja dengan cara yang persis sama seperti pada `list`, yaitu menggunakan tanda kurung siku:

```
xs = np.arange(10,16)
print(xs)
```

Output:

```
[10 11 12 13 14 15]
```

```
xs[2]
```

Output:

```
12
```

Pada pandangan pertama, *slicing* untuk larik tampak identik dengan cara kerja *slicing* pada `list`:

```
xs = np.arange(10,16)
print(xs[2:4])
```

Output:

```
[12 13]
```

Kita dapat terus menambahkan contoh lain, misal `xs[-1]` mengakses elemen terakhir dalam larik, `xs[1:5:2]` menggunakan ukuran langkah (*stride*), dan seterusnya. Namun, terdapat perbedaan yang krusial, yaitu bahwa potongan dari larik merupakan sebuah tampilan (*view*) dari larik aslinya:

```
r = np.array([11,7,19,22])
print(r)
```

Output:

```
[11  7 19 22]
```

```
sli = r[1:3]
print(sli)
```

Output:

```
[ 7 19]
```

```
sli[0] = 55
print(sli)
```

Output:

```
[55 19]
```

```
print(r)
```

Output:

```
[11 55 19 22]
```

Contoh ini persis sama dengan apa yang kita lakukan dalam tutorial Python menggunakan `list`. Seperti yang mungkin kita ingat dari sana, potongan (*slice*) pada `list` secara *de facto* menghasilkan salinan (*copy*), sehingga `list` aslinya tidak terpengaruh oleh perubahan selanjutnya pada *slice* tersebut. Sifat *slice* inilah yang mengarahkan kita untuk menggunakan `L[:]` guna mendapatkan salinan dari `list` `L`. Di sisi lain, larik NumPy dibuat efisien, menghilangkan kebutuhan untuk menyalin data. Tentu saja, kita semua harus selalu ingat bahwa beberapa *slice* yang berbeda semuanya tetap merujuk pada larik dasar yang sama. Dalam beberapa kasus, jika kita benar-benar memerlukan salinan nyata, kita dapat menggunakan `numpy.copy()`:

```
r = np.array([11,7,19,22])
sli = np.copy(r[1:3])
```

```
sli[0] = 55
print(sli)
```

Output:

```
[55 19]
```

```
print(r)
```

Output:

```
[11  7 19 22]
```

Sangat mudah dilihat bahwa kita juga bisa menyalin seluruh larik sekaligus, tanpa berdampak apa pun pada larik aslinya:

```
r2 = np.copy(r)
r2[1] = 0
print(r2)
```

Output:

```
[11  0 19 22]
```

```
print(r)
```

Output:

```
[11  7 19 22]
```

Dalam studi komputasi numerik, kita akan sering membuat salinan larik di dalam fungsi-fungsi, dengan spirit bahwa kita memengaruhi dunia eksternal hanya melalui nilai pemberian (*return value*) dari sebuah fungsi saja. Meskipun begitu, cara ini hanyalah pilihan pedagogis (pembelajaran). Pada praktiknya, kinerjanya akan jauh lebih baik jika kita memodifikasi suatu larik "langsung di tempat" (*in place*).

Perbedaan lain antara `list` dan array, yang akan diperluas pembahasannya pada tutorial-tutorial mendatang, berkaitan dengan sifat *broadcasting*. Secara kualitatif, *broadcasting* bermakna bahwa NumPy cukup tahu bagaimana menangani entitas yang dimensi-dimensinya tidak sepenuhnya cocok. Berikut adalah contohnya:

```
r = np.array([11, 7, 19, 22])
r[1:3] = 55
print(r)
```

Output:

```
[11 55 55 22]
```

Sebenarnya kode di atas adalah contoh serupa yang pernah kita coba di tutorial Python menggunakan `list`. Pada contoh tutorial tersebut, kita menerima `TypeError`, karena Python tidak tahu bagaimana menangani satu angka tunggal di sisi kanan, ketika sisi kirinya membutuhkan setidaknya dua angka (yaitu, Python mengharapkan adanya objek *iterable* di sisi kanannya). Sebaliknya, untuk kasus larik, sebuah nilai di sisi kanan disebarkan (*broadcasted*) ke sejumlah ruang (*slots*) yang diperlukan.

Karena kita baru saja membahas tentang *slicing* dan bagaimana sebuah nilai dapat di-*broadcast* ke beberapa slot sekaligus, di sini menjadi tempat yang tepat untuk melihat idiom larik NumPy yang sangat umum, yaitu "irisan segala" (*everything slice*) atau `xs[:]`. Ingat, karena *slice* dari larik adalah sekadar tampilan ke larik aslinya, `xs[:]` tidak dapat digunakan untuk membuat salinan larik. Akan tetapi, sintaksis ini cukup berguna seperti ditunjukkan di bawah.

```
xs = np.arange(10,16)
print(xs)
xs[:]  = 7
print(xs)
```

Output:

```
[10 11 12 13 14 15]
[7 7 7 7 7 7]
```

Kita menggunakan *everything-slice* untuk mem-*broadcast* angka 7 ke semua slot larik yang sudah ada sebelumnya.

Kita mungkin bertanya-tanya mengapa kita memerlukan *everything-slice*. Jawabannya seharusnya sudah jelas jika kita melihat bahwa:

```
xs = 7
print(xs)
```

Output:

```
7
```

Jika kita tidak menggunakan *everything-slice*, Python akan berpikir bahwa kita sedang membuat sebuah variabel integer baru yang kebetulan saja bernama `xs`, padahal itu (hampir bisa dipastikan) bukanlah niat kita sebenarnya.

Kita telah melihat beberapa sintaksis baru yang diizinkan oleh larik atau *array* dalam NumPy. Bagian selanjutnya akan membahas operasi larik semacam ini secara lebih terperinci dan menyoroti kekuatan-kekuatannya.

3 Vektorisasi

Saat memperkenalkan larik di atas, kita telah sebutkan bahwa larik memungkinkan kita untuk menghindari, setidaknya sebagian besar, keharusan menulis perulangan (*loops*). Sekarang saatnya untuk melihat apa maksudnya.

```
xs = np.arange(10,16)
ys = np.arange(20,26)
print(xs)
```

Output:

```
[10 11 12 13 14 15]
```

```
print(ys)
```

Output:

```
[20 21 22 23 24 25]
```

```
print(xs + ys)
```

Output:

```
[30 32 34 36 38 40]
```

Kita lihat bahwa untuk larik NumPy, penambahan (*addition*) diinterpretasikan sebagai operasi elemen demi elemen (*elementwise*), yakni setiap pasang elemen dijumlahkan secara berurutan. Kemampuan untuk melakukan operasi massal semacam ini pada seluruh elemen di dalam sebuah larik (atau dua larik) secara bersamaan sering kali disebut sebagai **vektorisasi**. Perhatikan betapa berbedanya operasi ini dari perilaku `list` Python:

```
xs = list(range(10,16))
ys = list(range(20,26))
```

```
xs
```

Output:

```
[10, 11, 12, 13, 14, 15]
```

```
ys
```

Output:

```
[20, 21, 22, 23, 24, 25]
```

```
xs+ys
```

Output:

```
[10, 11, 12, 13, 14, 15, 20, 21, 22, 23, 24, 25]
```

```
news = [x+y for x,y in zip(xs,ys)]
news
```

Output:

```
[30, 32, 34, 36, 38, 40]
```

Untuk tipe data `list`, tanda + berfungsi sebagai operator penggabungan (*concatenate*), alih-alih melakukan penambahan secara matematis per elemen. Jadi, untuk menghasilkan output yang sama dengan yang diberikan oleh larik menggunakan tanda + sederhana, pada kasus tipe data `list` kita harus menggunakan *`list` comprehension* dipadukan dengan `zip()`.

Kembali ke array NumPy, perkalian juga diinterpretasikan sebagai operasi elemen demi elemen (dan hal yang sama berlaku untuk pengurangan, pembagian, serta perpangkatan eksponen):

```
xs = np.arange(10,16)
ys = np.arange(20,26)
print(xs*ys)
```

Output:

```
[200 231 264 299 336 375]
```

```
np.sum(xs*ys)
```

Output:

```
1705
```

Di sini, kita sekaligus memperkenalkan `numpy.sum()`, yang menjumlahkan seluruh elemen dari sebuah array NumPy. Dengan demikian, baris terakhir tersebut secara sederhana sedang menghitung perkalian titik (*scalar product*) dari dua buah vektor, hanya dalam satu baris yang sangat ringkas.

Kita mungkin bertanya-tanya apakah kita bisa menggunakan fungsi `sum()` bawaan yang disediakan Python. Jawabannya adalah, meskipun kedua opsi tersebut sah saja untuk digunakan, dalam praktiknya fungsi bawaan NumPy bekerja jauh lebih cepat. Pelajaran penting di sini adalah kita harus selalu mendayagunakan fungsionalitas NumPy sebisa mungkin.

Sama menariknya dengan operasi *elementwise* antara dua array, adalah kemampuan NumPy untuk menggabungkan sebuah larik dengan skalar, yang merupakan konsekuensi dari sifat *broadcasting* yang telah disebutkan sebelumnya. NumPy tahu cara menafsirkan entitas dengan bentuk (*shapes*) yang saling berbeda namun kompatibel.

```
xs = np.arange(10,16)
print(xs)
```

Output:

```
[10 11 12 13 14 15]
```

```
print(2*xs)
```

Output:

```
[20 22 24 26 28 30]
```

Kita mungkin ingat bahwa perkalian `list` dengan sebuah skalar akan menghasilkan output yang sangat berbeda (sebenarnya, kita pernah menggunakan sifat tersebut pada bagian sebelumnya ketika kita menuliskan `[0.]*5`). Di sisi lain, NumPy tahu bagaimana cara menggabungkan skalar dengan sebuah larik untuk melakukan perkalian ke semua elemen (jadi ini adalah satu lagi kasus yang kita tidak memerlukan *`list` comprehension*). Kita dapat memikirkan apa yang terjadi di sini sebagai direntangkannya angka 2 menjadi sebuah larik yang memiliki ukuran sama dengan `xs`, lalu setelah itu perkalian *elementwise* dilakukan. Dengan begitu, tidak perlu diragukan lagi, kita juga bisa melakukan berbagai operasi lain pada larik NumPy dengan skalar.

```
xs = np.arange(10,16)
print(1/xs)
```

Output:

```
[0.1        0.09090909 0.08333333 0.07692308 0.07142857 0.06666667]
```

Kombinasi antara *broadcasting* (kita dapat melakukan operasi matematis antara skalar dengan larik) dan vektorisasi (kita hanya menuliskan sebuah ekspresi tetapi perhitungannya dieksekusi secara serentak ke semua elemen) bisa jadi agak sulit dipahami pada awalnya. Namun, fitur-fitur ini sesungguhnya sangat luar biasa tangguh (baik secara fleksibilitas ekspresi maupun efisiensi) begitu kita sudah terbiasa dengannya.

Selain tahu bagaimana cara mengeksekusi operasi *elementwise* antarlarik (maupun antara larik dan skalar), NumPy mencakup banyak fungsi yang terlihat mirip dengan padanan fungsi matematisnya. Misalnya, `numpy.sqrt()`, `numpy.exp()`, `numpy.log()`, `numpy.sin()` bekerja sesuai dengan ekspektasi kita. Perbedaan krusial dari fungsi matematika biasanya adalah bahwa fungsi-fungsi NumPy ini sudah dirancang untuk memproses larik utuh sehingga nyaris selalu bekerja lebih cepat. Fungsi-fungsi NumPy lain yang mungkin masih asing bagi kita, tetapi bisa sangat berguna, antara lain `numpy.isfinite()` atau `numpy.sort()`.

Perhatikan bahwa seluruh fungsi yang telah kita bahas sejauh ini bekerja dengan menerima input larik dan mengembalikan sebuah larik. NumPy juga memiliki fungsi yang menerima larik dan mengembalikan nilai skalar, seperti halnya `numpy.sum()` yang sudah kita temui di atas. Fungsi-fungsi ini diberi nama yang intuitif, misalnya `numpy.prod()` menghitung produk perkalian dari seluruh elemen dan `numpy.amin()` mengembalikan nilai elemen terkecil. Fungsi lainnya yang sangat membantu adalah `numpy.argmax()`, yang mengembalikan letak indeks dari nilai terbesarnya.

NumPy bahkan memiliki satu buah modul utuh bernama `numpy.random`, dengan ragam fungsi sangat berguna dan teroptimasi yang terkait dengan angka-angka acak (*random numbers*). Sangat direkomendasikan untuk meluangkan sedikit waktu guna membaca dokumentasi NumPy agar kita lebih familier dengan fungsi-fungsi semacam ini, sebab fungsi tersebut sangatlah intuitif dan efisien.

Kita juga dapat menggunakan fungsionalitas di dalam NumPy untuk meracik fungsi bebas buatan kita sendiri yang tahu cara memproses larik. Contohnya, pada kode berikut:

```
def fa(xs):
    return 1/np.sqrt(xs**2+1)
xs = np.arange(10,16)
print(fa(xs))
```

Output:

```
[0.09950372 0.09053575 0.08304548 0.0766965  0.07124705 0.06651901]
```

Kita manfaatkan `numpy.sqrt()`, yang sudah tahu cara menangani argumen larik, lalu operasi pemangkatan (*squaring*), dan operasi penambahan angka 1, masing-masing ditangani langsung oleh NumPy itu sendiri (alias, ditafsirkan sebagai sesuatu yang merujuk pada larik utuh juga).

Sekarang, kita siap untuk mengimplementasikan ulang suatu fungsi bernama

“tots”. Alih-alih menuliskan kode dengan cara list berikut,

```
import math
def f(x):
    return 1/math.sqrt(x**2+1)
ws = list(range(30,36))
xs = list(range(10,16))
tots = [w*f(x) for w,x in zip(ws,xs)]
print(tots)
```

Output:

```
[2.9851115706299676, 2.8066081273180745, 2.657455355319679,
↪  2.5309844631963223, 2.422399699588928, 2.3281653683320878]
```

solusi baru kita, yang sepenuhnya memakai larik, secara sederhana adalah:

```
def fa(xs):
    return 1/np.sqrt(xs**2+1)
ws = np.arange(30,36)
xs = np.arange(10,16)
tots = ws*fa(xs)
print(tots)
```

Output:

```
[2.98511157 2.80660813 2.65745536 2.53098446 2.4223997  2.32816537]
```

Perhatikan bagaimana baris:

```
tots = [w*f(x) for w,x in zip(ws,xs)]
```

yang merupakan suatu *list comprehension*, digantikan menjadi:

```
tots = ws*fa(xs)
```

Sekali lagi, begitu kita terbiasa dengan metode semacam ini, kita akan benar-benar memahami betapa praktisnya eksekusi operasi-operasi matematis menggunakan larik dari NumPy.

Studi Kasus

Kasus 1

Buatlah sebuah program Python dengan NumPy untuk melakukan operasi berikut pada sebuah larik satu dimensi $x = [2, 4, 6, 8, 10]$, yaitu: (1) Hitung jumlah seluruh elemen, nilai rata-rata elemen, nilai elemen terbesar,dan nilai elemen terkecil; (2) Bentuk larik baru yang setiap elemennya merupakan hasil

dari $y = 3x + 1$; dan (3) Tampilkan semua hasilnya.

Solusi

Kita bentuk terlebih dahulu larik satu dimensi x menggunakan `np.array`. Selanjutnya, beberapa besaran dasar dapat dihitung dengan fungsi-fungsi NumPy seperti `np.sum`, `np.mean`, `np.max`, dan `np.min`. Untuk membentuk larik baru y, kita cukup menuliskan ekspresi $3x + 1$ secara langsung, karena NumPy akan menerapkan operasi tersebut ke seluruh elemen larik.

```python
import numpy as np

# Membentuk larik satu dimensi
x = np.array([2, 4, 6, 8, 10])

# Menghitung beberapa besaran
jumlah = np.sum(x)
rata_rata = np.mean(x)
maksimum = np.max(x)
minimum = np.min(x)

# Operasi vektor pada seluruh elemen
y = 3 * x + 1

# Menampilkan hasil
print("x =", x)
print("Jumlah elemen =", jumlah)
print("Rata-rata =", rata_rata)
print("Elemen terbesar =", maksimum)
print("Elemen terkecil =", minimum)
print("y = 3x + 1 =", y)
```

Output:

```
x = [ 2  4  6  8 10]
Jumlah elemen = 30
Rata-rata = 6.0
Elemen terbesar = 10
Elemen terkecil = 2
y = 3x + 1 = [ 7 13 19 25 31]
```

Sekarang kita jelaskan hasilnya. Untuk larik

$$x = [2, 4, 6, 8, 10],$$

jumlah seluruh elemennya adalah

$$2 + 4 + 6 + 8 + 10 = 30.$$

Karena banyak elemen ada 5, rata-ratanya adalah

$$\frac{30}{5} = 6.$$

Elemen terbesar pada larik tersebut adalah

$$10,$$

sedangkan elemen terkecilnya adalah

$$2.$$

Selanjutnya, larik baru dibentuk dari hubungan

$$y = 3x + 1.$$

Artinya, setiap elemen pada x dikalikan 3, lalu ditambah 1. Dengan demikian diperoleh

$$y = [3(2) + 1,\ 3(4) + 1,\ 3(6) + 1,\ 3(8) + 1,\ 3(10) + 1]$$
$$= [7, 13, 19, 25, 31].$$

Kasus 2

Buatlah sebuah program Python dengan NumPy untuk menghasilkan 11 titik yang terdistribusi **merata** pada interval

$$0 \leq x \leq 5$$

menggunakan `numpy.linspace`. Hati-hati perhatikan batas tepat kiri maupun kanan pada inverval tersebut terkait dengan tanda kesamaan/ketaksamaan. Setelah itu:

1. hitung nilai fungsi $f(x) = x^2 - 2x + 3$ pada setiap titik tersebut,
2. tentukan indeks elemen tempat nilai fungsi terbesar berada,
3. tampilkan titik-titik x dan nilai $f(x)$.

Tuliskan programnya dan jelaskan hasilnya.

Solusi

Pada soal ini, kita gunakan `np.linspace(0, 5, 11)` untuk membentuk 11 titik yang berjarak sama dari 0 sampai 5. Setelah itu, fungsi

$$f(x) = x^2 - 2x + 3$$

dihitung pada semua elemen larik x sekaligus. Untuk menentukan letak nilai fungsi terbesar, kita gunakan `np.argmax`.

```python
import numpy as np

# Membentuk 11 titik dari 0 sampai 5
x = np.linspace(0, 5, 11)

# Menghitung nilai fungsi pada setiap titik
f = x**2 - 2*x + 3

# Menentukan indeks nilai fungsi terbesar
indeks_maks = np.argmax(f)

# Menampilkan hasil
print("x =", x)
print("f(x) =", f)
print("Indeks nilai fungsi terbesar =", indeks_maks)
print("Nilai fungsi terbesar =", f[indeks_maks])
print("Terjadi pada x =", x[indeks_maks])
```

Output:

```
x = [0.  0.5 1.  1.5 2.  2.5 3.  3.5 4.  4.5 5. ]
f(x) =
[ 3.    2.25  2.    2.25  3.    4.25  6.    8.25 11.   14.25 18.  ]
Indeks nilai fungsi terbesar = 10
Nilai fungsi terbesar = 18.0
Terjadi pada x = 5.0
```

Mari kita uraikan hasil tersebut. Perintah

`np.linspace(0, 5, 11)`

menghasilkan 11 titik yang tersebar merata dari 0 sampai dengan 5, yaitu

$$x = [0,\ 0.5,\ 1,\ 1.5,\ 2,\ 2.5,\ 3,\ 3.5,\ 4,\ 4.5,\ 5].$$

Karena ada 11 titik pada interval sepanjang 5, selisih antartitik adalah

$$\Delta x = \frac{5-0}{11-1} = 0.5.$$

Berikutnya kita hitung nilai fungsi

$$\begin{aligned}
f(0) &= 3,\\
f(0.5) &= 0.25 - 1 + 3 = 2.25,\\
f(1) &= 1 - 2 + 3 = 2,\\
f(1.5) &= 2.25 - 3 + 3 = 2.25,\\
f(2) &= 4 - 4 + 3 = 3,\\
f(2.5) &= 6.25 - 5 + 3 = 4.25,\\
f(3) &= 9 - 6 + 3 = 6,\\
f(3.5) &= 12.25 - 7 + 3 = 8.25,\\
f(4) &= 16 - 8 + 3 = 11,\\
f(4.5) &= 20.25 - 9 + 3 = 14.25,\\
f(5) &= 25 - 10 + 3 = 18.
\end{aligned}$$

Jadi larik nilai fungsi adalah

$$f = [3,\ 2.25,\ 2,\ 2.25,\ 3,\ 4.25,\ 6,\ 8.25,\ 11,\ 14.25,\ 18].$$

Nilai terbesar dari larik tersebut adalah 18 dan nilai ini terjadi saat $x = 5$. Karena NumPy menggunakan indeks mulai dari 0, posisi elemen terakhir pada larik f adalah indeks 10.

Serial Python Minimalis: Tutorial #09-Larik 2D dengan NumPy

Ahmad R. T. Nugraha

ver. 21 April 2026

Larik (*array*) NumPy yang kita pelajari pada bagian sebelumnya semuanya merupakan pengganti dari `list` Python biasa. Namun, di dalam Python, kita juga bisa menggunakan `list` di dalam `list` (*lists-of-lists*), sebagai contoh:

```
LL = [[11,12,13,14], [15,16,17,18], [19,20,21,22]]
```

Secara alamiah, NumPy turut mendukung penggunaan fungsi `array()` agar kita dapat menghasilkan sebuah larik yang berisi elemen-elemen dari `LL`:

```
A = np.array(LL)
A
```

Output:

```
array([[11, 12, 13, 14],
       [15, 16, 17, 18],
       [19, 20, 21, 22]])
```

Contoh di atas menunjukkan *interpreter* Python memahami `A` bukanlah sebuah `list` biasa, melainkan sebuah objek larik. Menariknya, entitas `A` ini langsung ditampilkan dalam tiga baris, sehingga memudahkan kita untuk melihatnya sebagai entitas **dua dimensi** (2D), mirip dengan matriks dalam matematika. Penggunaan fungsi `print()` untuk mencetak `A` akan memperjelas status `A`:

```
print(A)
```

Output:

```
[[11 12 13 14]
 [15 16 17 18]
 [19 20 21 22]]
```

Bandingkan dengan cara Python biasa menampilkan sebuah *list of lists*:

```
print(LL)
```

Output:

```
[[11, 12, 13, 14], [15, 16, 17, 18], [19, 20, 21, 22]]
```

Selain memisahkan ketiga baris tersebut ke baris yang berbeda, hasil cetakan dari sebuah array tidak lagi menampilkan tanda koma, sehingga memungkinkan kita untuk lebih fokus pada angka-angkanya saja.

1 Pendefinisian Larik 2D

Banyak hal yang telah kita pelajari mengenai array 1D turut berlaku untuk kasus 2D. Misalnya, penggunaan `type()` untuk memeriksa tipe larik atau tipe elemennya. Kita juga dapat membuat larik secara spesifik dengan tipe yang diinginkan.

```
B = np.array(LL,dtype=np.float64)
print(B)
```

Output:

```
[[11. 12. 13. 14.]
 [15. 16. 17. 18.]
 [19. 20. 21. 22.]]
```

```
B.dtype
```

Output:

```
dtype('float64')
```

Di sini, kita secara eksplisit memastikan bahwa kita sedang membuat larik berisi bilangan pecahan (*floats*). Selain itu, perhatikan bahwa semua larik memiliki atribut bernama `dtype` yang merujuk pada tipe data elemen-elemennya Kita juga pernah menggunakan atribut `size` pada larik 1D untuk melihat ada berapa banyak elemen di dalamnya. Atribut ini bekerja dengan sama baiknya pada kasus 2D:

```
A.size
```

Output:

```
12
```

Perhatikan bahwa atribut ini menghitung jumlah total elemen, yang mencakup baris maupun kolom.

Atribut lain yang belum kita bahas sebelumnya adalah `ndim`, yang memberi tahu kita dimensi dari larik NumPy tersebut:

```
A.ndim
```

Output:

```
2
```

Jumlah dimensi dapat dibayangkan sebagai jumlah “sumbu” (*axes*) berbeda yang kita gunakan untuk menyusun elemen. Terminologi ini mungkin agak membingungkan bagi mereka yang memiliki latar belakang aljabar linear. Dalam aljabar linear, kita mengatakan bahwa sebuah matriks A dengan m baris dan n kolom memiliki "dimensi" m dan n, atau terkadang disebut berdimensi $m \times n$. Berbeda dengan hal tersebut, konvensi di dalam NumPy adalah menyebut array seperti `A` memiliki “dimensi dua”, karena larik tersebut tersusun dari baris dan kolom (yaitu, tidak masalah ada berapa banyak elemen spesifik di masing-masing baris maupun kolomnya). Dalam contoh `A` kita di atas, dimensinya adalah 2, meskipun di sana terdapat 3 baris dan 4 kolom.

Pertanyaan logis selanjutnya adalah apakah kita bisa mengakses jumlah baris dan kolomnya secara khusus (ingat: atribut `size` memuat hasil kali dari keduanya). Jawabannya, tentu saja, ya. Terdapat atribut lain, yaitu `shape`, yang mengembalikan sebuah *tuple* berisi jumlah elemen pada setiap dimensinya. Sebagai contoh:

```
A.shape
```

Output:

```
(3, 4)
```

Perhatikan bahwa $3 \times 4 = 12$. Seperti yang seharusnya segera kita periksa sendiri, sebuah larik 1D akan mengembalikan nilai 1 untuk `ndim`. Demikian pula, atribut `shape`-nya akan berupa sebuah *tuple* dengan satu elemen, misalnya `(7,)`, yang angkanya untuk kasus 1D itu bernilai sama dengan ukuran total larik. Atribut yang akan paling banyak kita gunakan pada pembahasan selanjutnya adalah `shape`.

Sejauh ini kita baru melihat satu cara untuk membuat larik NumPy 2D, yaitu dengan meneruskan *list of lists* ke fungsi `array()`. Namun, pada dasarnya semua fungsi yang sebelumnya kita gunakan untuk larik 1D juga berfungsi untuk larik 2D. Misalnya, kita bisa menggunakan `zeros()` untuk membuat larik 2D yang penuh dengan angka nol. Alih-alih meneruskan satu angka, kali ini kita akan meneruskan sebuah *tuple*, yang merepresentasikan bentuk (*shape*) dari larik baru yang diinginkan:

```
np.zeros((2,3))
```

Output:

```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

Perlu dicermati baik-baik, kita menggunakan sepasang tanda kurung biasa untuk menandakan fungsi `zeros()` dan sepasang tanda kurung biasa tambahan untuk *tuple* yang memuat bentuk *shape*-nya.

Dengan cara serupa, kita bisa menggunakan `ones()` untuk membuat larik 2D yang penuh dengan angka satu. Kali ini, mari kita terapkan untuk membuat matriks persegi:

```
np.ones((3,3))
```

Output:

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]])
```

Sekali lagi, kita meneruskan sebuah *tuple* untuk merepresentasikan `shape`.

Dalam aljabar linear, terdapat sebuah matriks yang sangat penting yang terdiri dari angka nol dan satu dalam kombinasi khusus, yakni matriks identitas (*identity*), yang menurut definisinya merupakan matriks persegi. Fungsi NumPy bernama `identity()` dapat menghasilkan tepat apa yang kita inginkan ini:

```
np.identity(3)
```

Output:

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

Kita bisa sadari bahwa `identity()` hanya mengambil satu argumen tunggal, karena fungsi tersebut memang dirancang untuk menghasilkan matriks persegi. Serupa dengan itu, fungsi-fungsi seperti `zeros()` atau `ones()` sebenarnya bisa menghasilkan larik tiga dimensi, empat dimensi, dan seterusnya, meskipun kita belum akan membutuhkannya. Perhatikan juga bahwa secara bawaan, semua fungsi ini menghasilkan bilangan pecahan (*floats*).

Mari kita kembali sejenak pada kasus operan parameter `shape` saat mencoba membuat sebuah fungsi larik baru. Kita mungkin menyadari bahwa pendefinisian sebuah *list of lists* di Python, seperti pada variabel `LL` kita di atas, agak "kaku dan merepotkan" karena kita memerlukan kurung siku ganda serta beberapa tanda koma yang disisipkan di tempat-tempat yang tepat. Faktanya, seperti yang akan kita lihat di bagian berikutnya, upaya menghindari notasi yang tidak praktis ini merupakan salah satu dari sekian banyak manfaat menggunakan larik NumPy 2D.

Pada titik ini, tidak mengherankan lagi jika NumPy memberi kita pilihan untuk menghindari penggunaan *list of lists* sejak awal. Caranya adalah dengan memulai dari array 1D, yang kemudian dibentuk ulang (*reshaped*) menjadi array 2D melalui fungsi `reshape()`:

```
B = np.array([1,4,9,7,2,8,5,6])
B.shape
```

Output:

```
(8,)
```

```
B = B.reshape((2,4))
B.shape
```

Output:

```
(2, 4)
```

```
B
```

Output:

```
array([[1, 4, 9, 7],
       [2, 8, 5, 6]])
```

Awalnya, `B` adalah sebuah larik 1D (yang kita buat menggunakan `list` Python biasa), terlihat jelas dari nilai pada atribut `shape`-nya. Kita kemudian menerapkan metode `reshape()` pada larik 1D tersebut, sehingga menciptakan sebuah larik 2D (yang disimpan kembali ke dalam variabel `B`). Sekali lagi, nilai atribut `shape` menunjukkan bahwa kondisinya sekarang telah berubah.

Perhatikan bahwa kita bahkan tidak perlu memulainya dari `list` Python. Sebaliknya, contoh `A` yang kita diskusikan di atas dapat ditulis ulang menggunakan `arange()` yang menghasilkan sebuah larik 1D:

```
A = np.arange(11,23)
A = A.reshape((3,4))
A
```

Output:

```
array([[11, 12, 13, 14],
       [15, 16, 17, 18],
       [19, 20, 21, 22]])
```

Larik `A` ini bermula sebagai larik 1D dan kemudian dibentuk ulang menjadi larik 2D dengan meneruskan argumen `shape` yang sesuai. Setelah kita semakin terbiasa dengan larik, kita akan merasa lebih nyaman menggabungkan kedua langkah tersebut di dalam satu baris kode:

```
A = np.arange(11,23).reshape(3,4)
A
```

Output:

```
array([[11, 12, 13, 14],
       [15, 16, 17, 18],
       [19, 20, 21, 22]])
```

Perhatikan betapa mudahnya kita menghindari rentetan tanda kurung siku dan koma yang sebelumnya diperlukan saat mendefinisikan `LL`. Sebagai catatan tambahan, kita sebenarnya bisa membentuk ulang larik cukup dengan mengubah nilai *tuple* pada atribut `shape`, alih-alih menggunakan fungsi `reshape()`).

2 Pengindeksan dan Pemotongan

Kini saatnya untuk melihat salah satu fitur paling menyenangkan dari penggunaan larik 2D di NumPy yang ini kebetulan menjadi nilai jual utamanya, yaitu cara yang intuitif dalam mengakses elemen. Mari kita mulai dari contoh *list of lists* kita:

```
LL = [[11,12,13,14], [15,16,17,18], [19,20,21,22]]
```

Seperti yang mungkin kita ingat, jika kita tertarik pada elemen yang terletak di baris ke-2 dan kolom ke-1, kita akan mengeksekusi:

```
LL[2][1]
```

Output:

```
20
```

Ingatlah bahwa kita menggunakan indeks yang dimulai dari angka 0 ala Python, sehingga baris-barisnya diberi nomor ke-0, ke-1, ke-2, dan analogi yang sama berlaku untuk kolomnya.

Dalam percakapan sehari-hari, contoh di atas merujuk pada baris ketiga dan kolom kedua, tetapi kita akan menggunakan angka, misalnya ke-2, saat menggunakan konvensi indeks mulai 0. Pasangan tanda kurung ganda yang memisahkan angka-angka ini satu sama lain cukup merepotkan dan agak berbeda dari bagaimana notasi matriks umumnya ditulis, misalnya A_{ij} atau $A_{i,j}$. Oleh karena itu, sangat melegakan melihat bahwa elemen array NumPy dapat diindeks cukup dengan menggunakan sepasang tanda kurung siku saja beserta dua angka yang dipisahkan oleh tanda koma, contohnya:

```
A = np.arange(11,23).reshape(3,4)
A
```

Output:

```
array([[11, 12, 13, 14],
       [15, 16, 17, 18],
       [19, 20, 21, 22]])
```

```
A[2,1]
```

Output:

```
20
```

Faktanya, notasi `list` Python yang merepotkan tersebut tetap bekerja pada larik NumPy, namun kita tidak akan menggunakannya dan memilih alternatif yang lebih "bersih" yaitu dua nilai yang dipisahkan koma. Demikian pula, meskipun kita dapat membuat array 2D menggunakan *list of lists*, kita umumnya akan lebih memilih menggunakan array 1D yang diikuti dengan pemanggilan fungsi `reshape()`.

Sekarang kita beralih ke *slicing* (pemotongan). Kita mulai dari kasus yang paling sederhana, yaitu memilih baris tertentu (melalui pengindeksan) dan kemudian mengambil irisan dari beberapa kolom berbeda. Contohnya:

```
A[1,1:3]
```

Output:

```
array([16, 17])
```

Demikian pula, jika kita secara spesifik memilih sebuah baris dan menggunakan irisan segala (*everything-slice*) untuk kolomnya, kita akan mendapatkan baris tersebut seutuhnya:

```
A[1,:]
```

Output:

```
array([15, 16, 17, 18])
```

Menggunakan logika yang sama persis, jika kita memilih kolom tertentu dan menggunakan *everything-slice* untuk barisnya, kita akan mendapatkan kolom tersebut seutuhnya:

```
A[:,2]
```

Output:

```
array([13, 17, 21])
```

Kita bisa terus menumpuk contoh-contoh seperti ini dengan memilih sepasang baris, sepasang kolom, dan lain sebagainya.

Sebagai ringkasan, saat kita menggunakan dua angka untuk melakukan indeks (seperti `A[2,1]`), kita bergerak dari sebuah larik 2D menjadi satu buah angka tunggal. Saat kita menggunakan satu angka untuk mengindeks/*slice* (seperti `A[:,2]`), kita bergerak dari larik 2D menjadi larik 1D. Saat kita menggunakan *slice* penuh (seperti `A[::2,:]`), kita akan mendapatkan himpunan kumpulan baris, kolom, elemen individual, dsb.

Kita dapat menggabungkan apa yang baru saja dipelajari mengenai *slicing* larik 2D dengan apa yang sudah kita pahami tentang *broadcasting* di NumPy. Pertama, pilih subhimpunan (*subset*) tertentu dari larik kita dan tugaskan ke variabel baru:

```
B = A[:2,1:]
B
```

Output:

```
array([[12, 13, 14],
       [16, 17, 18]])
```

Manfaatkan *broadcasting* nilai tunggal maupun *everything-slice* (dua kali berturut-turut) untuk mengatur seluruh elemen dari entitas baru ini menjadi angka nol:

```
B[:,:]  = 0
B
```

Output:

```
array([[0, 0, 0],
       [0, 0, 0]])
```

Pada titik ini, kita perlu mengingat bahwa *everything-slices* di NumPy (walaupun memudahkan kita memasukkan nilai ke setiap elemen secara bersamaan seperti di atas) tidak akan menciptakan salinan. Seperti yang telah kita lihat pada larik 1D, irisan larik merupakan tampilan (*views*) ke larik aslinya. Akibatnya, dengan memodifikasi `B`, kita secara riil juga telah memodifikasi `A`:

```
A
```

Output:

```
array([[11,  0,  0,  0],
       [15,  0,  0,  0],
       [19, 20, 21, 22]])
```

Seperti sebelumnya, jika perilaku seperti ini tidak diinginkan, kita harus menggunakan `numpy.copy()`:

```
A = np.arange(11,23).reshape(3,4)
A
```

Output:

```
array([[11, 12, 13, 14],
       [15, 16, 17, 18],
       [19, 20, 21, 22]])
```

```
B = np.copy(A[:2,1:])
B[:,:]  = 0
B
```

Output:

```
array([[0, 0, 0],
       [0, 0, 0]])
```

```
A
```

Output:

```
array([[11, 12, 13, 14],
       [15, 16, 17, 18],
       [19, 20, 21, 22]])
```

3 Vektorisasi

Sama seperti larik 1D, operasi untuk larik 2D ditafsirkan elemen demi elemen (*elementwise*). Operasi yang tervektorisasi semacam ini memungkinkan kita menulis kode yang sangat rapi dan ringkas, contohnya untuk menjumlahkan dua matriks persegi:

```
A = np.arange(1,10).reshape(3,3)
B = np.arange(11,20).reshape(3,3)
A
```

Output:

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
B
```

Output:

```
array([[11, 12, 13],
       [14, 15, 16],
       [17, 18, 19]])
```

```
A+B
```

Output:

```
array([[12, 14, 16],
       [18, 20, 22],
```

```
       [24, 26, 28]])
```

Hasil ini merupakan perilaku yang memudahkan dan diharapkan.

Di sisi lain, operasi perkalian sederhana juga dijalankan secara elemen demi elemen, yang artinya setiap elemen di `A` dikalikan dengan elemen yang bersesuaian dengan posisinya di `B`:

```
A*B
```

Output:

```
array([[ 11,  24,  39],
       [ 56,  75,  96],
       [119, 144, 171]])
```

Hati-hati hasil ini bisa menjadi perilaku tak terduga jika kita sebenarnya sedang bermaksud melakukan perkalian matriks. Sekali lagi, operasi larik selalu dijalankan elemen demi elemen, sehingga produk hasil kali antara dua buah larik dua dimensi hanyalah $(\mathbf{A} \odot \mathbf{B})_{ij} = A_{ij}B_{ij}$. Sebagai informasi, di dalam ilmu matematika, operasi ini dikenal sebagai hasil kali Hadamard. Operasi perkalian matriks yang kita pelajari dari aljabar linear mengikuti rumus yang berbeda, yakni $(\mathbf{AB})_{ij} = \sum_k A_{ik}B_{kj}$. Operasi ini secara rapi sudah ada di dalam fungsi `dot()` milik NumPy:

```
np.dot(A,B)
```

Output:

```
array([[ 90,  96, 102],
       [216, 231, 246],
       [342, 366, 390]])
```

Hasilnya sesuai yang diharapkan untuk suatu perkalian matriks biasa, tetapi sintaksisnya mungkin terasa kurang intuitif. Untungnya, sejak Python 3.5 dan seterusnya kita dapat mengalikan dua buah matriks menggunakan operator *infix* `@`, yaitu:

```
A@B
```

Output:

```
array([[ 90,  96, 102],
       [216, 231, 246],
       [342, 366, 390]])
```

yang jelas jauh lebih enak digunakan. Uniknya, baik `@` maupun `numpy.dot()` mampu menangani larik 1D.

NumPy masih memiliki beberapa fungsi penting lain, dengan nama yang mendeskripsikan kegunaannya sendiri secara gamblang. Yang paling umum digunakan adalah `transpose()`:

```
np.transpose(A)
```

Output:

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

yang juga dapat dipanggil sebagai atribut dari larik:

```
A.T
```

Output:

```
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

Fungsi lain yang sangat berguna adalah `trace()`:

```
np.trace(A)
```

Output:

```
15
```

Seperti yang mungkin sudah kita tebak, fungsi terakhir ini akan menjumlahkan elemen-elemen yang berada pada diagonal utama matriks.

Berdasarkan hal-hal yang telah kita perkenalkan, kita dapat melihat betapa mudahnya mengiterasi baris-baris pada sebuah larik:

```
A = np.arange(1,10).reshape(3,3)
A
```

Output:

```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

```
for row in A:
    print(row)
```

Output:

```
[1 2 3]
[4 5 6]
[7 8 9]
```

maupun kolom-kolomnya:

```
for column in A.T:
    print(column)
```

Output:

```
[1 4 7]
[2 5 8]
[3 6 9]
```

Perhatikan bahwa dalam kedua kasus iterasi di atas kita tidak melihat adanya kurung siku ganda, karena kita mencetak sebuah larik 1D di setiap iterasinya. Iterasi melalui kolom akan sangat berguna terutama ketika kita menangani nilai *eigen* (dan *eigenvectors*) dari sebuah matriks.

Secara umum, larik 2D memiliki perilaku yang sangat mirip dengan larik 1D. Artinya, saat kita menggabungkan larik 2D dengan suatu nilai skalar, proses *broadcasting* akan terjadi, contohnya:

```
A/2
```

Output:

```
array([[0.5, 1. , 1.5],
       [2. , 2.5, 3. ],
       [3.5, 4. , 4.5]])
```

Broadcasting sering kali dimanfaatkan dalam skenario yang lebih kompleks (misalnya, untuk menggabungkan array dua dimensi dengan array satu dimensi), tetapi kita tidak akan menggunakan fungsionalitas semacam itu. Tentu saja, semua fungsi NumPy yang pernah kita singgung sebelumnya, seperti `numpy.sqrt()`, `numpy.exp()`, juga bisa dijalankan pada larik 2D, dan prinsip yang sama berlaku untuk proses pengurangan, perkalian, perpangkatan, dsb.

Menariknya, fungsi-fungsi seperti `numpy.sum()`, `numpy.amin()`, dan `numpy.amax()` masih mau menerima argumen (opsional) berupa `axis`. Jika kita menetapkan `axis = 0`, operasi akan berjalan di sepanjang baris, sementara untuk `axis = 1` beroperasi di sepanjang kolom. Sebagai contoh:

```
A = np.arange(1,10).reshape(3,3)
np.sum(A,axis=0)
```

Output:

```
array([12, 15, 18])
```

Serupa dengan itu, kita dapat menyusun fungsi-fungsi bebas buatan kita sendiri yang dirancang untuk sanggup menangani matriks 2D seutuhnya secara bersamaan. Mengingat fitur-fitur ini sangat identik dengan apa yang kita temui pada bagian

sebelumnya, dapat diasumsikan konsepnya sudah jelas bagi pembaca. Meskipun begitu, memang ada beberapa fungsi praktis yang secara intuitif lebih mudah dicerna penggunaannya untuk larik 2D seperti `numpy.diag()` yang digunakan untuk mendapatkan elemen-elemen dari diagonal, atau `numpy.tril()` untuk memperoleh segitiga bagian bawah (*lower triangle*) dari sebuah array serta `numpy.triu()` untuk segitiga bagian atas (*upper triangle*).

Terakhir, sebuah fungsi berguna lainnya adalah `numpy.eye()` yang wujudnya sangat ekuivalen dengan `numpy.identity()`. Namun, yang membuatnya lebih praktis, fungsi ini mengizinkan kita mengisi elemen pada diagonal yang letaknya ada di atas atau di bawah diagonal utama. Sebagai contoh, cobalah eksekusi `numpy.eye(5, k=-2)`. Kita dapat menelaah kembali dokumentasi yang ada untuk menggali lebih lanjut rupa-rupa fungsionalitas ini.

Satu poin tambahan, format penyimpanan bawaan untuk larik NumPy di dalam memori komputer adalah *row-major* (mengutamakan baris). Sebagai seorang pemula, kita tidak perlu terlalu merisaukan perihal ini, tetapi artinya adalah baris-baris akan disimpan secara berurutan, satu demi satu. Ini adalah format penyimpanan serupa yang diadopsi oleh bahasa pemrograman C. Jika sebaliknya kita ingin menggunakan format *column-major* (mengutamakan kolom) ala Fortran, mungkin demi tujuan bisa berinteraksi dengan kode dari bahasa tersebut, kita hanya perlu meneruskan argumen *keyword* yang sesuai saat membuat larik NumPy kita. Selama kita menyusun kode kita dalam cara yang "natural", yakni melakukan iterasi melalui baris terlebih dahulu dan barulah melalui kolom, segalanya akan berjalan normal dan mulus, sehingga kita sebenarnya dapat saja mengabaikan paragraf ini.

Studi Kasus

Kasus 1

Buatlah sebuah program Python dengan NumPy untuk melakukan operasi berikut pada sebuah matriks A berukuran 2×3:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}.$$

Lakukan langkah-langkah berikut:

1. Tampilkan *shape* dari matriks A.
2. Bentuk *transpose* dari A, yaitu A^T.
3. Hitung hasil perkalian matriks $B = AA^T$.

Solusi

Kita mulai dengan membentuk matriks A sebagai larik NumPy 2D. Setelah itu, *shape* matriks dapat diperoleh melalui atribut `shape`. Transpose matriks diperoleh dengan `A.T`. Untuk perkalian matriks, kita gunakan operator `@`, karena yang diminta adalah perkalian matriks, bukan perkalian elemen per elemen.

```
import numpy as np

# Membentuk matriks A
A = np.array([[1, 2, 3],
              [4, 5, 6]])

# Shape matriks
shape_A = A.shape

# Transpose matriks
AT = A.T

# Perkalian matriks A dengan transpose-nya
B = A @ AT

# Menampilkan hasil
print("A =")
print(A)
print("shape(A) =", shape_A)
print("A^T =")
print(AT)
print("B = A @ A^T =")
print(B)
```

Output:

```
A =
[[1 2 3]
 [4 5 6]]
shape(A) = (2, 3)
A^T =
[[1 4]
 [2 5]
 [3 6]]
B = A @ A^T =
[[14 32]
 [32 77]]
```

Sekarang kita jelaskan hasilnya. Matriks

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

memiliki 2 baris dan 3 kolom, sehingga

$$\text{shape}(A) = (2, 3).$$

Transpose matriks diperoleh dengan menukar baris menjadi kolom, sehingga

$$A^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

Selanjutnya kita hitung

$$B = AA^T.$$

Karena A berukuran 2×3 dan A^T berukuran 3×2, hasil perkalian akan berukuran 2×2. Perhitungannya adalah

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}.$$

Elemen-elemen matriks hasil diperoleh dari hasil kali titik baris dengan kolom:

$$B_{11} = 1 \cdot 1 + 2 \cdot 2 + 3 \cdot 3 = 14,$$

$$B_{12} = 1 \cdot 4 + 2 \cdot 5 + 3 \cdot 6 = 32,$$

$$B_{21} = 4 \cdot 1 + 5 \cdot 2 + 6 \cdot 3 = 32,$$

$$B_{22} = 4 \cdot 4 + 5 \cdot 5 + 6 \cdot 6 = 77.$$

Jadi,

$$B = \begin{bmatrix} 14 & 32 \\ 32 & 77 \end{bmatrix}.$$

Kasus 2

Buatlah sebuah program Python dengan NumPy untuk mempelajari indexing, slicing, dan operasi sepanjang sumbu tertentu pada sebuah matriks. Diberikan

matriks

$$M = \begin{bmatrix} 2 & 4 & 6 \\ 1 & 3 & 5 \\ 7 & 8 & 9 \end{bmatrix}.$$

Lakukan langkah-langkah berikut: (1) Ambil baris kedua dari matriks M, (2) Ambil kolom ketiga dari matriks M, (3) Ambil submatriks yang terdiri atas dua baris pertama dan dua kolom pertama, (4) Hitung jumlah setiap baris, dan (5) Hitung jumlah setiap kolom.

Solusi

Pada soal ini, kita akan menggunakan *indexing* dan *slicing* pada larik 2D. Dalam NumPy, elemen diakses dengan format `[baris, kolom]`. Untuk mengambil satu baris atau satu kolom, kita dapat menggunakan *slicing* yang sesuai. Penjumlahan sepanjang baris atau kolom dilakukan dengan fungsi `np.sum()` menggunakan parameter `axis`.

```python
import numpy as np

# Membentuk matriks M
M = np.array([[2, 4, 6],
              [1, 3, 5],
              [7, 8, 9]])

# Mengambil bagian-bagian matriks
baris_kedua = M[1, :]
kolom_ketiga = M[:, 2]
submatriks = M[:2, :2]

# Menjumlahkan elemen per baris dan per kolom
jumlah_baris = np.sum(M, axis=1)
jumlah_kolom = np.sum(M, axis=0)

# Menampilkan hasil
print("M =")
print(M)
print("Baris kedua =", baris_kedua)
print("Kolom ketiga =", kolom_ketiga)
print("Submatriks dua baris pertama dan dua kolom pertama =")
print(submatriks)
print("Jumlah setiap baris =", jumlah_baris)
print("Jumlah setiap kolom =", jumlah_kolom)
```

Output:

```
M =
[[2 4 6]
 [1 3 5]
 [7 8 9]]
Baris kedua = [1 3 5]
Kolom ketiga = [6 5 9]
Submatriks dua baris pertama dan dua kolom pertama =
[[2 4]
 [1 3]]
Jumlah setiap baris = [12  9 24]
Jumlah setiap kolom = [10 15 20]
```

Sekarang kita bahas satu per satu. Matriks yang diberikan adalah

$$M = \begin{bmatrix} 2 & 4 & 6 \\ 1 & 3 & 5 \\ 7 & 8 & 9 \end{bmatrix}.$$

Baris kedua diambil dengan

`M[1, :]`

karena indeks Python dimulai dari 0. Jadi, baris kedua adalah

$$[1, 3, 5].$$

Kolom ketiga diambil dengan

`M[:, 2]`,

artinya semua baris diambil, tetapi hanya kolom dengan indeks 2. Hasilnya adalah

$$[6, 5, 9].$$

Submatriks yang terdiri atas dua baris pertama dan dua kolom pertama diambil dengan

`M[:2, :2]`,

sehingga diperoleh

$$\begin{bmatrix} 2 & 4 \\ 1 & 3 \end{bmatrix}.$$

Selanjutnya, jumlah setiap baris dihitung dengan

`np.sum(M, axis=1)`.

Artinya, penjumlahan dilakukan sepanjang kolom-kolom pada setiap baris. Hasilnya:

$$2 + 4 + 6 = 12,$$

$$1 + 3 + 5 = 9,$$

$$7 + 8 + 9 = 24.$$

Jadi, kita peroleh jumlah setiap baris adalah:

$$[12, 9, 24].$$

Jumlah setiap kolom dihitung dengan

```
np.sum(M, axis=0).
```

Artinya, penjumlahan dilakukan sepanjang baris-baris pada setiap kolom. Hasilnya:

$$2 + 1 + 7 = 10,$$

$$4 + 3 + 8 = 15,$$

$$6 + 5 + 9 = 20.$$

Jadi, kita peroleh jumlah setiap kolom adalah:

$$[10, 15, 20].$$

Serial Python Minimalis:
Tutorial #10-Plot Grafik

Ahmad R. T. Nugraha

ver. 21 April 2026

Matplotlib adalah pustaka (*library*) pembuatan/*plotting* grafik yang paling populer untuk Python. Pustaka ini dapat digunakan untuk menghasilkan visualisasi data secara cepat. Dengan ketelitian, grafik berkualitas tinggi dan siap publikasi dalam berbagai format pun dapat dihasilkan. Sesi kali ini difokuskan pada fungsionalitas `pyplot` dari Matplotlib, yang digunakan untuk membuat dan menganotasi (menulis berbagai label) pada grafik ilmiah menggunakan antarmuka prosedural yang sederhana. `pyplot` perlu diimpor, dan biasanya diberi alias menggunakan pengenal yang disingkat, lazimnya adalah `plt`:

```
import matplotlib.pyplot as plt
```

1 Plot Garis

Penggunaan sederhana untuk memplot dan memberi label pada beberapa titik data adalah dengan memanggil `plt.plot` bersama dua *list* ataupun *array*/larik (sangat disarankan langsung menggunakan larik NumPy) berisi angka sebagai berikut:

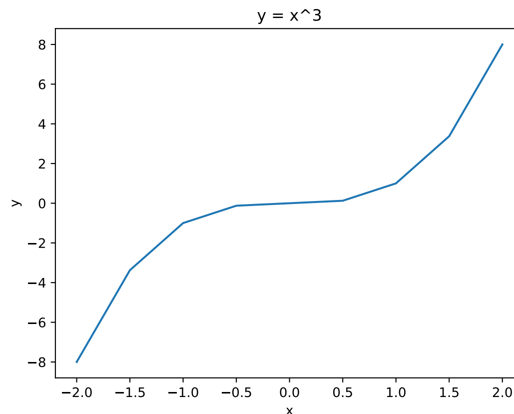
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([-2, -1.5, -1, -0.5, 0, 0.5, 1, 1.5, 2])
y = x**3

plt.plot(x, y)
```

Di dalam skrip Python atau sesi Python interaktif, biasanya tidak akan ada yang muncul di layar sampai kita memanggil `plt.show()`. Jadi, sebelum memanggil `plt.show()`, berbagai anotasi dapat ditambahkan, seperti label untuk sumbu dan sebuah judul:

```
plt.xlabel('x')
plt.ylabel('y')
```

```
plt.title('y = x^3')
plt.show()
```

Sementara itu, dalam Jupyter Notebook, grafik akan langsung ditampilkan segera setelah sel yang berisi pernyataan `plt.plot()` dieksekusi, sehingga kelengkapan anotasi di atas harus berada di dalam sel yang sama dengan `plt.plot()`. Hasil eksekusi perintah-perintah dasar Matplotlib tersebut ditunjukkan pada Gambar 1.



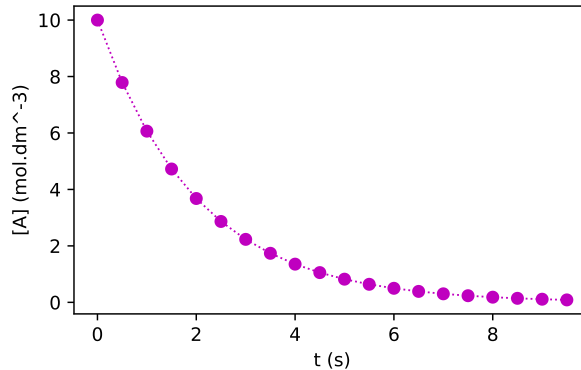
Gambar 1: Pemanfaatan Matplotlib untuk plot garis $y = x^3$ dengan menggunakan sembilan titik.

Walau tampaknya "cukup" untuk keperluan melihat bentuk grafik tertentu, tampilan pada Gambar 1 masih **belum memenuhi standar kelaziman grafik ilmiah** untuk publikasi. Secara bertahap, kita akan belajar kustomisasi dan pendetailan grafik ilmiah dengan Matplotlib. Sebagai awalan, grafik dapat dikustomisasi dengan penanda (`marker`), warna (*color*, `c`), ketebalan (*linewidth*, `lw`), serta gaya garis (*linestyle*, `ls`) menggunakan argumen-argumen tambahan, sebagaimana diperinci pada Tabel 1– 3. Sebagai contoh, kita buat gambar baru untuk plot grafik konsentrasi suatu materi, $[A] = [A]_0 e^{-kt}$, yang ditunjukkan pada Gambar 2.

```
import numpy as np
import matplotlib.pyplot as plt

A0, k = 10, 0.5
t = np.arange(0, 10.1, 0.5)
A = A0 * np.exp(-k * t)

plt.figure(figsize=(5,3))
plt.plot(t, A, c='m', lw=1, ls=':', marker='o')
plt.xlabel('t (s)')
plt.ylabel('[A] (mol.dm^-3)')
```

```
plt.show()
```

Perhatikan bahwa kita dapat memanfaatkan fungsi `plt.figure()` dengan argumen `figsize=(width,height)` untuk spesifikasi lebih terperinci terkait ukuran gambar. Parameter `width` dan `height` dinyatakan dalam satuan inci.



Gambar 2: Plot $[A] = [A]_0 e^{-kt}$.

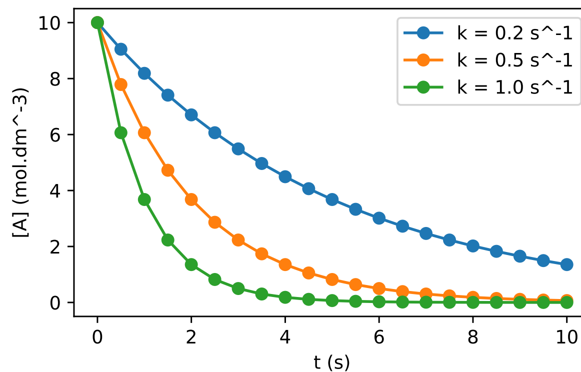
Tabel 1: Kode warna Matplotlib. Ada versi kode warna dasar satu huruf yang dapat dispesifikasi sebagai argumen atau parameter saat pemanggilan `plt.plot()`, serta ada siklus urutan warna bawaan palet "Tableau 10".

Kode warna dasar	Warna Tableau
`b` = biru (*blue*)	`tab:blue`
`g` = hijau (*green*)	`tab:orange`
`r` = merah (*red*)	`tab:green`
`c` = sian (*cyan*)	`tab:red`
`m` = magenta (*magenta*)	`tab:purple`
`y` = kuning (*yellow*)	`tab:brown`
`k` = hitam (*black*)	`tab:pink`
`w` = putih (*white*)	`tab:gray`
	`tab:olive`
	`tab:cyan`

Pemanggilan `plt.plot` secara berulang dengan data yang berbeda akan menambahkan garis-garis baru ke dalam grafik. Dengan adanya beberapa garis yang diplot, ada baiknya kita memberikan perbedaan melalui `string` yang sesuai ke argumen `label`. Dalam kasus seperti Gambar 2, kita dapat membuat beberapa garis dengan nilai k yang berbeda. Sebagai pembedanya, `plt.legend()` harus dipanggil sebelum menampilkan grafik-grafik tersebut.

Tabel 2: Beberapa gaya penanda (*marker*) Matplotlib dengan kode string satu karakter.

Kode	Penanda	Deskripsi
`.`	·	Titik (*Point*)
`o`	○	Lingkaran (*Circle*)
`+`	+	Plus (*Plus*)
`x`	×	Silang (*Cross*)
`D`	◇	Wajik (*Diamond*)
`v`	▽	Segitiga menghadap bawah (*Downward triangle*)
`^`	△	Segitiga menghadap atas (*Upward triangle*)
`s`	□	Persegi (*Square*)
`*`	⋆	Bintang (*Star*)



Gambar 3: Plot $[A] = [A]_0 e^{-kt}$ untuk konstanta laju k yang berbeda-beda.

```
import numpy as np
import matplotlib.pyplot as plt

A0 = 10
t = np.arange(0, 10.1, 0.5)
rate_constants = [0.2, 0.5, 1.0]

plt.figure(figsize=(5,3))

for k in rate_constants:
    A = A0 * np.exp(-k * t)
    plt.plot(t, A, marker='o', label=f'k = {k:.1f} s^-1')

plt.xlabel('t (s)')
plt.ylabel('[A] (mol.dm^-3)')
plt.legend()
```

```
plt.show()
```

Hasilnya ditunjukkan pada Gambar 3.

Perhatikan bahwa pada dasarnya kita tidak perlu menentukan warna baru untuk setiap pemanggilan `plt.plot` karena Matplotlib dapat memutar (*cycle*) urutan warna bawaan yang telah ditentukan sebelumnya untuk membedakan garis-garis yang diplot. Namun, untuk tampilan yang lebih baik dalam grafik ilmiah kita tentunya dapat spesifikasi ulang warna-warna setiap garis.

Tabel 3: Properti garis dan penanda Matplotlib.

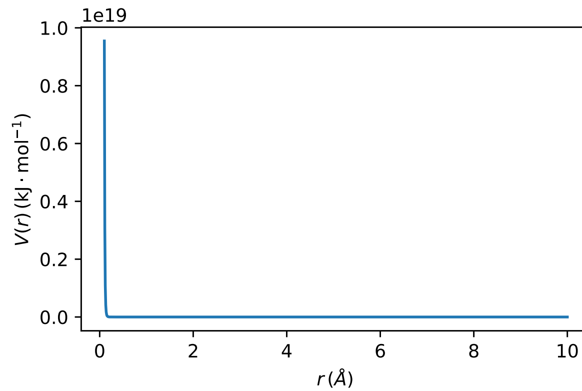
Argumen	Singkatan	Deskripsi
`color`	`c`	Warna garis
`alpha`		Opasitas garis: 0 (sepenuhnya transparan) dan 1 (sepenuhnya buram)
`linestyle`	`ls`	Gaya garis: `'solid'`, `'dotted'`, `'dashed'`, `'dashdot'`; atau disingkat `'-'`, `':'`, `'-'`, `'-.'`
`linewidth`	`lw`	Ketebalan garis dalam poin, nilai bawaannya adalah 1.5 pt.
`markersize`	`ms`	Ukuran penanda, dalam poin
`markevery`		Ditetapkan ke bilangan bulat positif, N, untuk mencetak penanda setiap N titik. Nilai bawaannya, yaitu `None`, mencetak penanda untuk setiap titik
`markerfacecolor`	`mfc`	Warna isian penanda
`markeredgecolor`	`mec`	Warna tepi penanda
`markeredgewidth`	`mew`	Ketebalan tepi penanda, dalam poin

Secara bawaan, Matplotlib memilih batas-batas untuk sumbu x dan y sedemikian rupa sehingga dapat menampilkan seluruh data yang diplot. Untuk menetapkan batas-batas ini secara manual, gunakan `plt.xlim` dan `plt.ylim`. Sebagai contoh, mari kita tinjau plot potensial interatomik Lennard-Jones untuk argon,

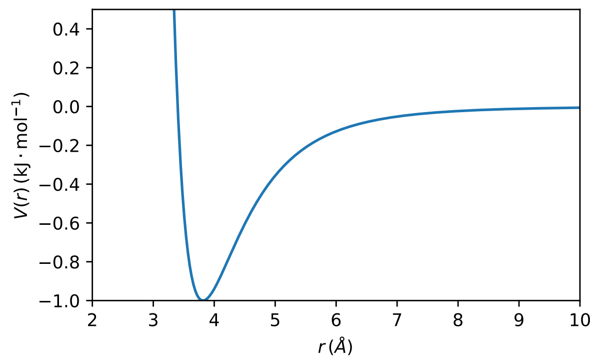
$$V(r) = 4\epsilon \left[\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right]$$

dengan r adalah jarak interatomik, $\epsilon = 3.4$ kJ mol^{-1} adalah kedalaman sumur potensial, dan $\sigma = 3.4$ Å adalah jarak pisah yang potensialnya bernilai nol: $V(\sigma) = 0$. Plot $V(r)$ begitu saja pada suatu kisi (*grid*) berisi nilai-nilai r melalui kode berikut ini tidak akan menghasilkan grafik yang terlalu memuaskan:

```
import numpy as np
import matplotlib.pyplot as plt

rmax = 10
```



Gambar 4: Plot potensial Lennard-Jones untuk argon tanpa menetapkan batas-batas sumbu horizontal maupun vertikal.



Gambar 5: Plot potensial Lennard-Jones untuk argon dengan batas-batas plot yang telah ditetapkan secara lebih baik.

```
r = np.linspace(0.1, rmax, 1000)

# Parameter Lennard-Jones untuk Ar:
# kedalaman sumur (kJ/mol) dan
# jarak pisah yang potensial interatomiknya nol
E, sigma = 1.0, 3.4

def LJ(r, E, sigma):
    fac = (sigma / r)**6
    return 4 * E * (fac**2 - fac)

V = LJ(r, E, sigma)
plt.plot(r, V)
plt.xlabel(r'$r\,(\AA$)')
```

```
plt.ylabel(r'$V(r)\,(\mathrm{kJ} \cdot \mathrm{mol}^{-1})$')
plt.title('Potensial Lennard-Jones untuk Argon')
plt.show()
```

Hasilnya ditunjukkan pada Gambar 4. Di sini, dominasi dari suku tolakan pada potensial (r^{-12}) berujung pada nilai $V(r)$ yang sangat besar pada r kecil, sehingga area sumur tarik-menarik ($V(r) < 0$) tidak terlihat. Solusinya adalah dengan menetapkan batas-batas plot secara tepat:

```
# Mulai sumbu jarak pada 2 Angstrom.
plt.xlim(2, rmax)
# Batasi rentang y antara dasar sumur dan setengah dari besaran
# kedalaman sumur.
plt.ylim(-E, 0.5 * E)
```

Bagian kode di atas dapat ditambahkan di mana pun setelah `plt.plot()` dan sebelum `plt.show()`. Hasilnya ditunjukkan pada Gambar 5. Pada anotasi `plt.xlabel` maupun `plt.ylabel`, kita dapat memanfaatkan format LaTeX dengan menambahkan karakter `r` sebelum tanda petik, kemudian di dalam tanda petik kita perlu mengapit format LaTeX dengan tanda dolar.

2 Plot Sebaran (*Scatter Plots*)

Terdapat beberapa jenis grafik umum lainnya yang disediakan oleh Matplotlib, termasuk plot sebaran (*scatter plot*):

```
plt.scatter(x, y, s=None, c=None, marker=None)
```

Baris kode di atas akan membuat plot sebaran dari data (x, y) menggunakan penanda berukuran `s` dan berwarna `c` yang ditentukan. Nilai bawaan akan digunakan jika tidak ada nilai yang diberikan untuk argumen-argumen tersebut. Jika `s` atau `c` berupa urutan (*sequences*), ukuran dan warna dari penandanya dapat diatur secara spesifik untuk masing-masing titik data.

Untuk menganotasi grafik dengan teks, panggil `plt.text`:

```
plt.text(x, y, s)
```

yang akan menempatkan string `s` pada lokasi (x, y) (dalam koordinat data). Argumen lebih lanjut, `ha` (atau `horizontalalignment`) dan `va` (atau `verticalalignment`) mengontrol bagaimana teks tersebut ditambatkan pada lokasi yang spesifik. Nilai yang valid adalah `ha = 'left'`, `'center'` atau `'right'` dan `va = 'top'`, `'center'`, `'baseline'` atau `'bottom'`. Sebagai contoh, untuk memastikan bahwa titik tengah dari label teks tersebut berada tepat di (x, y), gunakan `ha='center'`, `va='center'`.

Studi Kasus

Kasus 1 Titik Leleh dan Titik Didih Unsur

Berkas `elements_TmTb.csv`, yang dapat diunduh dari situs web `https://scipython.com/chem/ghc`, memuat nilai-nilai titik leleh dan titik didih dari beberapa unsur kimia. Buatlah plot sebaran (*scatter plot*) untuk data ini dengan memberikan warna penanda (*markers*) yang berbeda antara unsur logam dan non-logam. Beri anotasi juga pada grafik tersebut untuk menunjukkan unsur karbon dan unsur yang memiliki selisih terbesar antara titik leleh dan titik didihnya. Untuk kasus ini, unsur-unsur non-logam dapat diasumsikan sebagai unsur-unsur yang terangkum dalam `list` berikut:

```
nonmetals = ['H', 'He', 'C', 'N', 'O', 'F', 'Ne', 'S', 'P',
             'Ar', 'Se', 'Cl', 'Kr', 'Br', 'Xe', 'I', 'Rn']
```

Solusi

File yang disediakan berisi kolom-kolom untuk simbol unsur, titik leleh, dan titik didih yang dipisahkan oleh tanda koma. Jika kita mengintip beberapa baris pertamanya (misalnya menggunakan perintah `!head elements_TmTb.csv` di sistem Linux/WSL), formatnya akan terlihat seperti ini:

```
Element, Tm /K, Tb /K
H, 14.0, 20.3
He, 1.8, 4.2
Li, 453.7, 1603.0
Be, 1560.0, 2742.0
B, 2349.0, 4200.0
C, 3800.0, 4300.0
```

Tantangannya adalah kolom pertama terdiri dari karakter string, bukan angka, sementara kita membutuhkan simbol-simbol unsur tersebut. Kita dapat menggunakan metode `genfromtxt` dari NumPy untuk membaca kolom-kolom ini dan memisahkannya ke dalam larik. Namun, kita perlu menetapkan `dtype=None` (untuk memaksa NumPy menyimpulkan tipe datanya sendiri dan tidak sekadar menganggap kolom pertama sebagai bilangan pecahan) dan juga menentukan pengodean string-nya (biasanya baik ASCII atau UTF-8 sama-sama dapat digunakan).

```
import numpy as np
import matplotlib.pyplot as plt

# definisikan list nonmetals sesuai soal
nonmetals = ['H', 'He', 'C', 'N', 'O', 'F', 'Ne', 'S', 'P',
             'Ar', 'Se', 'Cl', 'Kr', 'Br', 'Xe', 'I', 'Rn']

# Baca data dari file CSV
elements, Tm, Tb = np.genfromtxt('elements_TmTb.csv', delimiter=',',
                                 skip_header=1, unpack=True,
                                 dtype=None, encoding='utf8')

print(elements[:5])
print(Tm[:5])
print(Tb[:5])
```

Output:

```
['H' 'He' 'Li' 'Be' 'B']
[  14.      1.8  453.7 1560.   2349. ]
[  20.3     4.2 1603.   2742.   4200. ]
```

Kita sebenarnya bisa memanggil `plt.scatter` dua kali (sekali untuk logam dan sekali untuk non-logam) atau merancang sebuah *list* warna yang akan digunakan. Jika kita memilih pendekatan kedua, ternyata lebih mudah jika kita mengonversi larik `elements` menjadi *list* Python biasa terlebih dahulu:

```
elements = list(elements)
n = len(elements)

# Unsur logam akan ditandai dengan lingkaran biru ('b').
c = ['b'] * n

for symbol in nonmetals:
    if symbol in elements:
        idx = elements.index(symbol)
        # Unsur non-logam akan ditandai dengan lingkaran merah ('r').
        c[idx] = 'r'
```

Unsur manakah yang memiliki selisih terbesar antara titik leleh dan titik didihnya?

```
# Catatan: Terdapat entri NaN (Not a Number) untuk unsur-unsur
# yang nilai Tm atau Tb-nya belum diketahui.
idx_max_diff = np.nanargmax(Tb - Tm)
print(elements[idx_max_diff])
```

Output:

```
Np
```

Ternyata unsur tersebut adalah Neptunium. Sekarang kita siap memplot datanya dan menambahkan fungsi untuk menganotasi titik-titik tersebut:

```
plt.scatter(Tm, Tb, c=c, s=15)
plt.xlabel('Titik Leleh (K)')
plt.ylabel('Titik Didih (K)')

def annotate_with_element_symbol(symbol):
    i = elements.index(symbol)
    # Tambah sedikit offset agar teks tidak menutupi titik penanda
    plt.text(Tm[i] + 50, Tb[i] + 50, symbol, ha='left', va='bottom')

annotate_with_element_symbol('C')
annotate_with_element_symbol('Np')

plt.show()
```

Unsur non-logam ditandai dengan warna merah dan logam dengan warna biru. Anotasi menunjukkan posisi Karbon (C) dan Neptunium (Np).

Kasus 2 Kelarutan Kalium Halida

Data berikut berkaitan dengan kelarutan tiga kalium halida (dalam gram garam per 100 gram air) sebagai fungsi dari suhu:

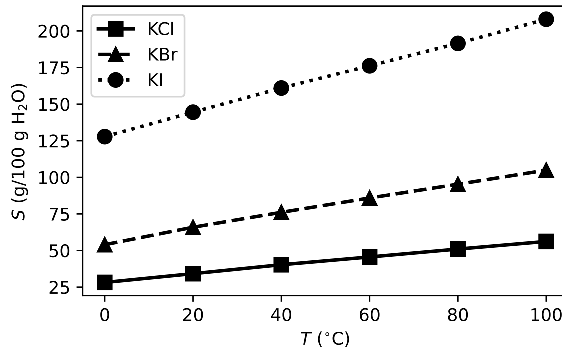
T / °C	KCl	KBr	KI
0	28.2	54.0	127.8
20	34.2	65.9	144.5
40	40.3	76.1	161.0
60	45.6	85.9	176.2
80	51.0	95.3	191.5
100	56.2	104.9	208.0

Buatlah plot data ini pada satu grafik tunggal yang diberi label dan judul, serta spesifikasi titik-titik datanya dengan gaya garis dan penanda yang berbeda-beda.

Solusi

Berikut adalah kode Python untuk menghasilkan grafik yang ditanyakan:

```
import numpy as np
import matplotlib.pyplot as plt

# Suhu (degC) dan kelarutan (g/100 g air).
T = np.arange(0, 101, 20)
S_KCl = np.array([28.2, 34.2, 40.3, 45.6, 51.0, 56.2])
S_KBr = np.array([54.0, 65.9, 76.1, 85.9, 95.3, 104.9])
S_KI = np.array([127.8, 144.5, 161.0, 176.2, 191.5, 208.0])

plt.figure(figsize=(5,3))

def plot_solubilities():
    """Plot kelarutan halida sebagai fungsi dari suhu."""
    plt.plot(T, S_KCl, c='k', ls='-', lw=2, marker='s', ms=8,
     ↪  label='KCl')
    plt.plot(T, S_KBr, c='k', ls='--', lw=2, marker='^', ms=8,
     ↪  label='KBr')
    plt.plot(T, S_KI, c='k', ls=':', lw=2, marker='o', ms=8,
     ↪  label='KI')

    plt.xlabel(r'$T~(^{\circ}\text{C})$')
    plt.ylabel(r'$S~(\text{g}/100~\text{g}~\text{H}_2\text{O})$')
    plt.legend()

    # Tambahkan judul plot:
    ## untuk memastikannya muat, gunakan jeda baris (\n).
    plt.title('Ketergantungan kelarutan beberapa garam kalium\n
     ↪  terhadap suhu')

plot_solubilities()
plt.show()
```



Kasus 3 Visualisasi Distribusi Maxwell-Boltzmann

Distribusi Maxwell-Boltzmann adalah distribusi probabilitas memperoleh kelajuan, v, dari partikel-partikel dalam gas ideal untuk kesetimbangan termodinamika pada suhu, T:

$$f(v) = \left(\frac{m}{2\pi k_B T}\right)^{3/2} 4\pi v^2 \exp\left(-\frac{mv^2}{2k_B T}\right),$$

dengan m adalah massa partikel.
Buatlah fungsi Python untuk menghitung distribusi Maxwell-Boltzmann dan plot distribusinya untuk gas nitrogen (N_2) pada rentang $v = 0$ hingga 2000 m/s untuk dua nilai suhu berbeda: $T = 300$ K dan $T = 600$ K.

Solusi

Fungsi Python untuk menghitung distribusi ini sangat mudah ditulis dan digunakan:

```python
import numpy as np
import matplotlib.pyplot as plt

# Konstanta Boltzmann (J/K) dan 1 amu (kg)
kB, u = 1.381e-23, 1.661e-27

def fMB(v, T, m):
    """
    Memberikan nilai distribusi Maxwell-Boltzmann untuk molekul
    bermassa m (dalam kg) pada suhu T (dalam K), dengan kelajuan v
    ↪   (m/s).
    """
    fac = m / (2 * kB * T)
    return (fac / np.pi)**1.5 * 4 * np.pi * v**2 * np.exp(-fac * v**2)

# Grid kelajuan antara 0 dan vmax sebanyak 1000 titik
vmax = 2000
v = np.linspace(0, vmax, 1000)

# Massa molekuler (dalam kg) dari molekul N2.
mN2 = 2 * 14 * u

# Hitung distribusi Maxwell-Boltzmann untuk gas N2
# pada dua suhu yang berbeda (dalam K).
T1, T2 = 300, 600
f_N2_T1 = fMB(v, T1, mN2)
f_N2_T2 = fMB(v, T2, mN2)
```

Fungsi Matplotlib `fill_between` dapat digunakan untuk mengisi area di bawah sebuah kurva. Argumen dasarnya adalah `x`, yaitu koordinat-x dari area yang akan diisi, serta `y1` dan `y2` yang merupakan batas-batas dari koordinat-y. Jika tidak diberikan, nilai bawaan `y2=0` akan digunakan, yang berarti area di bawah kurva `y1(x)` hingga menyentuh sumbu-x akan diisi. Warna yang digunakan untuk mengisi area tersebut dapat ditetapkan dengan argumen `facecolor` (atau disingkat `fc`).

```
plt.figure(figsize=(5,3))

def plot_MB():
    plt.plot(v, f_N2_T1)
    plt.plot(v, f_N2_T2, c='tab:red')

    plt.xlabel('v (m/s)')
    plt.ylabel('f(v) [s/m]')

    plt.fill_between(v, f_N2_T1, alpha=0.2, label=f'{T1} K')
    plt.fill_between(v, f_N2_T2, alpha=0.2, fc='tab:red',
                     label=f'{T2} K')

    plt.title('Distribusi Maxwell-Boltzmann gas nitrogen')
    plt.legend()

plot_MB()
plt.show()
```

Catatan: Argumen `alpha=0.2` pada kode di atas menetapkan tingkat opasitas/ketebalan isian warna sebesar 20%.