DASAR PEMROGRAMAN

Python



Noval Agung

Tutorial Belajar Python (Gratis!)

Python adalah bahasa pemrograman high-level yang sangat *powerful*, sintaksnya sederhana dan mudah dipelajari, juga memiliki performa yang bagus. Python memiliki komunitas yang besar, bahasa ini dipakai di berbagai platform diantaranya: web, data science, infrastructure tooling, dan lainnya.

Website/ebook tutorial Dasar Pemrograman Python ini cocok untuk pembaca yang ingin mempelajari pemrograman python dalam kurun waktu yang relatif cepat, dan gratis. Konten pembelajaran pada ebook ini disajikan secara ringkas tidak bertele-tele tapi tetap mencakup point penting yang harus dipelajari.

Selain topik fundamental python programming, nantinya akan disedikan juga pembahasan *advance* lainnya, **stay tuned!**

> *Versi website/ebook: **v1.2.20240923**, dan versi Python 3.12.0.*

Update Konten & Versi Terbaru

Setiap perubahan pada website/ebook bisa dilihat di halaman berikut:

> *Version Changelogs & Updates*

Download Ebook File (pdf)

Versi ebook bisa di-download dalam bentuk file di link berikut:

> *PDF/Ebook file: Dasar Pemrograman Python.pdf*

Source Code Praktik

Source code website/ebook & praktik program bisa diunduh di link berikut:

> - *Source code website/ebook ➡ https://github.com/novalagung/dasarpemrogramanpython*
> - *Source code praktik ➡ https://github.com/novalagung/dasarpemrogramanpython-example*

Dianjurkan untuk sekedar tidak copy-paste dari source code dalam proses belajar, usahakan tulis sendiri kode program agar cepat terbiasa dengan bahasa Python.

Kontribusi

Website/ebook ini merupakan project open source, teruntuk siapapun yang ingin berkontribusi silakan langsung saja cek github.com/novalagung/dasarpemrogramanpython. Cek juga halaman kontributor berikut untuk melihat list kontributor.

> *[Contribution & Tech Stack][/CONTRIBUTING]*

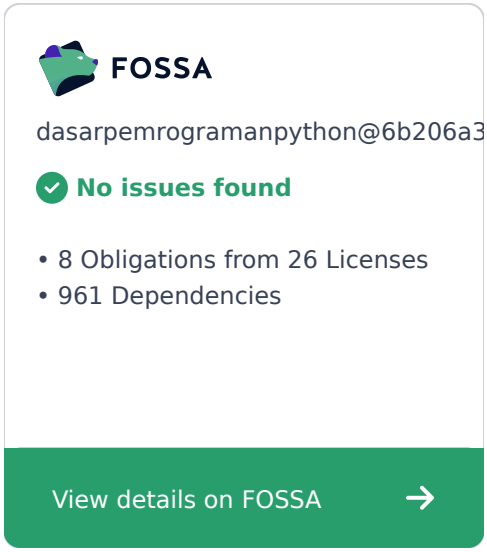
Lisensi dan Status FOSSA

Website/ebook tutorial Dasar Pemrograman Python gratis untuk disebarluaskan

secara bebas, baik untuk komersil maupun tidak, dengan catatan harus disertakan credit sumber aslinya (yaitu Dasar Pemrograman Python atau novalagung) dan tidak mengubah lisensi aslinya (yaitu CC BY-SA 4.0). Lebih jelasnya silakan cek halaman lisensi dan distribusi konten.

> *Lisensi dan Distribusi Konten*

FOSSA Status:



Author & Maintainer

Ebook ini dibuat oleh Noval Agung Prayogo. Untuk pertanyaan, kritik, dan saran, silakan drop email ke .

Version Changelogs & Updates

📝 Release v1.2.20240923 (2024-09-23)

◉ Chapter update

- Perbaikan typo dan/atau kesalahan kode
  - A.10. Perulangan ➜ while
  - A.11. Perulangan ➜ break, continue
  - A.12. List
  - A.15. Set
  - A.16. Python Dictionary
  - A.17. String
  - A.18. String ➜ Unicode
  - A.19. Number
  - A.21. Object ID & Reference
  - A.22. Function
  - A.23. Function ➜ Positional, Optional, Keyword Arguments
  - A.26. Function ➜ Lambda
  - A.27. Modules
  - A.48. DocString
  - A.49. File I/O
  - A.50. CLI Arguments & Flags
  - A.51. File/Data Format ➜ CSV
  - A.52. File/Data Format ➜ JSON
  - A.53. Date, Time, DateTime, Timezone

    - A.54. DateTime ➜ Parsing & Formatting
- Perbaikan kesalahan kode

◉ General update

- Penambahan halaman changelogs
- Update landing page untuk optimasi SEO

Contribution & Tech Stack

Ebook Dasar Pemrograman Python adalah project open source. Siapapun bebas untuk berkontribusi di sini, bisa dalam bentuk perbaikan typo, update kalimat, maupun submit tulisan baru.

Bagi kawan-kawan yang berminat untuk berkontribusi, silakan fork github.com/novalagung/dasarpemrogramanpython, kemudian langsung saja cek/buat issue kemudian submit relevan pull request untuk issue tersebut 😊

Checkout project

```
git clone https://github.com/novalagung/dasarpemrogramanpython.git
git submodule update --init --recursive --remote
```

Maintainer

E-book ini di-inisialisasi dan di-maintain oleh Noval Agung Prayogo.

Contributors

Berikut merupakan hall of fame kontributor yang sudah berbaik hati menyisihkan waktunya untuk membantu pengembangan e-book ini.

1. Arsy Opraza Akma
2. Barep Bimo Pangestu
3. Mohammad Safri Dwi Rizky
4. Muhammad Saleh Solahudin

5. ... anda 🚀

Ebook/webhook techstack & architecture

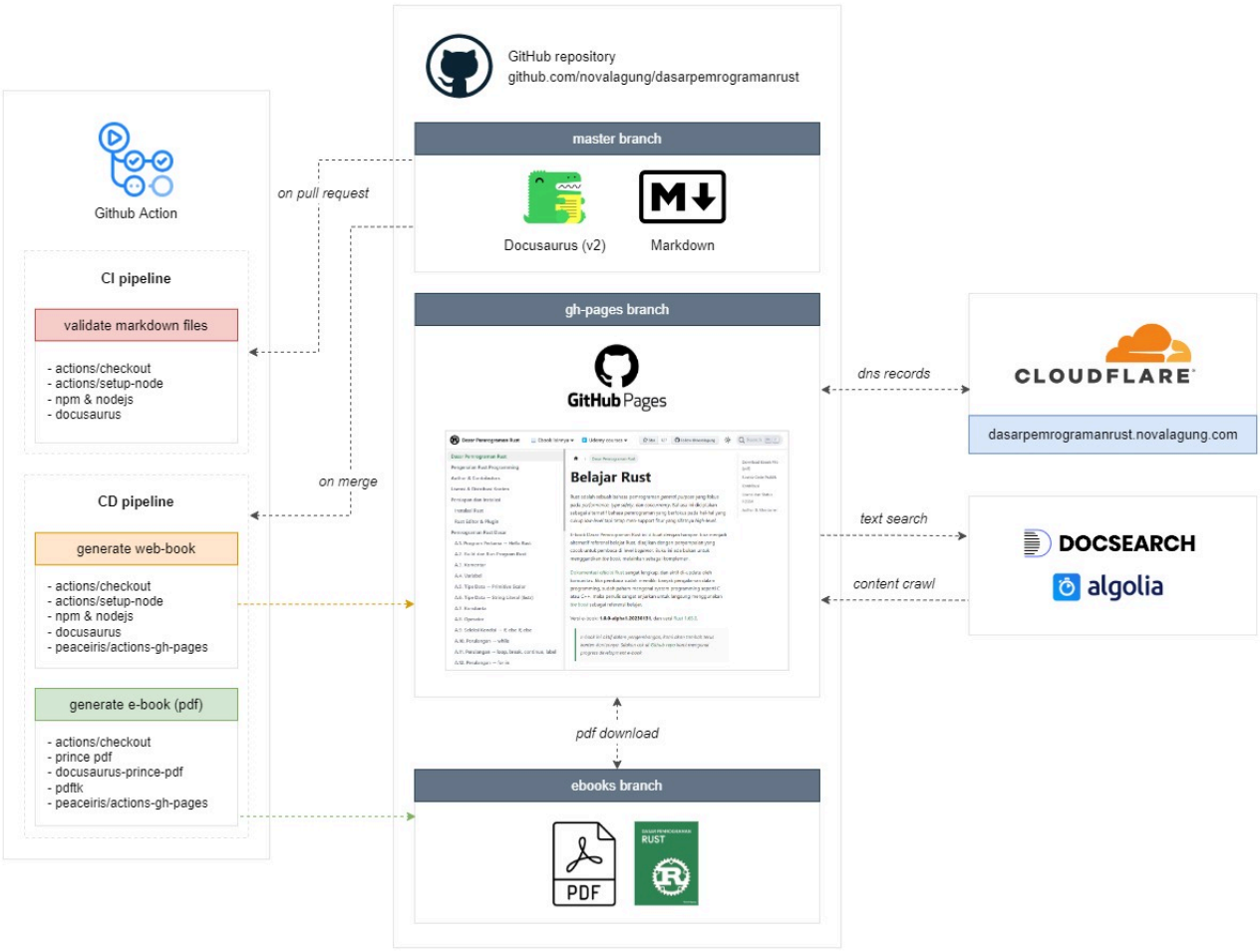
Jika pembaca tertarik untuk membuat konten ebook berbasis web sekaligus versi PDF file-nya, bisa menggunakan link berikut sebagai referensi:

> *https://medium.com/geekculture/serverless-e-book-web-book-using-docusaurus-v2-github-pages-actions-and-pdf-tools-4fef54847b85*

Tools yang digunakan:

- Web-book engine ➜ Docusaurus v2
- PDF generator ➜ Prince XML + docusaurus-prince-pdf + PDFtk
- Source code repository ➜ GitHub
- Hosting ➜ GitHub Pages
- CI/CD pipeline ➜ GitHub Actions
- Content Search ➜ Algolia DocSearch

Desain arsitektur:


GitHub repository
github.com/novalagung/dasarpemrograman rust
master branch
Docusaurus (v2)
Markdown
Github Action
on pull request
CI pipeline
validate markdown files
- actions/checkout
- actions/setup-node
- npm & nodejs
- docusaurus
on merge
CD pipeline
generate web-book
- actions/checkout
- actions/setup-node
- npm & nodejs
- docusaurus
- peaceiris/actions-gh-pages
generate e-book (pdf)
- actions/checkout
- prince pdf
- docusaurus-prince-pdf
- pdftk
- peaceiris/actions-gh-pages
gh-pages branch
GitHub Pages
Belajar Rust
dns records
CLOUDFLARE
dasarpemrograman rust.novalagung.com
text search
content crawl
DOCSEARCH
algolia
pdf download
ebooks branch
PDF
RUST

Download versi PDF

Ebook Dasar Pemrograman Python bisa di-download dalam bentuk file PDF, silakan gunakan link berikut:

> *Dasar Pemrograman Python.pdf*

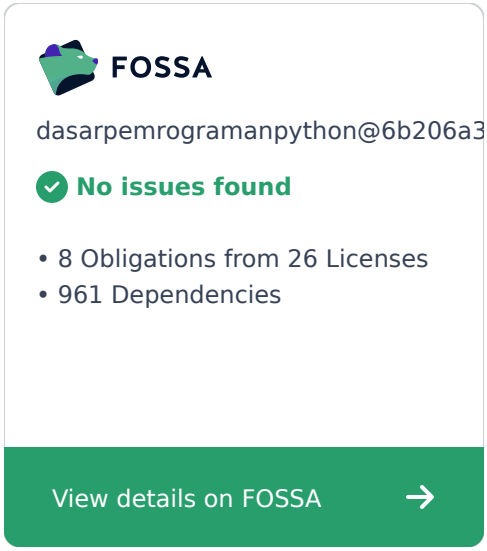
Lisensi & Distribusi Konten

Ebook Dasar Pemrograman Python gratis untuk disebarluaskan secara bebas, dengan catatan sesuai dengan aturan lisensi CC BY-SA 4.0 yang kurang lebih sebagai berikut:

- Diperbolehkan menyebar, mencetak, dan menduplikasi material dalam konten ini ke siapapun.
- Diperbolehkan memodifikasi, mengubah, atau membuat konten baru menggunakan material yang ada dalam ebook ini untuk keperluan komersil maupun tidak.

Dengan catatan:

- Harus ada credit sumber aslinya, yaitu Dasar Pemrograman Python atau novalagung
- Tidak mengubah lisensi aslinya, yaitu CC BY-SA 4.0
- Tidak ditambahi restrictions baru
- Lebih jelasnya silakan cek https://creativecommons.org/licenses/by-sa/4.0/.


FOSSA
dasarpemrogramanpython@6b206a3
No issues found
• 8 Obligations from 26 Licenses
• 961 Dependencies
View details on FOSSA

Instalasi Python

Ada banyak cara yang bisa dipilih untuk instalasi Python, silakan pilih sesuai preferensi dan kebutuhan.

Instalasi Python

◉ Instalasi di Windows

- Via Microsoft Store Package
- Via Official Python installer
- Via Chocolatey package manager
- Via Windows Subsystem for Linux (WSL)

◉ Instalasi di MacOS

- Via Homebrew
- Via Official Python installer

◉ Instalasi di OS lainnya

- Via package manager masing-masing sistem operasi

◉ Instalasi via source code

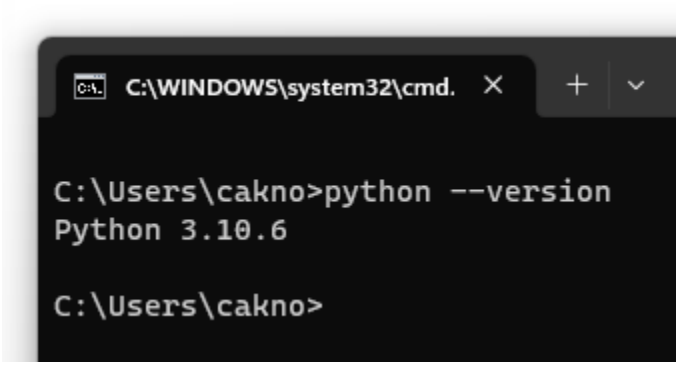
- Tarball source code bisa diunduh di situs official Python

◉ Instalasi via Anaconda

- File installer bisa diunduh di situs official Anaconda

Konfigurasi path Python

1. Pastikan untuk mendaftarkan path dimana Python ter-install ke OS environment variable, agar nantinya mudah dalam pemanggilan binary `python`.
2. Jika diperlukan, set juga variabel `PYTHONHOME` yang mengarah ke base folder dimana Python terinstall. Biasanya editor akan mengacu ke environment variabel ini untuk mencari dimana path Python berada.
3. Kemudian, jalankan command `python --version` untuk memastikan binary sudah terdaftar di `$PATH` variable.



Python Editor & Plugin

Editor/IDE

Ada cukup banyak pilihan editor dan IDE untuk development menggunakan Python, diantaranya:

- Eclipse, dengan tambahan plugin PyDev
- GNU Emacs
- JetBrains PyCharm
- Spyder
- Sublime Text, dengan tambahan package Python
- Vim
- Visual Studio
- Visual Studio Code (VSCode), dengan tambahan extension Python dan Jupyter

Selain list di atas, ada juga editor lainnya yang bisa digunakan, contohnya seperti:

- Python standard shell (REPL)
- Jupyter

Preferensi editor penulis

Penulis menggunakan editor Visual Studio Code dengan tambahan:

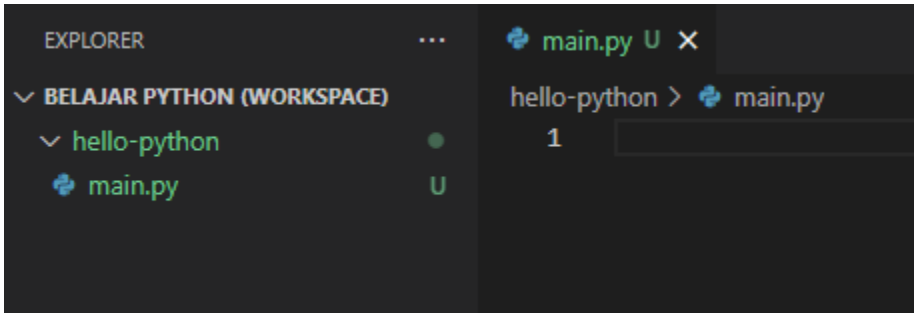
- Extension Python, untuk mendapatkan benefit API doc, autocompletion, linting, run feature, dan lainnya.
- Extension Jupyter, untuk interactive run program via editor.

A.1. Python Hello World

Bahasa pemrograman Python sangat sederhana dan mudah untuk dipelajari. Pada chapter ini kita akan langsung mempraktikannya dengan membuat program hello world.

A.1.1. Program Hello Python

Siapkan sebuah folder dengan isi satu file program Python bernama `main.py`.

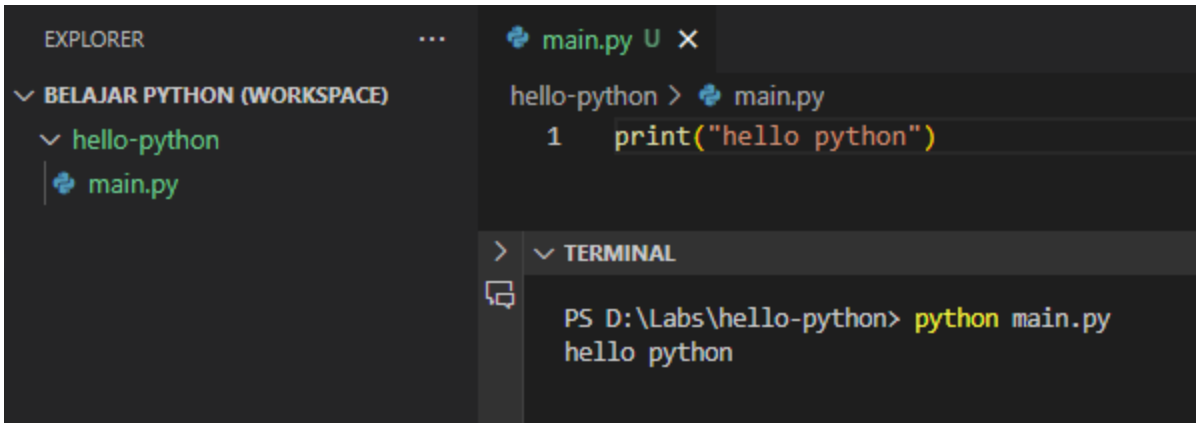


Pada file `main.py`, tuliskan kode berikut:

```
print("hello python")
```

Run program menggunakan command berikut:

```
# python <nama_file_program>
python main.py
```



Selamat, secara official sekarang anda adalah programmer Python! 🎉 Mudah bukan!?

A.1.2. Penjelasan program

Folder `hello-python` bisa disebut dengan folder **project**, dimana isinya adalah file-file program Python berekstensi `.py`.

File `main.py` adalah file program python. Nama file program bisa apa saja, tapi umumnya pada pemrograman Python, file program utama bernama `main.py`.

Command `python <nama_file_program>` digunakan untuk menjalankan program. Cukup ganti `<nama_file_program>` dengan nama file program (yang pada contoh ini adalah `main.py`) maka kode program di dalam file tersebut akan di-run oleh Python interpreter.

Statement `print("<pesan_text>")` adalah penerapan dari salah satu fungsi *built-in* yang ada dalam Python stdlib (standard library), yaitu fungsi bernama `print()` yang kegunaannya adalah untuk menampilkan pesan string (yang disipkan pada argument pemanggilan fungsi `print()`). Pesan tersebut akan mucnul ke layar output stdout (pada contoh ini adalah terminal milik editor

penulis).

> - *Pembahasan detail mengenai fungsi ada di chapter Function*
> - *Pembahasan detail mengenai Python standard library (stdlib) ada di chapter Python standard library (stdlib)*
>
> *Untuk sekarang, penulis tidak anjurkan untuk lompat ke pembahasan tersebut. Silakan ikuti pembelajaran chapter per chapter secara berurutan.*

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../hello-python
```

◉ Referensi

- https://www.learnpython.org/en/Hello,_World!
- https://docs.python.org/3/library/functions.html

A.2. Run Python di VSCode

Chapter ini membahas tentang pilihan opsi cara run program Python di Visual Studio Code.

A.2.1. Cara run program Python di VSCode

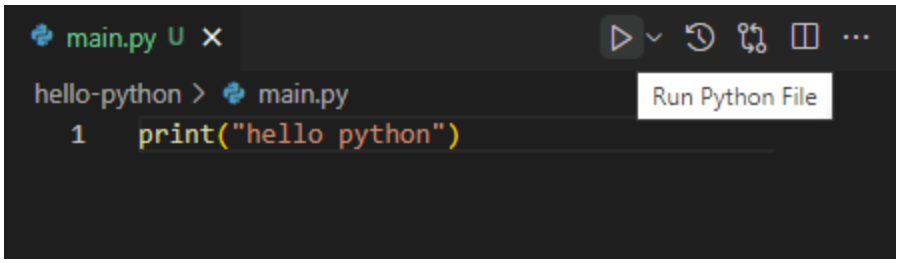
◉ Menggunakan command `python`

Command ini sudah kita terapkan pada chapter Program Pertama ➜ Hello Python, cara penggunaannya cukup mudah, tinggal jalankan saja command di terminal.

```
# python <nama_file_program>
python main.py
```

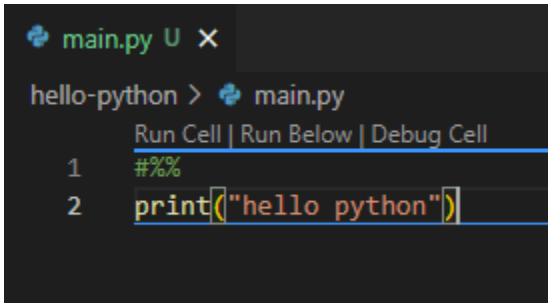
◉ Menggunakan tombol run ▶

Cara run program ini lebih praktis karena tingal klik-klik saja. Di toolbar VSCode sebelah kanan atas ada tombol ▶, gunakan tombol tersebut untuk menjalankan program.

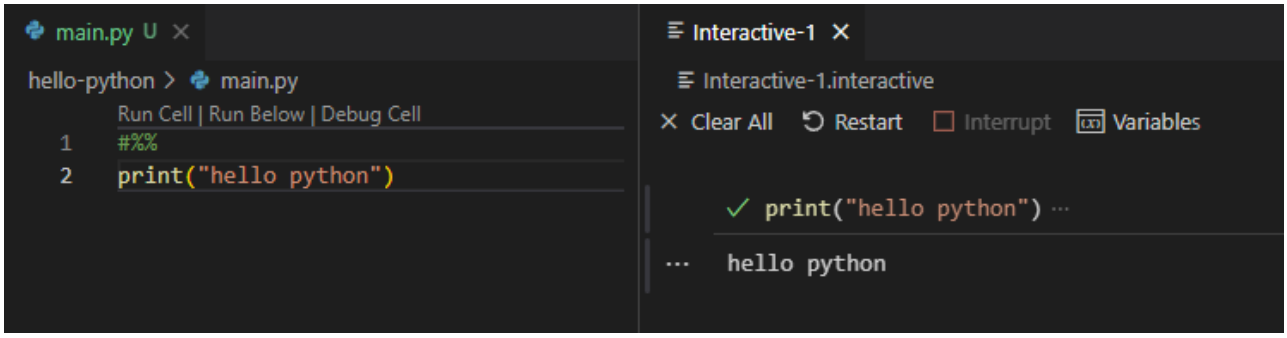


◉ Menggunakan jupyter `code cells`

Untuk menerapkan cara ini, tambahkan kode `#%%` atau `# %%` pada baris di atas statement `print("hello python")` agar blok kode di bawahnya dianggap sebagai satu `code cell`.



Setelah itu, muncul tombol `Run Cell`, klik untuk run program.



Catatan chapter 📑

◉ Chapter relevan lainnya

- Program Pertama ➜ Hello Python

◉ Referensi

- https://code.visualstudio.com/docs/python/python-tutorial
- https://code.visualstudio.com/docs/datascience/jupyter-notebooks
- https://docs.python.org/3/using/cmdline.html

A.3. Python Komentar

Komentar adalah sebuah statement yang tidak akan dijalankan oleh interpreter. Biasanya digunakan untuk menambahkan keterangan atau men-disable statements agar tidak dieksekusi saat run program.

Python mengenal dua jenis komentar, yaitu komentar satu baris dan multi-baris.

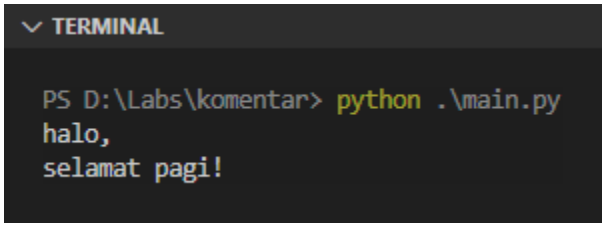
A.3.1. Komentar satu baris

Karakter `#` digunakan untuk menuliskan komentar, contoh:

```
# ini adalah komentar
print("halo,")
print("selamat pagi!") # ini juga komentar

# println("statement ini tidak akan dipanggil")
```

Jika di-run, outputnya:



Bisa dilihat statement yang diawali dengan tanda `#` tidak dieksekusi.

A.3.2. Komentar multi-baris

Komentar multi-baris bisa diterapkan melalui dua cara:

◉ Komentar menggunakan `#` dituliskan

```
# ini adalah komentar
# ini juga komentar
# komentar baris ke-3
```

◉ Komentar menggunakan `"""` atau `'''`

Karakter `"""` atau `'''` sebenarnya digunakan untuk membuat *multiline string* atau string banyak baris. Selain itu, bisa juga dipergunakan sebagai penanda komentar multi baris. Contoh penerapannya:

```
"""
ini adalah komentar
ini juga komentar
komentar baris ke-3
"""
```

Atau bisa juga ditulis seperti ini untuk komentar satu baris:

```
"""ini adalah komentar"""
```

> - *Pembahasan detail mengenai string ada di chapter String*
> - *Pembahasan detail mengenai DocString ada di chapter DocString*

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../komentar
```

◉ Chapter relevan lainnya

- String

◉ Referensi

- https://docs.python-guide.org/writing/documentation/

A.4. Python Variabel

Dalam konsep programming, variabel adalah suatu nama yang dikenali komputer sebagai penampung nilai/data yang disimpan di memory. Sebagai contoh nilai `3.14` disimpan di variabel bernama `PI`.

Pada chapter ini kita akan belajar tentang penerapan variabel di Python.

A.4.1. Deklarasi variabel

Agar dikenali oleh komputer, variabel harus dideklarasikan. Deklarasi variabel di Python cukup sederhana, caranya tinggal tulis saja nama variabel kemudian diikuti operator *assignment* beserta nilai awal yang ingin dimasukan ke variabel tersebut. Contoh:

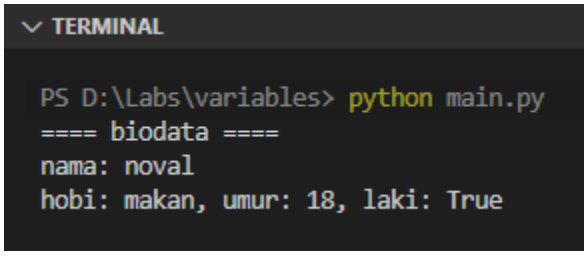
```
nama = "noval"
hobi = 'makan'
umur = 18
laki = True
```

Karakter `=` adalah **operator assignment**, digunakan untuk operasi penugasan. Nilai yang ada di sebelah kanan `=` ditugaskan untuk ditampung oleh variabel yang berada di sebelah kiri `=`. Contoh pada statement `nama = "noval"`, nilai `"nama"` ditugaskan untuk ditampung oleh variabel `nama`.

> *Nilai string bisa dituliskan dengan menggunakan literal `"` ataupun `'`*

Ok. Selanjutnya, coba kita munculkan nilai ke-empat variabel di atas ke layar menggunakan fungsi `print()`. Caranya:

```
print("==== biodata ====")
print("nama: %s" % (nama))
print("hobi: %s, umur: %d, laki: %r" % (hobi, umur, laki))
```



Penjelasan mengenai program di atas bisa dilihat di bawah ini:

◉ *String formatting* `print`

Di program yang sudah ditulis, ada statement berikut:

```
print("==== biodata ====")
```

Statement tersebut adalah contoh cara memunculkan string ke layar output (`stdout`):

Lalu di bawahnya ada statement ini, yang merupakan contoh penerapan teknik *string formatting* atau *output formatting* untuk mem-format string ke layar output:

```
print("nama: %s" % (nama))
# output → "nama: noval"
```

Karakter `%s` disitu akan di-replace dengan nilai variabel `nama` sebelum dimunculkan. Dan `%s` disini menandakan bahwa data yang akan me-replace-nya bertipe data `string`.

Selain `%s`, ada juga `%d` untuk data bertipe numerik integer, dan `%r` untuk data bertipe `bool`. Contoh penerapannya bisa dilihat pada statement ke-3 program yang sudah di tulis.

```
print("hobi: %s, umur: %d, laki: %r" % (hobi, umur, laki))
# output → "hobi: makan, umur: 18, laki: True"
```

> *Pembahasan detail mengenai string formatting ada di chapter String → formatting*

A.4.2. *Naming convention* variabel

Mengacu ke dokumentasi PEP 8 – Style Guide for Python Code, nama variabel dianjurkan untuk ditulis menggunakan `snake_case`.

```
pesan = 'halo, selamat pagi'
nilai_ujian = 99.2
```

A.4.3. Operasi *assignment*

Di pemrograman Python, deklarasi variabel adalah pasti operasi assignment. Variabel dideklarasikan dengan ditentukan langsung nilai awalnya.

```
nama = "noval"
umur = 18
nama = "noval agung"
umur = 21
```

A.4.4. Deklarasi variabel beserta tipe data

Tipe data variabel bisa ditentukan secara eksplisit, penulisannya bisa dilihat pada kode berikut:

```
nama: str = "noval"
hobi: str = 'makan'
umur: int = 18
laki: bool = True
nilai_ujian: float = 99.2
```

> *Pembahasan detail mengenai tipe data ada di chapter Tipe Data*

A.4.5. Deklarasi banyak variabel sebaris

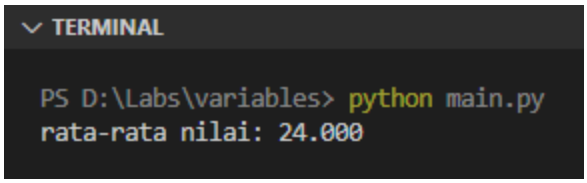
Contoh penulisan deklarasi banyak variabel dalam satu baris bisa dilihat pada kode berikut:

```
nilai1, nilai2, nilai3, nilai4 = 24, 25, 26, 21
nilai_rata_rata = (nilai1 + nilai2 + nilai3 + nilai4) / 4

print("rata-rata nilai: %f" % (nilai_rata_rata))
```

> *Karakter `%f` digunakan untuk mem-format nilai `float`*

Output program di atas:



Catatan chapter 📑

◉ Source code praktik

`github.com/novalagung/dasarpemrogramanpython-example/../variables`

◉ Chapter relevan lainnya

- Tipe Data
- String
- Number

◉ Referensi

- https://www.w3schools.com/python/python_datatypes.asp
- https://peps.python.org/pep-0008/
- https://en.wikipedia.org/wiki/Snake_case
- https://www.learnpython.org/en/String_Formatting

A.5. Python Konstanta

Konstanta (atau nilai konstan) adalah sebuah variabel yang nilainya dideklarasikan di awal dan tidak bisa diubah setelahnya.

Pada chapter ini kita akan mempelajari tentang penerapan Konstanta di Python.

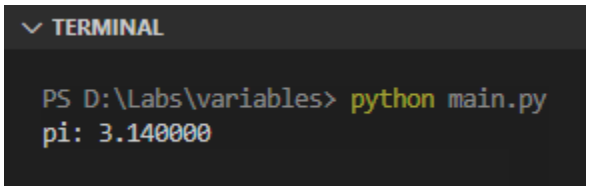
A.5.1. Konstanta di Python

Deklarasi konstanta di Python dilakukan menggunakan bantuan tipe *class* bernama `typing.Final`.

Untuk menggunakannya, `typing.Final` perlu di-import terlebih dahulu menggunakan keyword `from` dan `import`.

```
from typing import Final

PI: Final = 3.14
print("pi: %f" % (PI))
```



◉ Module import

Keyword `import` digunakan untuk meng-import sesuatu, sedangkan keyword `from` digunakan untuk menentukan dari module mana sesuatu tersebut akan

di-import.

> *Pembahasan detail mengenai `import` dan `from` ada di chapter Modules*

Statement `from typing import Final` artinya adalah meng-import tipe `Final` dari module `typing` yang dimana module ini merupakan bagian dari Python standard library (stdlib).

> *Pembahasan detail mengenai Python standard library (stdlib) ada di chapter Python standard library (stdlib)*

A.5.2. Tipe *class* `typing.Final`

Tipe `Final` digunakan untuk menandai suatu variabel adalah tidak bisa diubah nilainya (konstanta). Cara penerapan `Final` bisa dengan dituliskan tipe data konstanta-nya secara eksplisit, atau boleh tidak ditentukan (tipe akan diidentifikasi oleh interpreter berdasarkan tipe data nilainya).

```
# tipe konstanta PI tidak ditentukan secara explisit,
# melainkan didapat dari tipe data nilai
PI: Final = 3.14

# tipe konstanta TOTAL_MONTH ditentukan secara explisit yaitu `int`
TOTAL_MONTH: Final[int] = 12
```

> *Pembahasan detail mengenai tipe data ada di chapter Tipe Data*

A.5.3. *Naming convention* konstanta

Mengacu ke dokumentasi PEP 8 – Style Guide for Python Code, nama konstanta harus dituliskan dalam huruf besar (UPPER_CASE).

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../konstanta
```

◉ Chapter relevan lainnya

- Variabel
- Tipe Data
- Modules
- Python standard library (stdlib)

◉ Referensi

- https://docs.python.org/3/library/typing.html#typing.Final
- https://peps.python.org/pep-0008/

A.6. Python Tipe Data

Python mengenal cukup banyak tipe data, mulai dari yang *built-in* (atau bawaan) maupun custom type. Pada chapter ini kita akan mempelajari *high-level overview* tipe-tipe tersebut.

A.6.1. Tipe data numerik

Ada setidaknya 3 tipe data numerik di Python, yaitu:

Tipe data	Keterangan	Contoh
`int`	menampung bilangan bulat atau *integer*	`number_1 = 10000024`
`float`	menampung bilangan desimal atau *floating point*	`number_2 = 3.14`
`complex`	menampung nilai berisi kombinasi bilangan real dan imajiner	`number_3 = 120+3j`

> *Penjelasan detail mengenai string ada pada chapter Number*

A.6.2. Tipe data string / `str`

Tipe string direpresentasikan oleh `str`, pembuatannya bisa menggunakan

literal string yang ditandai dengan tanda awalan dan akhiran tanda `"` atau `'`.

- Menggunakan tanda petik dua (`"`)

```
# string sebaris
string_1 = "hello python"

# string multi-baris
string_2 = """Selamat
Belajar
Python"""
```

- Menggunakan tanda petik satu (`'`)

```
# string sebaris
string_3 = 'for the horde!'

# string multi-baris
string_4 = '''
Sesuk
Preiiii
'''
```

Jika ada baris baru (atau *newline*) di bagian awal penulisan `'''` atau `"""` maka baris baru tersebut merupakan bagian dari string. Jika ingin meng-*exclude*-nya bisa menggunakan `"""\` atau `'''\`. Contoh:

```
string_5 = '''\
Sesuk
Preiiii
'''
```

> *Penjelasan detail mengenai string ada pada chapter String*

A.6.3. Tipe data `bool`

Literal untuk tipe data boolean di Python adalah `True` untuk nilai benar, dan `False` untuk nilai salah.

```
bool_1 = True
bool_2 = False
```

A.6.4. Tipe data `None`

Tipe data `None` merepresentasikan nilai kosong (seperti nilai `null` atau `nil` di bahasa lain). Pengecekan nilai kosong digunakan

```
data = "hello"
print(data)
# output → hello

data = None
print(data)
# output → data

data = "python"
print(data)
# output → python
```

> *Penjelasan lebih detail mengenai `None` ada pada chapter None*

A.6.5. Tipe data list

List adalah tipe data di Python untuk menampung nilai kolektif yang disimpan secara urut, dengan isi bisa berupa banyak varian tipe data (tidak harus sejenis). Cara penerapan list adalah dengan menuliskan nilai kolektif dengan pembatas `,` dan diapit tanda `[` dan `]`.

```
# list with int as element's data type
list_1 = [2, 4, 8, 16]

# list with str as element's data type
list_2 = ["grayson", "jason", "tim", "damian"]

# list with various data type in the element
list_3 = [24, False, "Hello Python"]
```

Pengaksesan element list menggunakan notasi `list[index_number]`. Contoh:

```
list_1 = [2, 4, 8, 16]
print(list_1[2])
# output → 8
```

> *Penjelasan detail mengenai list ada pada chapter List*

A.6.6. Tipe data tuple

Tuple adalah tipe data kolektif yang mirip dengan list, dengan pembeda adalah:

- Nilai pada data list adalah bisa diubah (*mutable*), sedangkan nilai data `tuple` tidak bisa diubah (*immutable*).
- List menggunakan tanda `[` dan `]` untuk penulisan literal, sedangkan pada tuple yang digunakan adalah tanda `(` dan `)`.

```
# tuple with int as element's data type
tuple_1 = (2, 3, 4)

# tuple with str as element's data type
tuple_2 = ("numenor", "valinor")

# tuple with various data type in the element
tuple_3 = (24, False, "Hello Python")
```

Pengaksesan element tuple menggunakan notasi `tuple[index_number]`. Contoh:

```
tuple_1 = (2, 3, 4)
print(tuple_1[2])
# output → 4
```

> *Penjelasan detail mengenai tuple ada pada chapter Tuple*

A.6.7. Tipe data dictionary

Tipe data `dict` atau dictionary berguna untuk menyimpan data kolektif terstruktur berbentuk *key value*. Contoh penerapan:

```
profile_1 = {
  "name": "Noval",
```

Pengaksesan property dictionary menggunakan notasi `dict[property_name]`. Contoh:

```
print("name: %s" % (profile_1["name"]))
# output → name: Noval

print("hobbies: %s" % (profile_1["hobbies"]))
# output → name: ["gaming", "learning"]
```

Penulisan data dictionary diperbolehkan secara horizontal, contohnya seperti berikut:

```
profile_1 = { "name": "Noval", "hobbies": ["gaming", "learning"] }
```

> *Penjelasan detail mengenai dictionary ada pada chapter Dictionary*

A.6.8. Tipe data set

Tipe data set adalah cara lain untuk menyimpan data kolektif. Tipe data ini memiliki beberapa kelemahan:

- Tidak bisa menyimpan informasi urutan data
- Elemen data yang sudah dideklarasikan tidak bisa diubah nilainya (tapi bisa dihapus)
- Tidak bisa diakses menggunakan index (tetapi bisa menggunakan perulangan)

Contoh penerapan set:

```
set_1 = {"pineapple", "spaghetti"}
print(set_1)
# output → {"pineapple", "spaghetti"}
```

> *Penjelasan detail mengenai set ada pada chapter Set*

A.6.9. Tipe data lainnya

Selain tipe-tipe di atas ada juga beberapa tipe data lainnya, seperti frozenset, bytes, memoryview, range; yang kesemuanya akan dibahas satu per satu di chapter terpisah.

> - *Penjelasan detail mengenai frozenset ada pada chapter Set → frozenset*
> - *Penjelasan detail mengenai range ada pada chapter Perulangan → for & range*

Catatan chapter 🗒️

◉ Source code praktik

`github.com/novalagung/dasarpemrogramanpython-example/../tipe-data`

◉ Chapter relevan lainnya

- String

- List
- Tuple
- Set
- Dictionary

◉ Referensi

- https://docs.python.org/3/tutorial/introduction.html
- https://docs.python.org/3/library/stdtypes.html#typesseq

A.7. Python Operator

Operator adalah suatu karakter yang memiliki kegunaan khusus contohnya seperti `+` untuk operasi aritmatika tambah, dan `and` untuk operasi logika **AND**.

Pada chapter ini kita akan mempelajari macam-macam operator yang ada di Python.

A.7.1. Operator aritmatika

Operator	Keterangan	Contoh
`+`	operasi tambah	`num = 2 + 2` ➜ hasilnya `num` nilainya `4`
unary `+`	penanda nilai positif	`num = +2` ➜ hasilnya `num` nilainya `2`
`-`	operasi pengurangan	`num = 3 - 2` ➜ hasilnya `num` nilainya `1`
unary `-`	penanda nilai negatif	`num = -2` ➜ hasilnya `num` nilainya `-2`
`*`	operasi perkalian	`num = 3 * 3` ➜ hasilnya `num` nilainya `9`

Operator	Keterangan	Contoh
`/`	operasi pembagian	`num = 8 / 2` ➜ hasilnya `num` nilainya `4`
`//`	operasi bagi dengan hasil dibulatkan ke bawah	`num = 10 // 3` ➜ hasilnya `num` nilainya `3`
`%`	operasi modulo (pencarian sisa hasil bagi)	`num = 7 % 4` ➜ hasilnya `num` nilainya `3`
`**`	operasi pangkat	`num = 3 ** 2` ➜ hasilnya `num` nilainya `9`

A.7.2. Operator *assignment*

Operator assignment adalah `=`, digunakan untuk operasi assignment (penugasan nilai atau penentuan nilai), sekaligus untuk deklarasi variabel jika variabel tersebut sebelumnya belum terdeklarasi. Contoh:

```
# deklarasi variabel num_1
num_1 = 12

# deklarasi variabel num_2
num_2 = 24

# nilai baru ditugaskan ke variabel num_2
num_2 = 12

# deklarasi variabel num_3 dengan isi nilai hasil operasi aritmatika
`num_1 + num_2`
```

A.7.3. Operator perbandingan

Operator perbandingan pasti menghasilkan nilai kebenaran `bool` dengan kemungkinannya hanya dua nilai, yaitu benar (`True`) atau salah (`False`).

Python mengenal operasi perbandingan standar yang umumnya juga dipakai di bahasa lain.

Operator	Keterangan	Contoh
`==`	apakah kiri sama dengan kanan	`res = 4 == 5` ➜ hasilnya `res` nilainya `False`
`!=`	apakah kiri tidak sama dengan kanan	`res = 4 != 5` ➜ hasilnya `res` nilainya `True`
`>`	apakah kiri lebih besar dibanding kanan	`res = 4 > 5` ➜ hasilnya `res` nilainya `False`
`<`	apakah kiri lebih kecil dibanding kanan	`res = 4 < 5` ➜ hasilnya `res` nilainya `True`
`>=`	apakah kiri lebih besar atau sama dengan kanan	`res = 5 >= 5` ➜ hasilnya `res` nilainya `True`
`<=`	apakah kiri lebih kecil atau sama dengan kanan	`res = 4 <= 5` ➜ hasilnya `res` nilainya `False`

A.7.4. Operator logika

Operator	Keterangan	Contoh
`and`	operasi logika **AND**	`res = (4 == 5) and (2 != 3)` ➜ hasilnya `res` nilainya `False`
`or`	operasi logika **OR**	`res = (4 == 5) or (2 != 3)` ➜ hasilnya `res` nilainya `True`
`not` atau `!`	operasi logika **negasi** (atau **NOT**)	`res = not (2 == 3)` ➜ hasilnya `res` nilainya `True` `res = !(2 == 3)` ➜ hasilnya `res` nilainya `True`

A.7.5. Operator bitwise

Operator	Keterangan	Contoh
`&`	operasi bitwise **AND**	`x & y = 0 (0000 0000)`
`|`	operasi bitwise **OR**	`x | y = 14 (0000 1110)`
`~`	operasi bitwise **NOT**	`~x = -11 (1111 0101)`
`^`	operasi bitwise **XOR**	`x ^ y = 14 (0000 1110)`

Operator	Keterangan	Contoh
`>>`	operasi bitwise **right shift**	`x >> 2 = 2 (0000 0010)`
`<<`	operasi bitwise **left shift**	`x << 2 = 40 (0010 1000)`

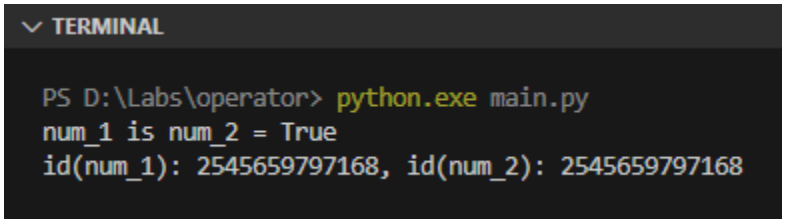
A.7.6. Operator *identity* (`is`)

Operator `is` memiliki kemiripan dengan operator logika `==`, perbedaannya pada operator `is` yang dibandingkan bukan nilai, melainkan identitas atau ID-nya.

Bisa saja ada 2 variabel bernilai sama tapi identitasnya berbeda. Contoh:

```
num_1 = 100001
num_2 = 100001

res = num_1 is num_2
print("num_1 is num_2 =", res)
print("id(num_1): %s, id(num_2): %s" % (id(num_1), id(num_2)))
```



> *Di Python ada special case yang perlu kita ketahui perihal penerapan operator `is` untuk operasi perbandingan identitas khusus tipe data numerik. Pembahasan detailnya ada di chapter Object ID & Reference.*

INFO

Pengecekan nilai kosong (atau `None`) dianjurkan untuk selalu dilakukan menggunakan operator `is`, dan menghindari penggunaan operator `==`.

Hal ini karena operator `is` membandingkan identitas data dan identitas data `None` selalu valid. Sedangkan operator `==` perbandingannya dilakukan dengan via special method `__eq__()` yang default method tersebut bisa di-override isinya.

> *Lebih jelasnya penggunaan operator `is` terhadap `None` di bahas di chapter None*

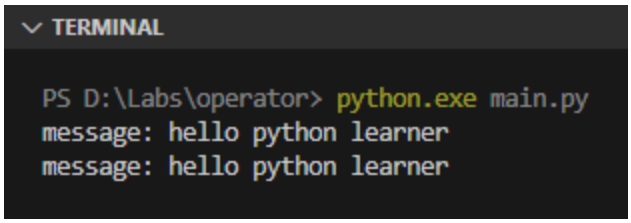
◉ Fungsi `print()` tanpa string formatting

Statement `print("num_1 is not num_2 =", res)` adalah salah satu cara untuk printing data tanpa menggunakan string formatting (seperti `%s`).

Yang terjadi pada statement tersebut adalah, semua nilai argument pemanggilan fungsi `print()` akan digabung dengan delimiter karakter spasi ( ) kemudian ditampilkan ke layar console.

Agar lebih jelas, silakan perhatikan statement berikut, keduanya adalah menghasilkan output yang sama.

```
print("message: %s %s %s" % ("hello", "python", "learner"))
print("message:", "hello", "python", "learner")
```



◉ Fungsi `id()`

Digunakan untuk mengambil nilai identitas atau ID suatu data. Contoh penerapannya sangat mudah, cukup panggil fungsi `id()` kemudian tulis data yang ingin diambil ID-nya sebagai argument pemanggilan fungsi tersebut.

```
data_1 = "hello world"
id_data_1 = id(data_1)

print("data_1:", data_1)
# output → data_1: hello world

print("id_data_1:", id_data_1)
# output → id_data_1: 19441xxxxxxxx
```

Nilai kembalian fungsi `id()` bertipe numerik.

> *Pembahasan detail mengenai fungsi `id()` ada di chapter Object ID & Reference*

A.7.7. Operator *membership* (`in`)

Operator `in` digunakan untuk mengecek apakah suatu nilai merupakan bagian dari data kolektif atau tidak.

Operator ini bisa dipergunakan pada semua tipe data kolektif seperti

dictionary, set, tuple, dan list. Selain itu, operator `in` juga bisa digunakan pada string untuk pengecekan substring

```
sample_list = [2, 3, 4]
is_3_exists = 3 in sample_list
print(is_3_exists)
# output → True

sample_tuple = ("hello", "python")
is_hello_exists = "hello" in sample_tuple
print(is_hello_exists)
# output → True

sample_dict = { "nama": "noval", "age": 12 }
is_key_nama_exists = "nama" in sample_dict
print(is_key_nama_exists)
# output → True

sample_set = { "sesuk", "preiiii" }
is_prei = "preiiii" in sample_set
print(is_prei)
# output → True

text = 'Hello world'
is_substring_exists = 'orl' in text
print(is_substring_exists)
# output → True
```

> *Operator `in` jika diterapkan pada tipe dictionary, yang di-check adalah key-nya bukan value-nya.*

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../operator
```

◉ Chapter relevan lainnya

- Variabel
- Tipe Data
- String ➜ formatting

◉ TBA

- Operator `@` for matrix multiplication

◉ Referensi

- https://realpython.com/python-operators-expressions/
- https://www.programiz.com/python-programming/operators
- https://stackoverflow.com/a/15172182/1467988

A.8. Seleksi kondisi Python ➜ if, elif, else

Seleksi kondisi adalah suatu blok kode yang dieksekusi hanya ketika kriteria yang ditentukan terpenuhi. Teknik seleksi kondisi banyak digunakan untuk kontrol alur program.

Python mengenal beberapa keyword seleksi kondisi, dan pada chapter ini akan kita pelajari.

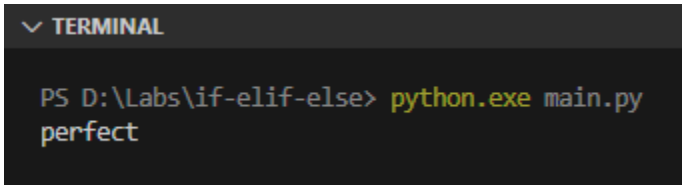
A.8.1. Keyword `if`

`if` adalah keyword seleksi kondisi di Python. Cara penerapan keyword ini sangat mudah, cukup tulis saja `if` diikuti dengan kondisi berupa nilai `bool` atau statement operasi logika, lalu dibawahnya ditulis blok kode yang ingin dieksekusi ketika kondisi tersebut terpenuhi. Contoh:

```
grade = 100

if grade == 100:
    print("perfect")

if grade == 90:
    print("ok")
    print("keep working hard!")
```



Bisa dilihat di output, hanya pesan `perfect` yang muncul karena kondisi `grade == 100` adalah yang terpenuhi. Sedangkan statement `print("ok")` tidak tereksekusi karena nilai variabel `grade` bukanlah `90`.

◉ *Block indentation*

Di python, suatu blok kondisi ditandai dengan *indentation* atau spasi, yang menjadikan kode semakin menjorok ke kanan.

Sebagai contoh, 2 blok kode `print` berikut merupakan isi dari seleksi kondisi `if grade == 90`.

```
if grade == 90:
    print("ok")
    print("keep working hard!")
```

> *Sesuai aturan PEP 8 – Style Guide for Python Code, indentation di Python menggunakan 4 karakter spasi dan bukan karakter tab.*

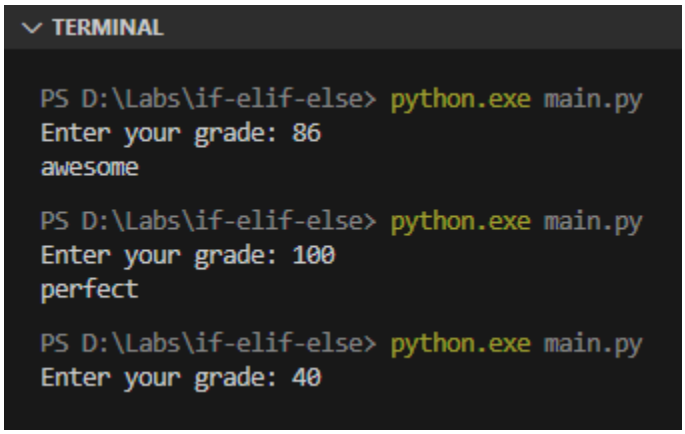
A.8.2. Keyword `elif`

`elif` (kependekan dari **else if**) digunakan untuk menambahkan blok seleksi kondisi baru, untuk mengantisipasi blok `if` yang tidak terpenuhi.

Dalam penerapannya, suatu blok seleksi kondisi harus diawali dengan `if`. Keyword `elif` hanya bisa dipergunakan pada kondisi setelahnya yang masih satu rantai (masih satu *chain*). Contoh:

```
str_input = input('Enter your grade: ')
```

Jalankan program di atas, kemudian inputkan suatu nilai numerik lalu tekan enter.



Kode di atas menghasilkan:

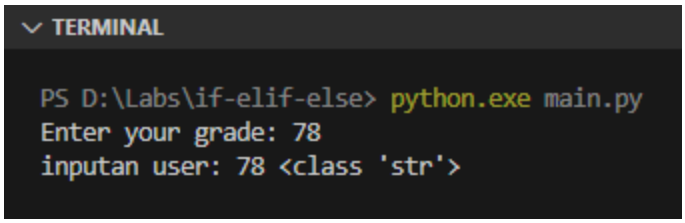
- Ketika nilai inputan adalah `86`, muncul pesan `awesome` karena blok seleksi kondisi yang terpenuhi adalah `elif grade >= 85`.
- Ketika nilai inputan adalah `100`, muncul pesan `perfect` karena blok seleksi kondisi yang terpenuhi adalah `grade == 100`.
- Ketika nilai inputan adalah `40`, tidak muncul pesan karena semua blok seleksi kondisi tidak terpenuhi.

◉ Fungsi *input()*

Fungsi input digunakan untuk menampilkan suatu pesan text (yang disisipkan saat fungsi dipanggil) dan mengembalikan nilai inputan user dalam bentuk string.

Agar makin jelas, silakan praktikan kode berikut:

```
str_input = input('Enter your grade: ')
print("inputan user:", str_input, type(str_input))
```



Kode di atas menghasilkan:

1. Text `Enter your grade :` muncul, kemudian kursor akan berhenti disitu.
2. User perlu menuliskan sesuatu kemudian menekan tombol enter agar eksekusi program berlanjut.
3. Inputan dari user kemudian menjadi nilai balik fungsi `input()` (yang pada contoh di atas ditampung oleh variabel `input_str`).
4. Nilai inputan user di print menggunakan statement `print("inputan user:", str_input)`.

◉ Fungsi `type()`

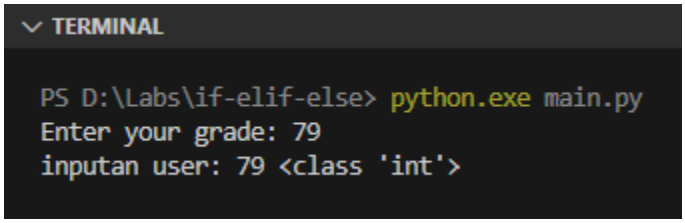
Fungsi `type()` digunakan untuk melihat informasi tipe data dari suatu nilai atau variabel. Fungsi ini mengembalikan string dalam format `<class 'tipe_data'>`.

◉ *Type conversion* / konversi tipe data

Konversi tipe data string ke `int` dilakukan menggunakan fungsi `int()`. Dengan menggunakan fungsi tersebut, data string yang disisipkan pada parameter, tipe datanya berubah menjadi `int`.

Sebagai contoh, bisa dilihat pada program berikut ini, hasil statement `type(grade)` adalah `<class 'int'>` yang menunjukan bahwa tipe datanya adalah `int`.

```
str_input = input('Enter your grade: ')
grade = int(str_input)
print("inputan user:", grade, type(grade))
```



> *Pembahasan detail mengenai type conversion ada di chapter Konversi Tipe Data*

A.8.3. Keyword `else`

`else` digunakan sebagai blok seleksi kondisi penutup ketika blok `if` dan/atau `elif` dalam satu *chain* tidak ada yang terpenuhi. Contoh:

```
str_input = input('Enter your grade: ')
grade = int(str_input)

if grade == 100:
    print("perfect")
elif grade >= 85:
    print("awesome")
elif grade >= 65:
    print("passed the exam")
else:
    print("below the passing grade")
```

A.8.4. Seleksi kondisi bercabang / *nested*

Seleksi kondisi bisa saja berada di dalam suatu blok seleksi kondisi. Teknik ini biasa disebut dengan seleksi kondisi bercabang atau bersarang.

Di Python, cara penerapannya cukup dengan menuliskan blok seleksi kondisi tersebut. Gunakan *indentation* yang lebih ke kanan untuk seleksi kondisi terdalam.

```
str_input = input('Enter your grade: ')
grade = int(str_input)

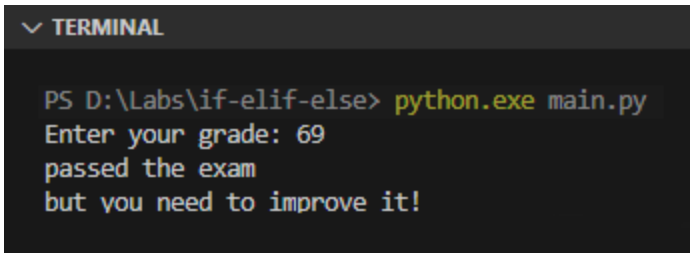
if grade == 100:
    print("perfect")

elif grade >= 85:
    print("awesome")

elif grade >= 65:
    print("passed the exam")

    if grade <= 70:
        print("but you need to improve it!")
    else:
        print("with ok grade")

else:
    print("below the passing grade")
```



> *Pada kode di atas, pada seleksi kondisi terluar, di bawah blok `if` dan `elif` sengaja penulis tulis di baris baru agar lebih mudah untuk dibaca. Hal seperti ini diperbolehkan.*

A.8.5. Seleksi kondisi dengan operasi logika

Keyword `and`, `or`, dan `not` bisa digunakan dalam seleksi kondisi. Contohnya:

```
grade = int(input('Enter your current grade: '))
prev_grade = int(input('Enter your previous grade: '))

if grade >= 90 and prev_grade >= 65:
    print("awesome")
if grade >= 90 and prev_grade < 65:
    print("awesome. you definitely working hard, right?")
elif grade >= 65:
    print("passed the exam")
else:
    print("below the passing grade")

if (grade >= 65 and not prev_grade >= 65) or (not grade >= 65 and
prev_grade >= 65):
    print("at least you passed one exam. good job!")
```

A.8.6. Seleksi kondisi sebaris & *ternary*

Silakan perhatikan kode sederhana berikut, isinya adalah seleksi kondisi sederhana pengecekan nilai `grade >= 65` atau tidak.

```
if grade >= 65:
    print("passed the exam")
else:
    print("below the passing grade")
```

Kode di atas bisa dituliskan dalam bentuk alternatif penulisan kode lainnya:

◉ *One-line* / sebaris

```
if grade >= 65: print("passed the exam")
if grade < 65: print("below the passing grade")
```

Metode penulisan sebaris ini cocok diterapkan pada situasi dimana seleksi kondisi hanya memiliki 1 kondisi saja.

◉ *Ternary*

```
print("passed the exam") if grade >= 65 else print("below the passing
grade")
```

Metode penulisan *ternary* umum diterapkan pada blok kode seleksi kondisi yang memiliki 2 kondisi (`True` dan `False`).

◉ *Ternary* dengan nilai balik

```
message = "passed the exam" if grade >= 65 else "below the passing grade"
print(message)
```

Metode penulisan ini sebenarnya adalah sama seperti penerapan ternary sebelumnya, perbedaannya: pada metode ini setiap kondisi menghasilkan nilai balik yang umumnya ditampung oleh variabel. Pada contoh di atas, nilai balik ditampung variabel `message`.

A.8.7. Pattern matching

Selain keyword `if`, Python menyediakan keyword lain yaitu `match` dan `case` yang kegunaannya adalah untuk pattern matching. Pattern matching sendiri merupakan teknik seleksi kondisi yang cukup advance, mendukung banyak variasi pencocokan pola.

Lebih detailnya mengenai pattern matching ada di chapter Pattern Matching ➜ match

---

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../if-elif-else
```

◉ Referensi

- https://docs.python.org/3/tutorial/controlflow.html

A.9. Perulangan Python ➜ for & range

Perulangan atau *loop* merupakan teknik untuk mengulang-ulang eksekusi suatu blok kode, atau mengiterasi elemen milik tipe data kolektif (contohnya: list). Chapter ini membahas tentang penerapannya di Python.

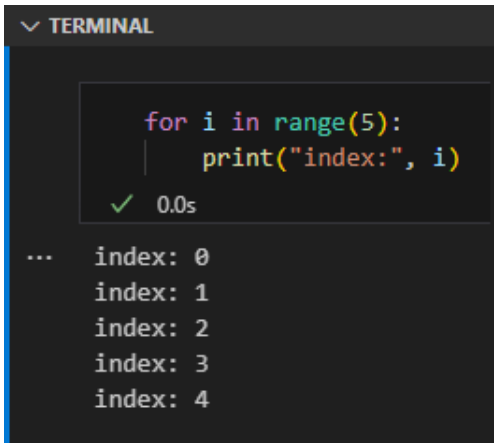
A.9.1. Keyword `for` dan fungsi `range()`

Perulangan di Python bisa dibuat menggunakan kombinasi keyword `for` dan fungsi `range()`.

- Keyword `for` adalah keyword untuk perulangan, dalam penerapannya diikuti dengan keyword `in`.
- Fungsi `range()` digunakan untuk membuat object *range*, yang umumnya dipakai sebagai kontrol perulangan.

Agar lebih jelas, silakan perhatikan dan test kode berikut:

```
for i in range(5):
    print("index:", i)
```



Penjelasan:

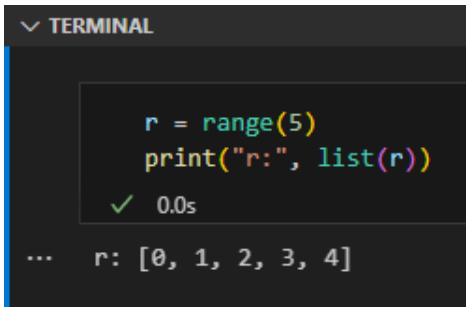
- Statement `print("index:", i)` muncul 5 kali, karena perulangan dilakukan dengan kontrol `range(5)` dimana statement tersebut menghasilkan object *range* dengan isi deret angka sejumlah `5` dimulai dari angka `0` hingga `4`.
- Statement `for i in range(5):` adalah contoh penulisan perulangan menggunakan `for` dan `range()`. Variabel `i` berisi nilai *counter* setiap iterasi, yang pada konteks ini adalah angka `0` hingga `4`.
- Statement `print("index:", i)` wajib ditulis menjorok ke kanan karena merupakan isi dari blok perulangan `for i in range(5):`.

◉ Fungsi `list()`

Fungsi `range()` menghasilkan object *sequence*, yaitu jenis data yang strukturnya mirip seperti list (tapi bukan list) yang kegunaan utamanya adalah untuk kontrol perulangan.

Object *sequence* bisa dikonversi bentuk list dengan cara dibungkus menggunakan fungsi `list()`.

```
r = range(5)
print("r:", list(r))
```



> - *Pembahasan detail mengenai list ada di chapter List*
> - *Pembahasan detail mengenai type conversion ada di chapter Konversi Tipe Data*

A.9.2. Penerapan fungsi `range()`

Statement `range(n)` menghasilkan data *range* sejumlah `n` yang isinya dimulai dari angka `0`. Syntax `range(n)` adalah bentuk paling sederhana penerapan fungsi ini.

Selain `range(n)` ada juga beberapa cara penulisan lainnya:

- Menggunakan `range(start, stop)`. Hasilnya data *range* dimulai dari `start` dan hingga `stop - 1`. Sebagai contoh, `range(1, 4)` menghasilkan data range `[1, 2, 3]`.
- Menggunakan `range(start, stop, step)`. Hasilnya data *range* dimulai dari `start` dan hingga `stop - 1`, dengan nilai *increment* sejumlah `step`. Sebagai contoh, `range(1, 10, 3)` menghasilkan data range `[1, 4, 7]`.

Agar lebih jelas, silakan perhatikan kode berikut. Ke-3 perulangan ini ekuivalen, menghasilkan output yang sama.

```
for i in range(3):
```

```
for i in range(0, 3):
```

```
for i in
```

```
TERMINAL
PS D:\Labs\for-range> python.exe main.py
index: 0
index: 1
index: 2
```

```
TERMINAL
PS D:\Labs\for-range> python.exe main.py
index: 0
index: 1
index: 2
```

```
TERMINAL
PS D:\Labs\for-range> python.exe main.py
index: 0
index: 1
index: 2
```

Tambahan contoh penerapan `for` dan `range()`:

```
for i in range(2, 10, 2):
print("index:", i)
```

```
for i in range(2, 10, 2):
    print("index:", i)
✓ 0.0s
index: 2
index: 4
index: 6
index: 8
```

```
for i in range(5, -5, -1):
print("index:", i)
```

```
for i in range(5, -5, -1):
    print("index:", i)
✓ 0.0s
index: 5
index: 4
index: 3
index: 2
index: 1
index: 0
index: -1
index: -2
index: -3
index: -4
```

A.9.3. Iterasi element data kolektif

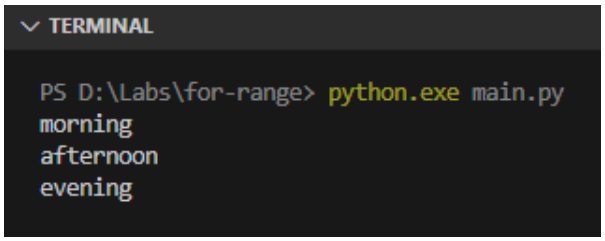
Perulangan menggunakan `for` bisa dilakukan pada beberapa jenis tipe data (seperti list, string, tuple, dan lainnya) caranya dengan langsung menuliskan saja variabel atau data tersebut pada statement `for`.

> *Tipe data yang bisa digunakan pada keyword `for` bisasa disebut dengan tipe iterator. Lebih detailnya dibahas pada chapter Iterator.*

Contoh penerapannya bisa dilihat di bawah ini:

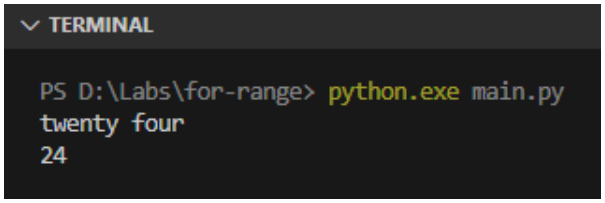
◉ Iterasi data list

```
messages = ["morning", "afternoon", "evening"]
for m in messages:
    print(m)
```



◉ Iterasi data tuple

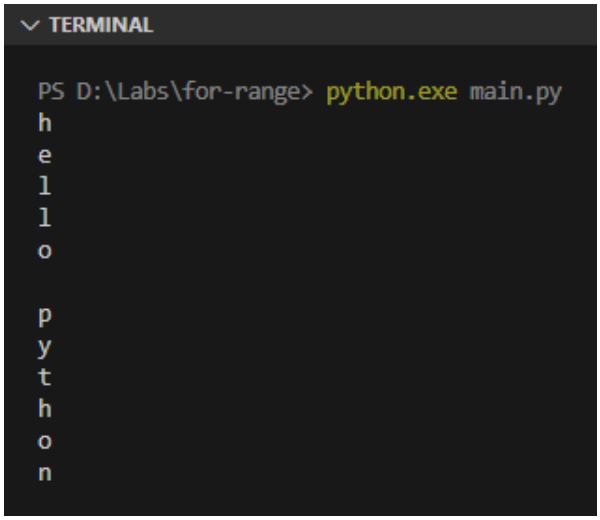
```
numbers = ("twenty four", 24)
for n in numbers:
    print(n)
```



◉ Iterasi data string

Penggunaan keyword `for` pada tipe data string akan mengiterasi setiap karakter yang ada di string.

```python
for char in "hello python":
    print(char)
```

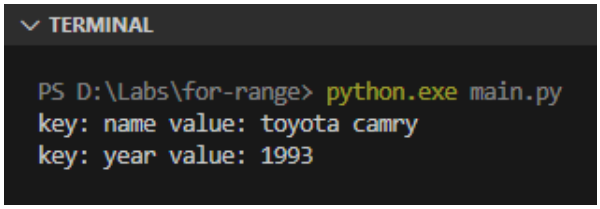


◉ Iterasi data dictionary

Penggunaan keyword `for` pada tipe data `dict` (atau dictionary) akan mengiterasi *key*-nya. Dari *key* tersebut *value* bisa diambil dengan mudah menggunakan notasi `dict[key]`.

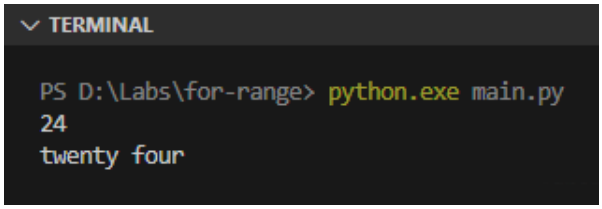
```python
bio = {
    "name": "toyota camry",
    "year": 1993,
}

for key in bio:
    print("key:", key, "value:", bio[key])
```



◉ Iterasi data set

```
numbers = {"twenty four", 24}
for n in numbers:
    print(n)
```

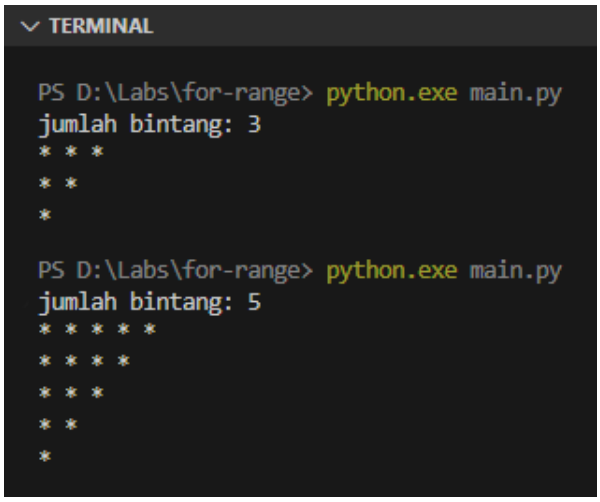


A.9.4. Perulangan bercabang / *nested* `for`

Cara penerapan *nested loop* adalah cukup dengan menuliskan statement `for` sebagai isi dari statement `for` atasnya. Contoh:

```
max = int(input("jumlah bintang: "))

for i in range(max):
    for j in range(0, max - i):
        print("*", end=" ")
    print()
```



◉ Parameter opsional `end` pada fungsi `print()`

Fungsi `print()` memiliki parameter opsional bernama `end`, kegunaannya untuk mengubah karakter akhir yang muncul setelah data string di-*print*. *Default* nilai paramter `end` ini adalah `\n` atau karakter baris baru, itulah kenapa setiap selesai print pasti ada baris baru.

Statement `print("*", end=" ")` akan menghasilkan pesan `* ` yang di-akhiri dengan karakter spasi karena nilai parameter `end` di-set dengan nilai karakter spasi (atau ` `).

> *Pembahasan detail mengenai fungsi dan parameter opsional ada di chapter berikut:*
>
> - *Function*
> - *Function → Positional, Optional, Keyword Arguments*

◉ Fungsi `print()` tanpa parameter

Pemanggilan fungsi `print()` argument/parameter menghasilkan baris baru.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../for-range
```

◉ Chapter relevan lainnya

- List
- String
- Function

◉ Referensi

- https://docs.python.org/3/library/functions.html#func-range
- https://docs.python.org/3/library/functions.html#print
- https://python-reference.readthedocs.io/en/latest/docs/functions/range.html

A.10. Perulangan Python ➜ while

Di Python, selain keyword `for` ada juga keyword `while` yang fungsinya kurang lebih sama yaitu untuk perulangan. Bedanya, perulangan menggunakan `while` terkontrol via operasi logika atau nilai `bool`.

Pada chapter ini kita akan mempelajari cara penerapannya.

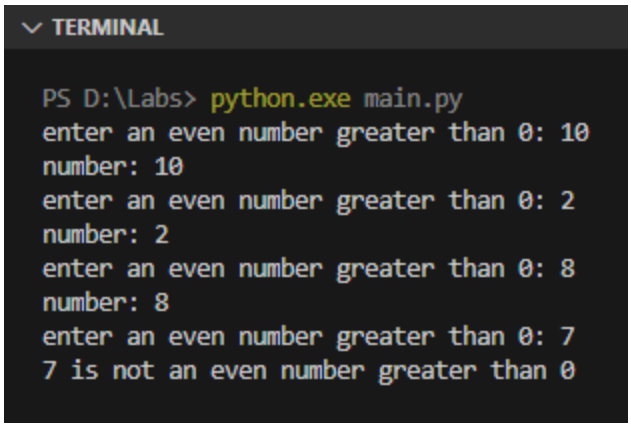
A.10.1. Keyword `while`

Cara penerapan perulangan ini adalah dengan menuliskan keyword `while` kemudian diikuti dengan nilai `bool` atau operasi logika. Contoh:

```
should_continue = True

while should_continue:
    n = int(input("enter an even number greater than 0: "))

    if n <= 0 or n % 2 == 1:
        print(n, "is not an even number greater than 0")
        should_continue = False
    else:
        print("number:", n)
```

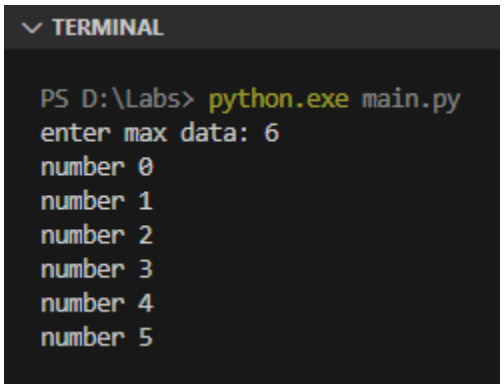


Program di atas memunculkan *prompt* inputan `enter an even number greater than 0:` yang dimana akan terus muncul selama user tidak menginputkan angka ganjil atau angka dibawah sama dengan `0`.

Contoh lain penerapan `while` dengan kontrol adalah operasi logika:

```
n = int(input("enter max data: "))
i = 0

while i < n:
    print("number", i)
    i += 1
```



◉ Operasi *increment* dan *decrement*

Python tidak mengenal operator *unary* `++` dan `--`. Solusi untuk melakukan operasi *increment* maupun *decrement* bisa menggunakan cara berikut:

Operasi	Cara 1	Cara 2
Increment	`i += 1`	`i = i + 1`
Decrement	`i -= 1`	`i = i - 1`

A.10.2. Perulangan `while` vs `for`

Operasi `while` cocok digunakan untuk perulangan yang dimana kontrolnya adalah operasi logika atau nilai boolean yang tidak ada kaitannya dengan *sequence*.

Pada program yang sudah di tulis di atas, perulangan akan menjadi lebih ringkas dengan pengaplikasian keyword `for`, silakan lihat perbandingannya di bawah ini:

- Dengan keyword `while`:

```
n = int(input("enter max data: "))
i = 0

while i < n:
    print("number", i)
    i += 1
```

- Dengan keyword `for`:

```
n = int(input("enter max data: "))

for i in range(n):
    print("number", i)
```

Sedangkan keyword `for` lebih pas digunakan pada perulangan yang kontrolnya adalah data *sequence*, contohnya seperti range dan list.

A.10.3. Perulangan bercabang / *nested* `while`

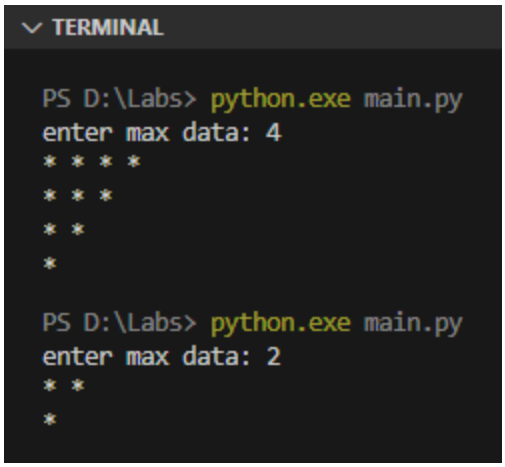
Contoh perulangan bercabang bisa dilihat pada kode program berikut ini. Caranya cukup tulis saja keyword `while` di dalam block kode `while`.

```
n = int(input("enter max data: "))
i = 0

while i < n:
    j = 0

    while j < n - i:
        print("*", end=" ")
        j += 1

    print()
    i += 1
```



A.10.4. Kombinasi `while` dan `for`

Kedua keyword perulangan yang sudah dipelajari, yaitu `for` dan `while` bisa dikombinasikan untuk membuat suatu *nested loop* atau perulangan bercabang.

Pada contoh berikut, kode program di atas diubah menggunakan kombinasi keyword `for` dan `while`.

```
n = int(input("enter max data: "))
i = 0

for i in range(n):
    j = 0

    while j < n - i:
        print("*", end=" ")
        j += 1

    print()
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../while
```

◉ Referensi

- https://docs.python.org/3/tutorial/controlflow.html

A.11. Perulangan Python ➜ break, continue

Keyword `break` dan `continue` sering dipergunakan dalam perulangan untuk alterasi flow secara paksa, seperti memberhentikan perulangan atau memaksa perulangan untuk lanjut ke iterasi berikutnya.

Pada chapter ini kita akan mempelajarinya.

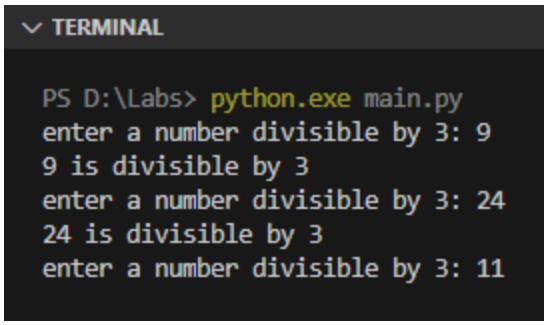
A.11.1. Keyword `break`

Pengaplikasian `break` biasanya dikombinasikan dengan seleksi kondisi. Sebagai contoh program sederhana berikut, yaitu program dengan spesifikasi:

- Berisi perulangan yang sifatnya berjalan terus-menerus tanpa henti (karena menggunakan nilai `True` sebagai kontrol).
- Perulangan hanya berhenti jika nilai `n` (yang didapat dari inputan user) adalah tidak bisa dibagi dengan angka `3`.

```
while True:
    n = int(input("enter a number divisible by 3: "))
    if n % 3 != 0:
        break

    print("%d is divisible by 3" % (n))
```



A.11.2. Keyword `continue`

Keyword `continue` digunakan untuk memaksa perulangan lanjut ke iterasi berikutnya (seperti proses skip).

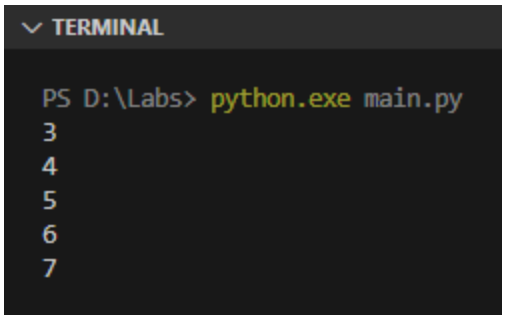
Contoh penerapannya bisa dilihat pada program berikut, yang spesifikasinya:

- Program berisi perulangan dengan kontrol adalah data *range* sebanyak 10 (dimana isinya adalah angka numerik `0` hingga `9`).
- Ketika nilai variabel counter `i` adalah dibawah `3` atau di atas `7` maka iterasi di-skip.

```
for i in range(10):
    if i < 3 or i > 7:
        continue
    print(i)
```

Efek dari `continue` adalah semua statement setelahnya akan di-skip. Pada program di atas, statement `print(i)` tidak dieksekusi ada `continue`.

Hasilnya bisa dilihat pada gambar berikut, nilai yang di-print adalah angka `3` hingga `7` saja.



A.11.3. Label perulangan

Python tidak mengenal konsep perulangan yang memiliki label.

Teknik menamai perulangan dengan label umumnya digunakan untuk mengontrol flow pada perulangan bercabang / *nested*, misalnya untuk menghentikan perulangan terluar secara paksa ketika suatu kondisi terpenuhi.

Di Python, algoritma seperti ini bisa diterapkan namun menggunakan tambahan kode. Contoh penerapannya bisa dilihat pada kode berikut:

```
max = int(input("jumlah bintang: "))

outer_loop = True
for i in range(max):
    if not outer_loop:
        break

    for j in range(i + 1):
        print("*", end=" ")
        if j >= 7:
            outer_loop = False
            break
    print()
```

Penjelasan:

- Program yang memiliki perulanga *nested* dengan jumlah perulangan ada 2.
- Disiapkan sebuah variabel `bool` bernama `outer_loop` untuk kontrol perulangan terluar.
- Ketika nilai `j` (yang merupakan variabel counter perulangan terdalam) adalah lebih dari atau sama dengan `7`, maka variabel `outer_loop` di set nilainya menjadi `False`, dan perulangan terdalam di-`break` secara paksa.
- Dengan ini maka perulangan terluar akan terhenti.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../break-continue
```

◉ Chapter relevan lainnya

- Perulangan ➜ for & range
- Perulangan ➜ while

◉ Referensi

- https://docs.python.org/3/tutorial/controlflow.html

A.12. Python List

List adalah tipe data kolektif yang disimpan secara urut dan bisa diubah nilainya (istilah lainnya adalah tipe data *sequence*).

Pada bahasa pemrograman umumnya ada tipe data **array**. List di Python ini memiliki banyak kemiripan dengan array, bedanya list bisa berisi data dengan berbagai macam tipe data, jadi tidak harus sejenis tipe datanya.

Pada chapter ini kita akan belajar lebih detail mengenai list dan pengoperasiannya.

A.12.1. Pengenalan list

Deklarasi variabel dan data list adalah menggunakan *literal* list dengan notasi penulisan seperti berikut:

```
# contoh list
list_1 = [10, 70, 20]

# list dengan deklarasi element secara vertikal
list_2 = [
    'ab',
    'cd',
    'hi',
    'ca'
]

# list dengan element berisi bermacam-macam tipe data
list_3 = [3.14, 'hello python', True, False]

# list kosong
list_4 = []
```

Data dalam list biasa disebut dengan **element**. Setiap elemen disimpan dalam list secara urut dengan penanda urutan yang disebut **index**. Nilai index dimulai dari angka `0`.

Sebagai contoh, pada variabel `list_1` di atas:

- Element index ke-`0` adalah data `10`
- Element index ke-`1` adalah data `70`
- Element index ke-`2` adalah data `20`

A.12.2. Perulangan list

List adalah salah satu tipe data yang dapat digunakan langsung pada perulangan `for`. Contoh:

```
list_1 = [10, 70, 20]

for e in list_1:
    print("elem:", e)
```

Selain itu, perulangan list bisa juga dilakukan menggunakan index, contohnya seperti berikut:

```
list_1 = [10, 70, 20]
for i in range(0, len(list_1)):
    print("index:", i, "elem:", list_1[i])
```

Fungsi `len()` digunakan untuk menghitung jumlah element list. Dengan mengkombinasikan nilai balik fungsi ini dan fungsi `range()` bisa terbentuk data range dengan lebar sama dengan lebar list.

◉ Fungsi `enumerate()`

Fungsi `enumerate()` digunakan untuk membuat data sequence menjadi data enumerasi, yang jika dimasukan ke perulangan di setiap iterasinya bisa kita akses index beserta element-nya.

```
list_1 = [10, 70, 20]

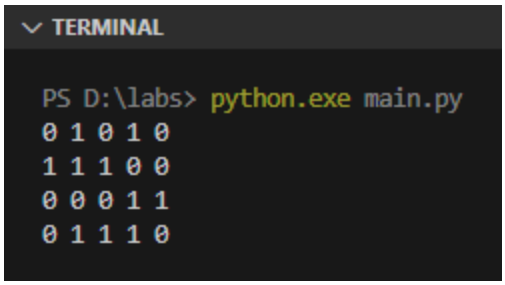
for i, v in enumerate(list_1):
    print("index:", i, "elem:", v)
```

A.12.3. Nested list

Penulisan nested list cukup mudah, contohnya bisa dilihat pada program matrix berikut:

```
matrix = [
    [0, 1, 0, 1, 0],
    [1, 1, 1, 0, 0],
    [0, 0, 0, 1, 1],
    [0, 1, 1, 1, 0],
]

for row in matrix:
    for cel in row:
        print(cel, end=" ")
    print()
```



A.12.4. Fungsi `list()`

◉ Konversi range ke list

Data range (hasil pemanggilan fungsi `range()`) bisa dikonversi ke bentuk list menggunakan fungsi `list()`. Cara ini cukup efisien untuk pembuatan data list yang memiliki *pattern* atau pola. Sebagai contoh:

- List dimulai angka `0` hingga `9`:

```
range_1 = range(0, 10)
list_1 = list(range_1)
print(list_1)
# output → [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

- List dimulai angka `1` hingga `21` dengan penambahan `3`:

```
range_2 = range(0, 22, 3)
list_2 = list(range_2)
print(list_2)
# output → [0, 3, 6, 9, 12, 15, 18, 21]
```

- List dimulai angka `100` hingga `0` dengan pengurangan `-10`:

```
range_3 = range(100, 0, -10)
list_3 = list(range_3)
print(list_3)
# output → [100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
```

> *Selain metode ini, ada juga cara lainnya untuk membuat list, yaitu menggunakan metode list comprehension, yang akan dibahas pada chapter berikutnya, yaitu List Comprehension*

◉ Konversi string ke list

Selain untuk konversi data range ke list, fungsi `list()` bisa digunakan untuk konversi data string ke list, dengan hasil adalah setiap karakter string menjadi element list.

```
alphabets = list('abcdefgh')
print(alphabets)
# output → ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
```

◉ Konversi tuple ke list

Tipe data tuple bisa diubah bentuknya menjadi list dengan menggunakan fungsi `list()`. Contoh penerapannya:

```
tuple_1 = (1, 2, 3, 4)
numbers = list(tuple_1)
print(numbers)
# output → [1, 2, 3, 4]
```

> *Pembahasan detail mengenai tuple ada di chapter Tuple*

A.12.5. Operasi pada list

◉ Mengakses element via index

Nilai elemen list bisa diakses menggunakan notasi `list[index]`. Contoh:

```
list_1 = [10, 70, 20]

elem_1st = list_1[0]
elem_2nd = list_1[1]
elem_3rd = list_1[2]

print(elem_1st, elem_2nd, elem_3rd)
# output → [10, 70, 20]
```

> 🔥 **DANGER**
>
> Pengaksesan elemen menggunakan index di-luar kapasitas data akan menghasilkan error.
>
> Sebagai contoh, data `list_1` di atas jika diakses index ke-3-nya misalnya (`list_1[3]`) hasilnya adalah error.

◉ Mengecek apakah element ada

Kombinasi keyword `if` dan `in` bisa digunakan untuk mengidentifikasi apakah suatu element merupakan bagian dari list atau tidak. Contoh penerapannya:

```
list_1 = [10, 70, 20]
n = 70

if n in list_1:
    print(n, "is exists")
else:
    print(n, "is NOT exists")

# output → 70 is exists
```

◉ *Slicing* list

Slicing adalah metode pengaksesan list menggunakan notasi slice. Notasi ini mirip seperti array, namun mengembalikan data bertipe tetap slice.

Contoh pengaplikasian metode slicing bisa dilihat pada kode berikut. Variabel `list_2` diakses element-nya mulai index `1` hingga sebelum `3`:

```
list_2 = ['ab', 'cd', 'hi', 'ca']
print('list_2:', list_2)
# output → list2: ['ab', 'cd', 'hi', 'ca']

slice_1 = list_2[1:3]
print('slice_1:', slice_1)
# output → slice_1: ['cd', 'hi']
```

> *Pembahasan detail mengenai slice ada di chapter Slice*

◉ Mengubah nilai element

Cara mengubah nilai element list dengan cara mengakses nilai element menggunakan index, kemudian diikuti operator assignment `=` dan nilai baru.

```
list_2 = ['ab', 'cd', 'hi', 'ca']
print('before:', list_2)
# output → before: ['ab', 'cd', 'hi', 'ca']

list_2[1] = 'zk'
list_2[2] = 'sa'
print('after: ', list_2)
# output → after: ['ab', 'zk', 'sa', 'ca']
```

◉ Append element

Operasi *append* atau menambahkan element baru setelah index terakhir, bisa menggunakan 2 cara:

- via method `append()`:

```
list_1 = [10, 70, 20]
print('before: ', list_1)
# output → before: [10, 70, 20]

list_1.append(88)
list_1.append(87)
print('after: ', list_1)
# output → after : [10, 70, 20, 88, 87]
```

- via slicing:

```
list_1 = [10, 70, 20]
print('before: ', list_1)
# output → before: [10, 70, 20]

list_1[len(list_1):] = [88, 87]
print('after: ', list_1)
# output → after : [10, 70, 20, 88, 87]
```

> *Pembahasan detail mengenai method ada 4 chapter berikut:*
>
> - *OOP ➜ Instance Method*
> - *OOP ➜ Class Method*
> - *OOP ➜ Static Method*
> - *OOP ➜ Abstract Method*

◉ *Extend/concat/union* element

Operasi *extend* (atau *concat* atau *union*) adalah operasi penggabungan dua data list. Ada beberapa metode yang tersedia, diantaranya:

- via method `extend()`:

```
list_1 = [10, 70, 20]
list_2 = [88, 77]
list_1.extend(list_2)
print(list_1)
# output ➜ [10, 70, 20, 88, 87]
```

- via slicing:

```
list_1 = [10, 70, 20]
list_2 = [88, 77]
list_1[len(list_1):] = list_2
print(list_1)
# output ➜ [10, 70, 20, 88, 87]
```

- via operator `+`:

```
list_1 = [10, 70, 20]
list_2 = [88, 77]
list_3 = list_1 + list_2
print(list_3)
# output → [10, 70, 20, 88, 87]
```

Metode extend menggunakan operator `+` mengharuskan hasil operasi untuk ditampung ke variabel.

◉ Menyisipkan element pada index `i`

Method `insert()` digunakan untuk menyisipkan element baru pada posisi index tertentu (misalnya index `i`). Hasil operasi ini membuat semua element setelah index tersebut posisinya bergeser ke kanan.

Pada penggunaannya, para parameter pertama diisi dengan posisi index, dan parameter ke-2 diisi nilai.

```
list_3 = [10, 70, 20, 70]

list_3.insert(0, 15)
print(list_3)
# output → [15, 10, 70, 20, 70]

list_3.insert(2, 25)
print(list_3)
# output → [15, 10, 25, 70, 20, 70]
```

- Variabel `list_3` awalnya berisi `[10, 70, 20, 70]`
- Ditambahkan angka `15` pada index `0`, hasilnya nilai `list_3` sekarang adalah `[15, 10, 70, 20, 70]`
- Ditambahkan lagi, angka `25` pada index `2`, hasilnya nilai `list_3`

sekarang adalah `[15, 10, 25, 70, 20, 70]`

◉ Menghapus element

Method `remove()` digunakan untuk menghapus element. Isi parameter fungsi dengan element yang ingin di hapus.

Jika element yang ingin dihapus ditemukan ada lebih dari 1, maka yang dihapus hanya yang pertama (sesuai urutan index).

```
list_3 = [10, 70, 20, 70]

list_3.remove(70)
print(list_3)
# output → [10, 20, 70]

list_3.remove(70)
print(list_3)
# output → [10, 20]
```

◉ Menghapus element pada index `i`

Method `pop()` berfungsi untuk menghapus element pada index tertentu. Jika tidak ada index yang ditentukan, maka data element terakhir yang dihapus.

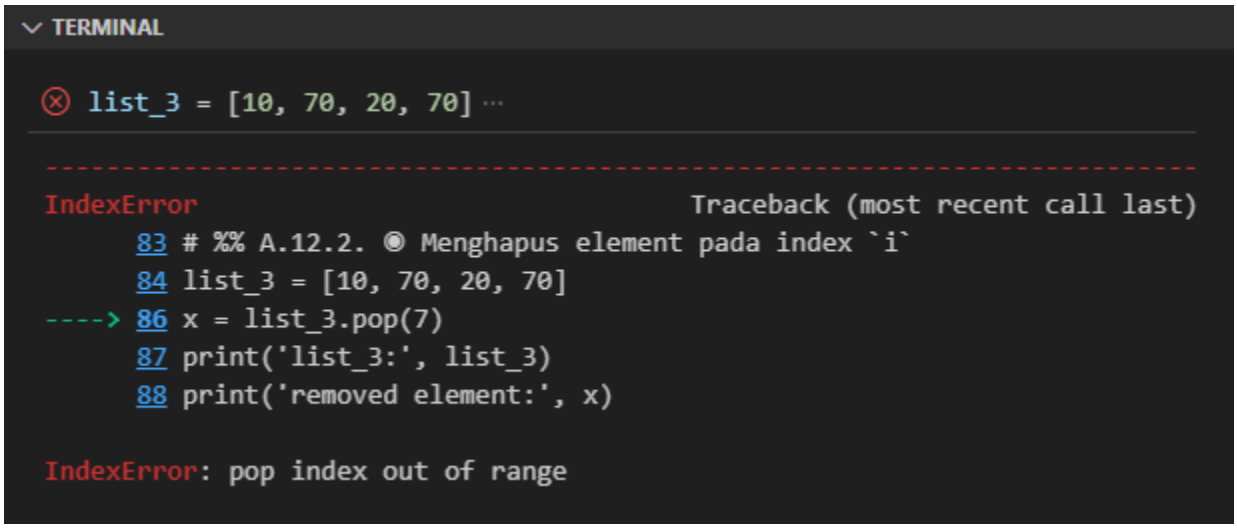
Method `pop()` mengembalikan data element yang berhasil dihapus.

```
list_3 = [10, 70, 20, 70]

x = list_3.pop(2)
print('list_3:', list_3)
# output → list_3: [10, 70, 70]
print('removed element:', x)
```

Jika index `i` yang ingin dihapus tidak diketemukan, maka error `IndexError` muncul.

```
list_3 = [10, 70, 20, 70]
x = list_3.pop(7)
```



> - *Lebih detailnya mengenai error dibahas pada chapter Error*

Selain menggunakan method `pop()`, keyword `del` bisa difungsikan untuk hal yang sama, yaitu menghapus elemen tertentu. Contoh penerapannya:

```
list_3 = [10, 70, 20, 70]
print('len:', len(list_3), "data:", list_3)

del list_3[1]
print('len:', len(list_3), "data:", list_3)
```

◉ Menghapus element pada range index

Python memiliki keyword `del` yang berguna untuk menghapus suatu data.

Dengan menggabungkan keyword ini dan operasi slicing, kita bisa menghapus element dalam range tertentu dengan cukup mudah.

Contoh, menghapus element pada index `1` hingga sebelum `3`:

```
list_3 = [10, 70, 20, 70]

del list_3[1:3]
print(list_3)
# output → [10, 70]
```

◉ Menghitung jumlah element

Fungsi `len()` digunakan untuk menghitung jumlah element.

```
list_3 = [10, 70, 20, 70]
total = len(list_3)
print(total)
# output → 4
```

Selain fungsi `len()`, ada juga method `count()` milik method slice yang kegunaannya memiliki kemiripan. Perbedaannya, method `count()` melakukan operasi pencarian sekaligus menghitung jumlah element yang ditemukan.

Agar lebih jelas, silakan lihat kode berikut:

```
list_3 = [10, 70, 20, 70]
count = list_3.count(70)
print('jumlah element dengan data `70`:', count)
# output → jumlah element dengan data `70`: 2
```

◉ Mencari index element list

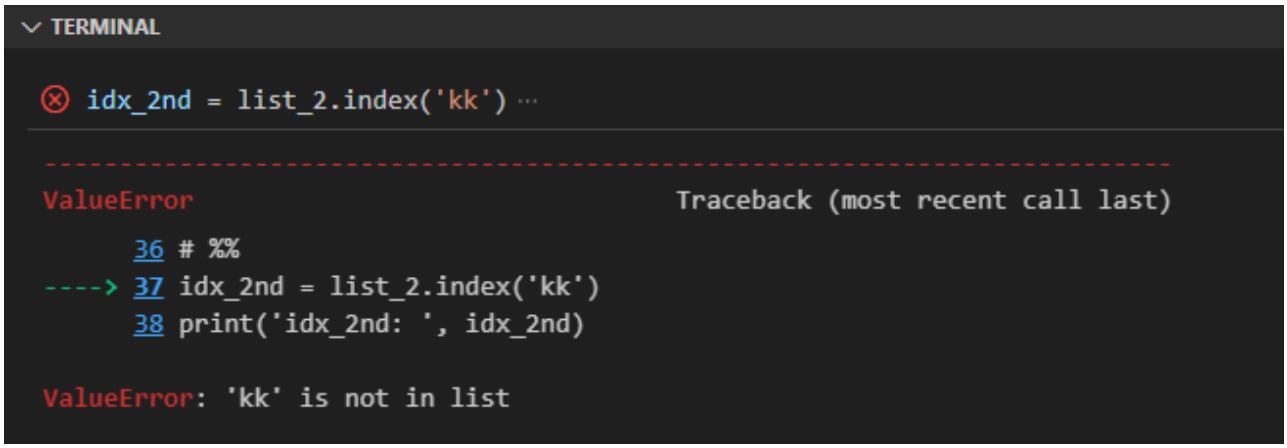
Untuk mencari index menggunakan nilai element, gunakan method `index()` milik list. Contoh bisa dilihat berikut, data `cd` ada dalam list pada index `1`.

```
list_2 = ['ab', 'cd', 'hi', 'ca']

idx_1st = list_2.index('cd')
print('idx_1st: ', idx_1st)
# output → idx_1st: 1
```

Jika data element yang dicari tidak ada, maka akan muncul error `ValueError`:

```
idx_2nd = list_2.index('kk')
print('idx_2nd: ', idx_2nd)
```



◉ Mengosongkan list

Ada dua cara untuk mengosongkan list:

- via method `clear()`:

```
list_1 = [10, 70, 20]
list_1.clear()
print(list_1)
# output → []
```

- Menimpanya dengan `[]`:

```
list_1 = [10, 70, 20]
list_1 = []
print(list_1)
# output → []
```

- Menggunakan keyword `del` dan slicing:

```
list_1 = [10, 70, 20]
del list_1[:]
print(list_1)
# output → []
```

◉ Membalik urutan element list

Method `reverse()` digunakan untuk membalik posisi element pada list.

```
list_1 = [10, 70, 20]
list_1.reverse()
print(list_1)
# output → [20, 70, 10]
```

◉ *Copy* list

Ada 2 cara untuk menduplikasi list, menggunakan method `copy()` dan teknik

slicing.

- Menggunakan method `copy()`:

```
list_1 = [10, 70, 20]
list_2 = list_1.copy()
print(list_1)
# output → [10, 70, 20]
print(list_2)
# output → [10, 70, 20]
```

- Kombinasi operasi assignment dan slicing:

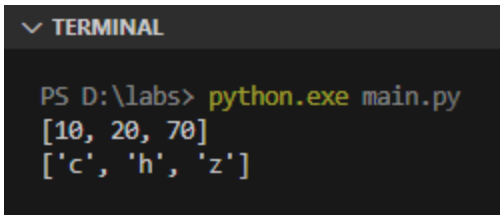
```
list_1 = [10, 70, 20]
list_2 = list_1[:]
print(list_1)
# output → [10, 70, 20]
print(list_2)
# output → [10, 70, 20]
```

> *Operasi copy disini jenisnya adalah shallow copy.*
> *Lebih detailnya mengenai shallow copy vs deep copy dibahas pada chapter terpisah.*

◉ Sorting

Mengurutkan data list bisa dilakukan menggunakan *default sorter* milik Python, yaitu method `sort()`.

```
list_1 = [10, 70, 20]
list_1.sort()
```



Method ini sebenarnya menyediakan kapasitas sorting yang cukup advance, caranya dengan cara menambahkan closure/lambda pada argument method ini.

> *Pembahasan detail mengenai closure ada di chapter Function ➜ Closure*
> *Pembahasan detail mengenai lambda ada di chapter Function ➜ Lambda*

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../list
```

◉ Chapter relevan lainnya

- Perulangan ➜ for & range
- List Comprehension
- Slice
- Function ➜ Closure
- Function ➜ Lambda

◉ Referensi

- https://docs.python.org/3/tutorial/datastructures.html

- https://docs.python.org/3/library/stdtypes.html#typesseq

A.13. Python List Comprehension

List comprehension adalah metode ringkas pembuatan list (selain menggunakan literal `[]` atau menggunakan fungsi `list()`). Cara ini lebih banyak diterapkan untuk operasi list yang menghasilkan struktur baru.

Pada chapter ini kita akan mempelajarinya.

A.13.1. Pengenalan list comprehension

Metode penulisan list comprehension membuat kode menjadi sangat ringkas, dengan konsekuensi agak sedikit membingungkan untuk yang belum terbiasa. Jadi penulis sarankan gunakan sesuai kebutuhan.

Silakan pelajari contoh berikut agar lebih mudah memahami seperti apa itu *list comprehension*.

◉ Contoh #1

Perulangan berikut:

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

print(seq)
# output → [0, 2, 4, 6, 8]
```

... bisa dituliskan lebih ringkas menggunakan *list comprehension*, menjadi

seperti berikut:

```
seq = [i * 2 for i in range(5)]

print(seq)
# output → [0, 2, 4, 6, 8]
```

◉ Contoh #2

Perulangan berikut:

```
seq = []
for i in range(10):
    if i % 2 == 1:
        seq.append(i)

print(seq)
# output → [1, 3, 5, 7, 9]
```

... bisa dituliskan lebih ringkas menjadi seperti berikut:

```
seq = [i for i in range(10) if i % 2 == 1]

print(seq)
# output → [1, 3, 5, 7, 9]
```

◉ Contoh #3

Perulangan berikut:

```
seq = []
for i in range(1, 10):
```

... bisa dituliskan lebih ringkas menjadi dengan bantuan *ternary* menjadi seperti ini:

```
seq = []
for i in range(1, 10):
    seq.append(i * (2 if i % 2 == 0 else 3))

print(seq)
# output → [3, 4, 9, 8, 15, 12, 21, 16, 27]
```

... dan bisa dijadikan lebih ringkas lagi menggunakan *list comprehension*:

```
seq = [(i * (2 if i % 2 == 0 else 3)) for i in range(1, 10)]

print(seq)
# output → [3, 4, 9, 8, 15, 12, 21, 16, 27]
```

◉ Contoh #4

Perulangan berikut:

```
list_x = ['a', 'b', 'c']
list_y = ['1', '2', '3']

seq = []
for x in list_x:
    for y in list_y:
        seq.append(x + y)

print(seq)
# output → ['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3']
```

... bisa dituliskan lebih ringkas menjadi seperti berikut:

```
seq = [x + y for x in list_x for y in list_y]

print(seq)
# output → ['a1', 'a2', 'a3', 'b1', 'b2', 'b3', 'c1', 'c2', 'c3']
```

◉ Contoh #5

Perulangan berikut:

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

transposed = []
for i in range(4):
    tr = []
    for row in matrix:
        tr.append(row[i])
    transposed.append(tr)

print(transposed)
# output → [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

... bisa dituliskan lebih ringkas menjadi seperti ini:

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

transposed = []
```

... dan bisa dijadikan lebih ringkas lagi menggunakan *list comprehension*:

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

transposed = [[row[i] for row in matrix] for i in range(4)]

print(transposed)
# output → [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Catatan chapter 🗒️

◉ Source code praktik

`github.com/novalagung/dasarpemrogramanpython-example/../list-comprehension`

◉ Chapter relevan lainnya

- Perulangan ➜ for & range
- List
- Generator & Yield

◉ TBA

- Stack vs Queue

◉ Referensi

- https://docs.python.org/3/tutorial/datastructures.html
- https://docs.python.org/3/library/stdtypes.html#typesseq

A.14. Python Tuple

Tuple adalah tipe data sequence yang ideal digunakan untuk menampung nilai kolektif yang isinya tidak akan berubah (*immutable*), berbeda dengan list yang lebih cocok untuk data yang bisa berubah nilai elemen-nya (*mutable*).

Pada chapter ini kita akan belajar tentang topik ini.

A.15.1. Tuple vs. List

Tipe data tuple sekilas memiliki beberapa kemiripan dan juga perbedaan jika dibandingkan dengan list.

	Tuple	**List**
Literal	`()`, atau `tuple()`, atau elemen ditulis tanpa `()`	`[]`, atau `list()`

Contoh	`x = ()`
	`x = tuple()`
	`x = (1, True, "h", 2, 1)`
	`x = 1, True, "h", 2, 1`
		`x = []`
		`x = list()`
		`x = [1, True, "h", 2, 1]`

Urutan elemen	urut sesuai index	

	Tuple	List
Pengaksesan elemen	via index dan perulangan	
Mutability	elemen tidak bisa diubah	elemen bisa diubah
Duplikasi elemen	elemen bisa duplikat	
Tipe data elemen	bisa sejenis maupun berbeda satu sama lain	

A.14.2. Pengenalan Tuple

Deklarasi tuple menggunakan literal `()` dengan delimiter tanda koma (`,`). Contoh syntax-nya bisa dilihat pada kode berikut:

```
tuple_1 = (2, 3, 4, "hello python", False)

print("data:", tuple_1)
# output → data: (2, 3, 4, "hello python", False)

print("total elem:", len(tuple_1))
# output → total elem: 5
```

- Tuple bisa menampung element yang tipe datanya bisa sejenis bisa tidak, sama seperti list.
- Fungsi `len()` digunakan untuk menghitung lebar tuple.

A.14.3. Mengakses element tuple via index

Element tuple bisa diakses menggunakan notasi `tuple[index]`.

```
tuple_1 = (2, 3, 4, 5)

print("elem 0:", tuple_1[0])
# output → elem 0: 2

print("elem 1:", tuple_1[1])
# output → elem 1: 3
```

DANGER

Pengaksesan elemen menggunakan index di-luar kapasitas data akan menghasilkan error.

Sebagai contoh, data `tuple_1` di atas jika diakses index ke-4-nya misalnya (`tuple_1[4]`) hasilnya adalah error.

```
tuple_1 = (2, 3, 4, 5)

print("elem 0:", tuple_1[5])
# output → IndexError: tuple index out of range
```

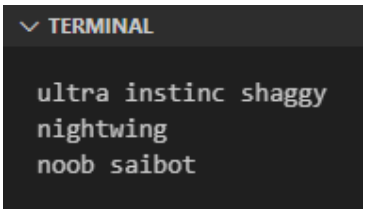
A.14.4. Perulangan tuple

Tuple adalah salah satu tipe data yang bisa digunakan secara langsung pada perulangan menggunakan keyword `for`.

Pada contoh berikut, variabel `tuple_2` dimasukan ke blok perulangan. Di setiap iterasinya, variabel `t` berisi element tuple.

```
tuple_2 = ('ultra instinc shaggy', 'nightwing', 'noob saibot')

for t in tuple_2:
    print(t)
```



Perulangan di atas ekuivalen dengan perulangan berikut:

```
tuple_2 = ('ultra instinc shaggy', 'nightwing', 'noob saibot')

for i in range(0, len(tuple_2)):
    print("index:", i, "elem:", tuple_2[i])
```

◉ Fungsi `enumerate()`

Fungsi `enumerate()` digunakan untuk membuat data sequence menjadi data enumerasi, yang jika dimasukan ke perulangan di setiap iterasinya bisa kita akses index beserta element-nya.

```
tuple_2 = ('ultra instinc shaggy', 'nightwing', 'noob saibot')

for i, v in enumerate(tuple_2):
    print("index:", i, "elem:", v)
```

A.14.5. Mengecek apakah element ada

Kombinasi keyword `if` dan `in` bisa digunakan untuk mengidentifikasi apakah suatu element merupakan bagian dari tuple atau tidak. Contoh penerapannya:

```
tuple_1 = (10, 70, 20)
n = 70

if n in tuple_1:
    print(n, "is exists")
else:
    print(n, "is NOT exists")

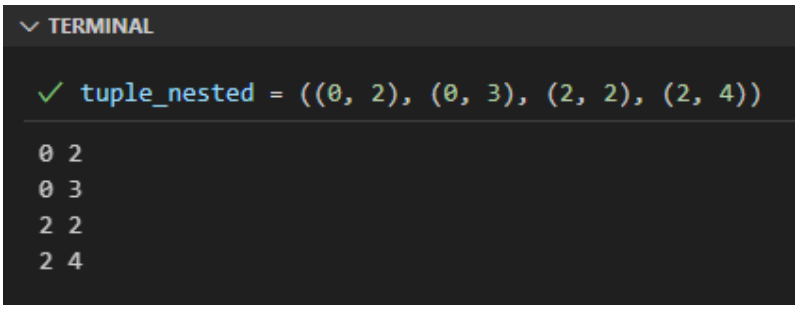
# output → 70 is exists
```

A.14.6. Nested tuple

Nested tuple dibuat dengan menuliskan data tuple sebagai element tuple. Contoh:

```
tuple_nested = ((0, 2), (0, 3), (2, 2), (2, 4))

for row in tuple_nested:
    for cell in row:
        print(cell, end=" ")
    print()
```



Penulisan data literal nested tuple bisa dalam bentuk horizontal maupun vertikal. Perbandingannya bisa dilihat pada kode berikut:

```
# horizontal
tuple_nested = ((0, 2), (0, 3), (2, 2), (2, 4))
```

A.14.7. List dan tuple

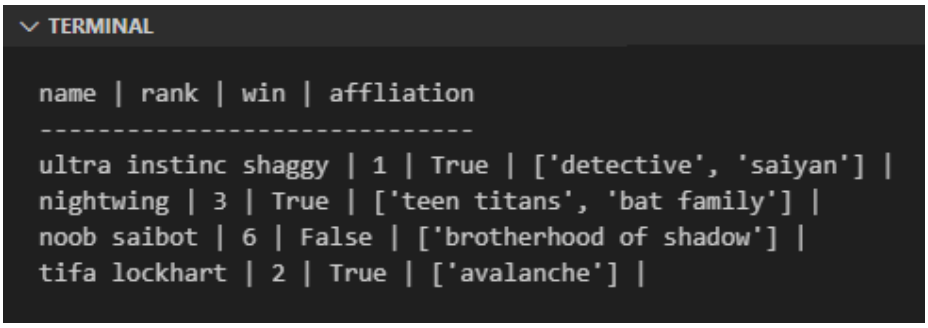
Tipe data list dan tuple umum dikombinasikan. Keduanya sangat mirip tapi memiliki perbedaan yang jelas, yaitu nilai tuple tidak bisa dimodifikasi sedangkan list bisa.

```
# deklarasi data list berisi elemen tuple
data = [
    ("ultra instinc shaggy", 1, True, ['detective', 'saiyan']),
    ("nightwing", 3, True, ['teen titans', 'bat family']),
]

# append tuple ke list
data.append(("noob saibot", 6, False, ['brotherhood of shadow']))

# append tuple ke list
data.append(("tifa lockhart", 2, True, ['avalanche']))

# print data
print("name | rank | win | affliation")
print("-----------------------------")
for row in data:
    for cell in row:
        print(cell, end=" | ")
    print()
```



A.14.8. Fungsi `tuple()`

◉ Konversi string ke tuple

Fungsi `tuple()` bisa digunakan untuk konversi data string ke tuple. Hasilnya adalah nilai tuple dengan element berisi setiap karakter yang ada di string. Contoh:

```
alphabets = tuple('abcdefgh')
print(alphabets)
# output → ('a', 'b', 'c', 'd', 'e', 'f', 'g', 'h')
```

◉ Konversi list ke tuple

Konversi list ke tuple bisa dilakukan dengan mudah menggunakan fungsi `tuple()`. Contoh penerapannya:

```
numbers = tuple([2, 3, 4, 5])
print(numbers)
# output → (2, 3, 4, 5)
```

◉ Konversi range ke tuple

Range juga bisa dikonversi ke tuple menggunakan fungsi `tuple()`.

```
r = range(0, 3)
rtuple = tuple(r)
print(rtuple)
# output → (0, 1, 2)
```

A.14.9. Tuple *packing* dan *unpacking*

◉ Tuple *packing*

Packing adalah istilah untuk menggabungkan beberapa data menjadi satu data kolektif. Contoh pengaplikasiannya bisa dilihat pada program berikut, ada 3 variabel dengan isi berbeda di-*pack* menjadi satu data tuple.

```
first_name = "aerith gainsborough"
rank = 11
win = False

row_data = (first_name, rank, win)

print(row_data)
# output → ('aerith gainsborough', 11, False)
```

Bisa dilihat penerapan metode *packing* cukup mudah. Tulis saja data atau variabel yang ingin di-*pack* dalam notasi tuple, kemudian gunakan sebagai nilai pada operasi *assignment*.

Pada contoh di atas, variabel `row_data` menampung nilai tuple hasil *packing* variabel `first_name`, `rank`, dan `win`.

O iya, penulisan tuple boleh juga dituliskan tanpa menggunakan karakterk `(` & `)`.

```
# dengan ()
row_data = (first_name, rank, win)

# tanpa ()
row_data = first_name, rank, win
```

Namun, pastikan untuk hati-hati dalam penerapan penulisan tuple tanpa `()`, karena bisa jadi salah paham. Jangan gunakan metode ini pada saat menggunakan

tuple sebagai nilai argument pemanggilan fungsi, karena interpreter akan menganggapnya sebagai banyak argument.

```
# fungsi print() dengan satu argument berisi tuple (first_name, rank, win)
print((first_name, rank, win))

# fungsi print() dengan isi 3 arguments: first_name, rank, win
print(first_name, rank, win)
```

> *Penjelasan detail mengenai packing ada di chapter Pack Unpack ➜ Tuple, List, Set, Dict*

◉ Tuple *unpacking*

Unpacking adalah istilah untuk menyebar isi suatu data kolektif ke beberapa variabel. *Unpacking* merupakan kebalikan dari *packing*.

Contoh penerapan tuple *unpacking*:

```
row_data = ('aerith gainsborough', 11, False)
first_name, rank, win = row_data

print(first_name, rank, win)
# output ➜ aerith gainsborough 11 False
```

> *Penjelasan detail mengenai packing ada di chapter Pack Unpack ➜ Tuple, List, Set, Dict*

A.14.10. Tuple kosong `()`

Tuple bisa saja tidak berisi apapun, contohnya data `()`, yang cukup umum digunakan untuk merepresentasikan data kolektif yang isinya bisa saja kosong.

```
empty_tuple = ()
print(empty_tuple)
# output → ()
```

Berikut adalah contoh penerapannya, dimisalkan ada data kolektif yang didapat dari database berbentuk array object. Data tersebut perlu disimpan oleh variabel list yang element-nya adalah tuple dengan spesifikasi:

- Tuple element index 0 berisi `name`.
- Tuple element index 1 berisi `rank`.
- Tuple element index 2 berisi `win`.
- Tuple element index 3 berisi `affliation`, dimana affliation bisa saja kosong.

Sample data bisa dilihat berikut ini:

```
data = [
    ("ultra instinc shaggy", 1, True, ('detective', 'saiyan')),
    ("nightwing", 3, True, ('teen titans', 'bat family')),
    ("kucing meong", 7, False, ()),
]
```

Bisa dilihat data `kucing meong` tidak memiliki `affliation`, karena terisi dengan nilai tuple `()`.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../tuple
```

◉ Chapter relevan lainnya

- List

◉ TBA

- Slicing tuple
- Zip

◉ Referensi

- https://docs.python.org/3/tutorial/datastructures.html
- https://docs.python.org/3/library/stdtypes.html#typesseq

A.15. Python Set

Set adalah tipe data yang digunakan untuk menampung nilai kolektif unik, jadi tidak ada duplikasi elemen. Elemen yang ada pada set disimpan secara tidak urut.

Pada chapter ini, selain mempelajari tentang `set` kita akan bahas juga satu variasinya yaitu `frozenset`.

A.15.1. Set vs. Tuple vs. List

Tipe data set sekilas memiliki kemiripan jika dibandingkan dengan tuple dan list, namun sebenarnya lebih banyak perbedaannya. Silakan lihat tabel berikut untuk lebih jelasnya.

	Set	**Tuple**	**List**
Literal	`set()`, atau elemen ditulis diapit `{` dan `}`	`()`, atau `tuple()`, atau elemen ditulis tanpa `()`	`[]`, atau `list()`

Contoh	`x = set()`
	`x = {1, True, "h", 2}`
		`x = ()`
		`x = tuple()`
		`x = (1, True, "h", 2, 1)`
		`x = 1, True, "h", 2, 1`
			`x = []`
			`x = list()`
			`x = [1, True, "h", 2, 1]`

Urutan elemen	tidak urut	urut sesuai index	
Pengaksesan elemen	hanya via perulangan	via index dan perulangan	
Mutability	elemen bisa diubah	elemen tidak bisa diubah	elemen bisa diubah
Duplikasi elemen	elemen selalu unik	elemen bisa duplikat	
Tipe data elemen	bisa sejenis maupun berbeda satu sama lain		

A.15.2. Pengenalan Set

Implementasi tipe data set cukup mudah, langsung tulis saja nilai elemen dengan separator `,` dan diapit menggunakan tanda kurung kurawal `{ }`. Contoh:

```
data_1 = {1, 'abc', False, ('banana', 'spaghetti')}

print("data:", data_1)
# output → data: {1, 'abc', False, ('banana', 'spaghetti')}

print("len:", len(data_1))
# output → len: 4
```

- Set bisa menampung element yang tipe datanya bisa sejenis bisa tidak, sama seperti tuple dan list.
- Fungsi `len()` digunakan untuk menghitung lebar set.

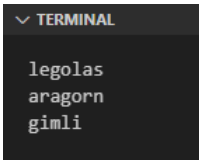
> **INFO**
>
> Untuk deklarasi set kosong (tanpa isi), gunakan fungsi `set()`, bukan `{}` karena literal tersebut akan menciptakan data bertipe lainnya yaitu dictionary.
>
> ```
> data_2 = set()
>
> print("data:", data_2)
> # output → data: set()
>
> print("len:", len(data_2))
> # output → len: 0
> ```
>
> Hanya gunakan kurung kurawal buka dan tutup untuk deklarasi set yang ada elemennya (tidak kosong).

A.15.3. Mengakses elemen set

Nilai set *by default* hanya bisa diakses menggunakan perulangan:

```
fellowship = {'aragorn', 'gimli', 'legolas'}

for p in fellowship:
    print(p)
```



Dari limitasi ini, set difungsikan untuk menyelesaikan masalah yang cukup spesifik seperti eliminasi elemen duplikat.

◉ Eliminasi elemen duplikat

Tipe data set memang didesain untuk menyimpan data unik, duplikasi elemen tidak mungkin terjadi, bahkan meskipun dipaksa. Contoh:

```
data = {1, 2, 3, 2, 1, 4, 5, 2, 3, 5}
print(data)
# output → {1, 2, 3, 4, 5}
```

Variabel `data` yang diisi dengan data set dengan banyak elemen duplikasi, sewaktu di-print elemennya adalah unik.

Ok, selanjutnya, pada contoh kedua berikut kita akan coba gunakan set untuk mengeliminasi elemen duplikat pada suatu list.

```
data = [1, 2, 3, 2, 1, 4, 5, 2, 3, 5]
print(data)
# output → [1, 2, 3, 2, 1, 4, 5, 2, 3, 5]

data_unique_set = set(data)
print(data_unique_set)
# output → {1, 2, 3, 4, 5}

data_unique = list(data_unique_set)
print(data_unique)
# output → [1, 2, 3, 4, 5]
```

Penjelasan untuk kode di atas:

- Variabel `data` berisi list dengan banyak elemen duplikasi
- Data list kemudian dikonversi ke bentuk set dengan cara membungkus variabelnya menggunakan fungsi `set()`. Operasi ini menghasilkan nilai set berisi elemen unik.
- Selanjutnya data set dikonversi lagi ke bentuk list menggunakan fungsi `list()`.

◉ Mengecek apakah element ada

Selain untuk kasus di atas, set juga bisa digunakan untuk pengecekan membership dengan kombinasi keyword `if` dan `in`.

Pada contoh berikut, variabel `fellowship` dicek apakah berisi string `gimli` atau tidak.

```
fellowship = {'aragorn', 'gimli', 'legolas'}
to_find = 'gimli'

if to_find in fellowship:
```

A.15.4. Operasi pada set

◉ Menambah element

Method `add()` milik tipe data set digunakan untuk menambahkan element baru. O iya, perlu diingat bahwa tipe data ini didesain untuk mengabaikan urutan elemen, jadi urutan tersimpannya elemen bisa saja acak.

```
fellowship = set()

fellowship.add('aragorn')
print("len:", len(fellowship), "data:", fellowship)
# output → len: 1 data: {'aragorn'}

fellowship.add('gimli')
print("len:", len(fellowship), "data:", fellowship)
# output → len: 2 data: {'gimli', 'aragorn'}

fellowship.add('legolas')
print("len:", len(fellowship), "data:", fellowship)
# output → len: 3 data: {'gimli', 'legolas', 'aragorn'}
```

◉ Menghapus element secara acak

Gunakan method `pop()` untuk menghapus satu elemen secara acak atau random.

```
fellowship = {'narya', 'nenya', 'nilya'}

fellowship.pop()
print("len:", len(fellowship), "data:", fellowship)
# output → len: 2 data: {'narya', 'nilya'}

fellowship.pop()
print("len:", len(fellowship), "data:", fellowship)
# output → len: 1 data: {'nilya'}

fellowship.pop()
print("len:", len(fellowship), "data:", fellowship)
# output → len: 0 data: set()
```

◉ Menghapus spesifik elemen

Ada dua method tersedia untuk kebutuhan menghapus elemen tertentu dari suatu set, yaitu `discard()` dan `remove()`. Penggunaan keduanya adalah sama, harus disertai dengan 1 argument pemanggilan method, yaitu elemen yang ingin dihapus.

Pada contoh berikut, elemen `boromir` dihapus dari set menggunakan method `discard()`, dan elemen `gandalf` dihapus menggunakan method `remove()`.

```
fellowship = {'aragorn', 'gimli', 'legolas', 'gandalf', 'boromir', 'frodo', 'sam', 'merry', 'pippin'}
print("fellowship:", fellowship)
# output → fellowship: {'aragorn', 'gimli', 'legolas', 'gandalf', 'boromir', 'frodo', 'sam', 'merry',
'pippin'}

fellowship.discard('boromir')
print("fellowship:", fellowship)
# output → fellowship: {'legolas', 'pippin', 'sam', 'aragorn', 'gimli', 'frodo', 'gandalf', 'merry'}

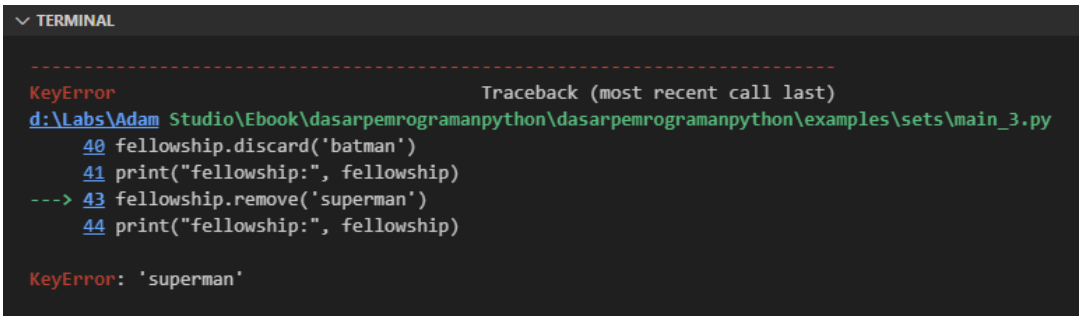
fellowship.remove('gandalf')
print("fellowship:", fellowship)
# output → fellowship: {'legolas', 'pippin', 'sam', 'aragorn', 'gimli', 'frodo', 'merry'}
```

Perbedaan dua method di atas: jika elemen yang ingin dihapus tidak ada, method `discard()` tidak memunculkan error, sedangkan method `remove()` memunculkan error. Contoh:

```
fellowship = {'aragorn', 'gimli', 'legolas', 'gandalf', 'boromir', 'frodo', 'sam', 'merry', 'pippin'}
print("fellowship:", fellowship)

fellowship.discard('batman')
print("fellowship:", fellowship)

fellowship.remove('superman')
print("fellowship:", fellowship)
```



◉ Mengosongkan isi set

Method `clear()` digunakan untuk mengosongkan isi set.

```
fellowship = {'aragorn', 'gimli', 'legolas'}
fellowship.clear()

print("len:", len(fellowship), "data:", fellowship)
# output → len: 0 data: set()
```

◉ *Copy* set

Method `copy()` digunakan untuk meng-*copy* set, menghasilkan data set baru.

```
data1 = {'aragorn', 'gimli', 'legolas'}
print("len:", len(data1), "data1:", data1)
# output → len: 3 data1: {'gimli', 'legolas', 'aragorn'}

data2 = data1.copy()
print("len:", len(data2), "data2:", data2)
# output → len: 3 data2: {'gimli', 'legolas', 'aragorn'}
```

Pada contoh di atas, statement `data1.copy()` menghasilkan data baru dengan isi sama seperti isi `data1` ditampung oleh variabel bernama `data2`.

> *Operasi copy disini jenisnya adalah shallow copy.*
> *Lebih detailnya mengenai shallow copy vs deep copy dibahas pada chapter terpisah.*

◉ Pengecekan *difference* antar set

Method `difference()` digunakan untuk mencari perbedaan elemen antara data (dimana method dipanggil) vs. data pada argument pemanggilan method tersebut.

Sebagai contoh, pada variabel `fellowship` berikut akan dicari elemen yang tidak ada di variabel `hobbits`.

```
fellowship = {'aragorn', 'gimli', 'legolas', 'gandalf', 'boromir', 'frodo', 'sam', 'merry', 'pippin'}
hobbits = {'frodo', 'sam', 'merry', 'pippin', 'bilbo'}

diff = fellowship.difference(hobbits)
print("diff:", diff)
# output → diff: {'boromir', 'legolas', 'aragorn', 'gimli', 'gandalf'}
```

Selain method di atas, adalagi method `difference_update()` yang kegunaannya adalah mengubah nilai data (dimana method dipanggil) dengan nilai baru yang didapat dari perbedaan elemen antara data tersebut vs. data pada argument pemanggilan method.

```
fellowship.difference_update(hobbits)
print("fellowship:", fellowship)
# output → fellowship: {'boromir', 'legolas', 'aragorn', 'gimli', 'gandalf'}
```

◉ Pengecekan *intersection* antar set

Method `intersection()` digunakan untuk mencari elemen yang ada di data (dimana method dipanggil) vs. data pada argument pemanggilan method tersebut.

Pada variabel `fellowship` berikut akan dicari elemen yang juga ada pada variabel `hobbits`.

```
fellowship = {'aragorn', 'gimli', 'legolas', 'gandalf', 'boromir', 'frodo', 'sam', 'merry', 'pippin'}
hobbits = {'frodo', 'sam', 'merry', 'pippin', 'bilbo'}
```

Tersedia juga method `intersection_update()` yang berguna untuk mengubah nilai data (dimana method dipanggil) dengan nilai baru yang didapat dari kesamaan elemen antara data tersebut vs. data pada argument pemanggilan method.

```
fellowship.intersection_update(hobbits)
print("fellowship:", fellowship)
# output → fellowship: {'frodo', 'pippin', 'sam', 'merry'}
```

◉ Pengecekan keanggotaan *subset*

Di awal chapter ini kita telah sedikit menyinggung pengecekan membership menggunakan kombinasi keyword `if` dan `in`. Selain metode tersebut, ada alternatif cara lain yang bisa digunakan untuk mengecek apakah suatu data (yang pada konteks ini adalah set) merupakan bagian dari element set lain, caranya menggunakan method `issubset()`.

Method `issubset()` menerima argument berupa data set. Contohnya bisa dilihat pada kode berikut:

```
fellowship = {'aragorn', 'gimli', 'legolas', 'gandalf', 'boromir', 'frodo', 'sam', 'merry', 'pippin'}

hobbits_1 = {'frodo', 'sam', 'merry', 'pippin', 'bilbo'}
res_1 = hobbits_1.issubset(fellowship)
print("res_1:", res_1)
# output → res_1: False

hobbits_2 = {'frodo', 'sam', 'merry', 'pippin'}
res_2 = hobbits_2.issubset(fellowship)
print("res_2:", res_2)
# output → res_2: True
```

- Nilai `res_1` adalah `False` karena set `hobbits_1` memiliki setidaknya satu elemen yang bukan anggota dari `fellowship`, yaitu `bilbo`.
- Nilai `res_2` adalah `True` karena set `hobbits_2` semua elemennya adalah anggota dari `fellowship`.

◉ Pengecekan keanggotaan *superset*

Selain `issubset()`, ada juga `issuperset()` yang fungsinya kurang lebih sama namun kondisinya pengecekannya dibalik.

Agar lebih jelas, silakan lihat kode berikut:

```
fellowship = {'aragorn', 'gimli', 'legolas', 'gandalf', 'boromir', 'frodo', 'sam', 'merry', 'pippin'}

hobbits_1 = {'frodo', 'sam', 'merry', 'pippin', 'bilbo'}
res_1 = fellowship.issuperset(hobbits_1)
print("res_1:", res_1)
# output → res_1: False

hobbits_2 = {'frodo', 'sam', 'merry', 'pippin'}
res_2 = fellowship.issuperset(hobbits_2)
```

- Nilai `res_1` adalah `False` karena set `hobbits_1` memiliki setidaknya satu elemen yang bukan anggota dari `fellowship`, yaitu `bilbo`.
- Nilai `res_2` adalah `True` karena set `hobbits_2` semua elemennya adalah anggota dari `fellowship`.

◉ Pengecekan keanggotaan *disjoint*

Method ini mengembalikan nilai `True` jika set pada pemanggilan fungsi berisi elemen yang semuanya bukan anggota data dimana method dipanggil.

```
fellowship = {'aragorn', 'gimli', 'legolas', 'gandalf', 'boromir', 'frodo', 'sam', 'merry', 'pippin'}

res_1 = fellowship.isdisjoint({'aragorn', 'gimli'})
print("res_1:", res_1)

res_2 = fellowship.isdisjoint({'pippin', 'bilbo'})
print("res_2:", res_2)

res_3 = fellowship.isdisjoint({'bilbo'})
print("res_3:", res_3)
```

- Nilai `res_1` adalah `False` karena beberapa anggota set `fellowship` adalah `aragorn` dan `gimli'`.
- Nilai `res_2` adalah `False` karena beberapa anggota set `fellowship` adalah `pippin`. Sedangkan `bilbo` ia bukanlah anggota `fellowship`, tapi karena setidaknya ada 1 elemen yang match, maka method `isdisjoint` mengembalikan nilai `False`.
- Nilai `res_3` adalah `True` karena `bilbo` bukanlah anggota `fellowship`.

◉ *Extend/concat/union* element

Operasi *extend* (atau *concat* atau *union*) adalah operasi penggabungan dua data set. Ada beberapa metode yang tersedia, diantaranya:

- via method `union()`:

```
hobbits = {'frodo', 'sam', 'merry', 'pippin'}
dunedain = {'aragorn'}
elf = {'legolas'}
dwarf = {'gimly'}
human = {'boromir'}
maiar = {'gandalf'}

fellowship_1 = hobbits.union(dunedain).union(dunedain).union(elf).union(dwarf).union(human).union(maiar)
print("fellowship_1:", fellowship_1)
# output → fellowship_1: {'boromir', 'gimly', 'legolas', 'pippin', 'sam', 'aragorn', 'frodo', 'gandalf',
'merry'}
```

- via method `update()`:

```
hobbits = {'frodo', 'sam', 'merry', 'pippin'}
```

Bisa dilihat perbedaannya ada di-bagaimana nilai balik method disimpan.

- Pada method `union()`, pemanggilan method tersebut mengembalikan data setelah penggabungan, dan bisa di-chain langsung dengan pemanggilan method `union()` lainnya.
- Pada method `update()`, data yang digunakan untuk memanggil method tersebut diubah secara langsung nilainya.

◉ Operator bitwise pada set

- Operasi `or` pada set menggunakan operator `|`

```
a = set('abracadabra') # {'c', 'a', 'r', 'd', 'b'}
b = set('alacazam')    # {'c', 'z', 'a', 'm', 'l'}

res = a | b
print(res)
# output → {'c', 'z', 'a', 'r', 'd', 'b', 'm', 'l'}
```

  Nilai `res` berisi elemen set unik kombinasi set `a` dan set `b`.

- Operasi `and` pada set menggunakan operator `&`

```
a = set('abracadabra') # {'c', 'a', 'r', 'd', 'b'}
b = set('alacazam')    # {'c', 'z', 'a', 'm', 'l'}

res = a & b
print(res)
# output → {'c', 'a'}
```

  Nilai `res` berisi elemen set yang merupakan anggota set `a` dan set `b`. Operasi seperti ini biasa disebut dengan operasi *and*.

- Operasi `exclusive or` pada set menggunakan operator `^`

```
a = set('abracadabra') # {'c', 'a', 'r', 'd', 'b'}
b = set('alacazam')    # {'c', 'z', 'a', 'm', 'l'}

res = a ^ b
print(res)
# output → {'z', 'r', 'b', 'd', 'm', 'l'}
```

  Nilai `res` berisi elemen set yang ada di set `a` atau set `b` tetapi tidak ada di-keduanya.

◉ Operator `-` pada set

Digunakan untuk pencarian perbedaan elemen. Contoh penerapan:

```
a = set('abracadabra') # {'c', 'a', 'r', 'd', 'b'}
```

Nilai `res` berisi elemen set unik yang merupakan anggota set `a` tapi bukan anggota set `b`

A.15.5. Fungsi `set()`

◉ Konversi string ke set

String dibungkus menggunakan method `set()` menghasilkan data set berisi karakter string yang unik.

```
data = set('abcda')
print('data', data)
# output → data {'c', 'b', 'a', 'd'}
```

◉ Konversi list ke set

Data list bisa diubah menjadi set dengan mudah dengan cara membungkusnya menggunakan fungsi `set()`. Isi dari set adalah elemen unik list.

```
data = set(['a', 'b', 'c', 'd', 'a'])
print('data', data)
# output → data {'c', 'b', 'a', 'd'}
```

◉ Konversi tuple ke set

Data tuple juga bisa diubah menjadi set via fungsi `set()`. Isi dari set adalah elemen unik tuple.

```
data = set(('a', 'b', 'c', 'd', 'a'))
print('data', data)
# output → data {'c', 'b', 'a', 'd'}
```

◉ Konversi range ke set

Data range (hasil dari pemanggilan fungsi `range()`) bisa dikonversi ke bentuk set via fungsi `set()`.

```
data = set(range(1, 5))
print('data', data)
# output → data {1, 2, 3, 4}
```

A.15.6. Set comprehension

Metode comprehension juga bisa diterapkan pada set. Contohnya bisa dilihat pada kode berikut, statement set comprehension dibuat untuk melakukan pengecekan apakah ada element pada set `set('abracadabra')` yang bukan anggota element `set('abc')`.

```
res = {x for x in set('abracadabra') if x not in set('abc')}
print(res)
# output → {'d', 'r'}
```

A.15.7. `frozenset`

`frozenset` adalah `set` yang *immutable* atau tidak bisa diubah nilai elemennya setelah dideklarasikan.

Cara penggunaannya seperti set, perbedaannya pada deklarasi frozenset, fungsi `frozenset()` digunakan dan bukan `set()`.

```
a = frozenset('abracadabra')
print(a)
# output → frozenset({'c', 'a', 'r', 'd', 'b'})

b = frozenset('alacazam')
print(b)
# output → frozenset({'c', 'z', 'a', 'm', 'l'})
```

Semua operasi set, method milik set bisa digunakan pada frozenset, kecuali beberapa operasi yang sifatnya *mutable* atau mengubah elemen. Contohnya seperti method `add()`, `pop()`, `remove()` dan lainnya tidak bisa digunakan di frozenset.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../set
```

◉ Chapter relevan lainnya

- List
- List Comprehension
- Tuple

◉ Referensi

- https://docs.python.org/3/tutorial/datastructures.html
- https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset

A.16. Python Dictionary

Pada chapter ini kita akan belajar salah satu tipe data *mapping* di Python, yaitu Dictionary.

A.16.1. Pengenalan Dictionary

Dictionary atau `dict` adalah tipe data kolektif berbentuk **key-value**. Contoh penulisannya:

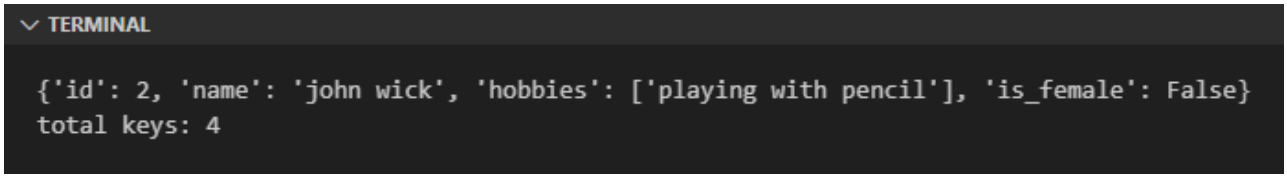
```
profile = {
    "id": 2,
    "name": "john wick",
    "hobbies": ["playing with pencil"],
    "is_female": False,
}
```

Literal dictionary ditulis dengan menggunakan `{ }`, mirip seperti tipe data set, hanya saja bedanya pada tipe dictionary isinya berbentuk **key-value**.

> *Pembahasan detail mengenai tipe data set ada di chapter Set*

Ok, sekarang dari kode di atas, coba tambahkan kode berikut untuk melihat bagaimana data dictionary dimunculkan di layar console.

```
print("data:", profile)
print("total keys:", len(profile))
```



Sedangkan untuk memunculkan nilai item tertentu berdasarkan key-nya, bisa dilakukan menggunakan notasi `dict["key"]`. Contoh:

```
print("name:", profile["name"])
# output → name: john wick

print("hobbies:", profile["hobbies"])
# output → ['playing with pencil']
```

DANGER

Pengaksesan item menggunakan key yang tidak dikenali akan menghasilkan error.

Sebagai contoh, variabel `profile` di atas jika diakses item dengan key `age` misalnya (`profile["age"]`) hasilnya adalah error.

```
profile = {
    "id": 2,
    "name": "john wick",
    "hobbies": ["playing with pencil"],
    "is_female": False,
}

print("age:", profile["age"])
# KeyError: 'age'
```

◉ Urutan item dictionary

Mulai dari Python version 3.7, item dictionary tersimpan secara urut. Artinya urutan item dictionary akan selalu sesuai dengan bagaimana inisialisasi awalnya.

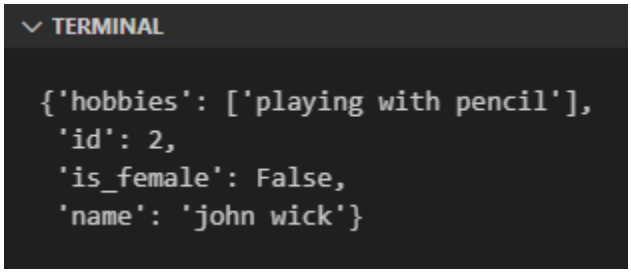
◉ Pretty print dictionary

Ada tips agar data dictionary yang di-print di console muncul dengan tampilan yang lebih mudah dibaca, dua diantaranya:

- Menggunakan `pprint.pprint()`:

  Import terlebih dahulu module `pprint`, lalu gunakan fungsi `pprint()` untuk memunculkan data ke console.

  ```
  import pprint
  pprint.pprint(profile)
  ```

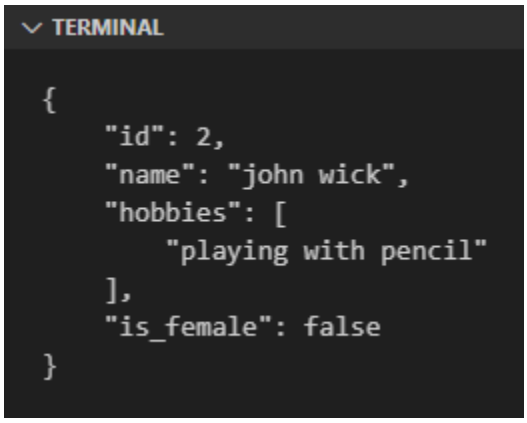
  

- Menggunakan `json.dumps()`:

  Import terlebih dahulu module `json`, lalu gunakan fungsi `dumps()` untuk memformat dictionary menjadi bentuk string yang mudah dibaca, kemudian print menggunakan `print()`.

Tentukan lebar *space indentation* sesuai selera (pada contoh di bawah ini di set nilainya `4` spasi).

```
import json
print(json.dumps(profile, indent=4))
```



> *Lebih detailnya mengenai JSON dibahas di chapter JSON*

A.16.2. Inisialisasi dictionary

Pembuatan data dictionary bisa dilakukan menggunakan beberapa cara:

- Menggunakan `{ }`:

```
profile = {
    "id": 2,
    "name": "john wick",
    "hobbies": ["playing with pencil"],
    "is_female": False,
}
```

- Menggunakan fungsi `dict()` dengan isi argument **key-value**:

```
profile = dict(
    set="id",
    name="john wick",
    hobbies=["playing with pencil"],
    is_female=False,
)
```

- Menggunakan fungsi `dict()` dengan isi list tuple:

```
profile = dict([
    ('set', "id"),
    ('name', "john wick"),
    ('hobbies', ["playing with pencil"]),
    ('is_female', False)
])
```

Sedangkan untuk membuat dictionary tanpa item atau kosong, bisa cukup menggunakan `dict()` atau `{}`:

```
profile = dict()
print(profile)
# output → {}

profile = {}
print(profile)
# output → {}
```

A.16.3. Perulangan item dictionary

Gunakan keyword `for` dan `in` untuk mengiterasi data tiap key milik

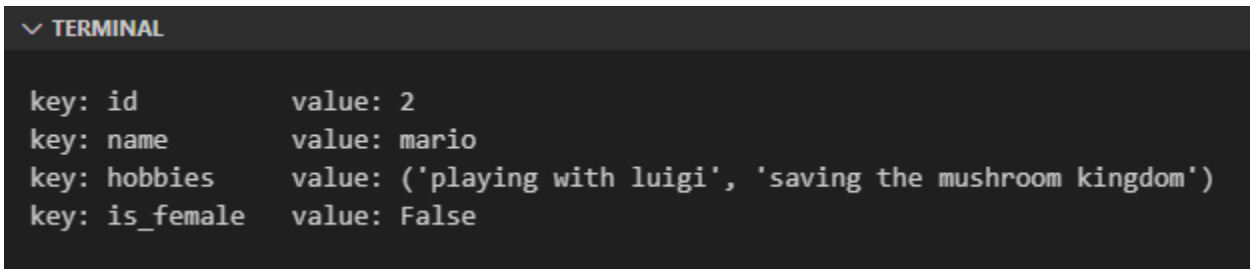
dictionary. Dari key tersebut kemudian akses value-nya.

```
profile = {
    "id": 2,
    "name": "mario",
    "hobbies": ("playing with luigi", "saving the mushroom kingdom"),
    "is_female": False,
}

for key in profile:
    print("key:", key, "\t value:", profile[key])
```

> *Karakter `\t` menghasilkan tab. Penggunaan karakter ini bisa membuat rapi tampilan output.*

Program di atas ketika di run outputnya:



A.16.4. Nested dictionary

Dictionary bercabang atau **nested dictionary** bisa dimanfaatkan untuk menyimpan data dengan struktur yang kompleks, misalnya dictionary yang salah satu value item-nya adalah list.

Penerapannya tak berbeda seperti inisialisasi dictionary umumnya, langsung tulis saja dictionary sebagai child dictionary. Contoh:

```
profile = {
    "id": 2,
    "name": "mario",
    "hobbies": ("playing with luigi", "saving the mushroom kingdom"),
    "is_female": False,
    "affliations": [
        {
            "name": "luigi",
            "affliation": "brother"
        },
        {
            "name": "mushroom kingdom",
            "affliation": "protector"
        },
    ]
}

print("name:", profile["name"])
print("hobbies:", profile["hobbies"])
print("affliations:")

for item in profile["affliations"]:
    print("  ➜ %s (%s)" % (item["name"], item["affliation"]))

# output ↓
#
# name: mario
# hobbies: ('playing with luigi', 'saving the mushroom kingdom')
# affliations:
#   ➜ luigi (brother)
#   ➜ mushroom kingdom (protector)
```

Pada kode di atas, key `affliations` berisi list object dictionary.

Contoh cara mengakses value nested item dictionary:

```
value = profile["affliations"][0]["name"],
profile["affliations"][0]["affliation"]
print("  ➜ %s (%s)" % (value))
# output ➜ luigi (brother)

value = profile["affliations"][1]["name"],
profile["affliations"][1]["affliation"]
print("  ➜ %s (%s)" % (value))
# output ➜ mushroom kingdom (protector)
```

A.16.5. Dictionary mutability

Item dictionary adalah mutable, perubahan value item bisa dilakukan langsung menggunakan operator assignment `=`.

```
profile = {
    "id": 2,
    "name": "mario",
    "hobbies": ("playing with luigi", "saving the mushroom kingdom"),
    "is_female": False,
    "affliations": [
        {
            "name": "luigi",
            "affliation": "brother"
        },
        {
            "name": "mushroom kingdom",
            "affliation": "protector"
        },
    ]
}

print(profile["affliations"][0]["name"])
# output ➜ luigi
```

A.16.6. Operasi data dictionary

◉ Pengaksesan item

Pengaksesan item dilakukan lewat notasi `dict["key"]`, atau bisa dengan menggunakan method `get()`.

```
profile = {
    "id": 2,
    "name": "mario",
    "hobbies": ("playing with luigi", "saving the mushroom kingdom"),
    "is_female": False,
}

print("id:", profile["id"])
# output → id: 2

print("name:", profile.get("name"))
# output → name: mario
```

◉ Mengubah isi dictionary

Cara mengubah value item dictionary adalah dengan mengaksesnya terlebih dahulu, kemudian diikuti operasi assignment.

```
profile = {
    "id": 2,
    "name": "mario",
    "hobbies": ("playing with luigi", "saving the mushroom kingdom"),
    "is_female": False,
}
print("name:", profile["name"])
```

◉ Menambah item dictionary

Caranya adalah mirip seperti operasi pengubahan value item, perbedaannya ada pada key-nya. Key yang ditulis adalah key item baru yang ingin ditambahkan.

```
profile = {
    "name": "mario",
}
print("len:", len(profile), "data:", profile)
# output → len: 1 data: {'name': 'mario'}

profile["favourite_color"] = "red"
print("len:", len(profile), "data:", profile)
# output → len: 2 data: {'name': 'mario', 'favourite_color': 'red'}
```

Selain cara tersebut, bisa juga dengan menggunakan method `update()`. Tulis key dan value baru yang ingin ditambahkan sebagai argument method `update()` dalam bentuk dictionary.

```
profile.update({"race": "italian"})
print("len:", len(profile), "data:", profile)
# output → len: 3 data: {'name': 'mario', 'favourite_color': 'red',
'race': 'italian'}
```

◉ Menghapus item dictionary

Method `pop()` digunakan untuk menghapus item dictionary berdasarkan key.

```
profile.pop("hobbies")
print(profile)
```

Keyword `del` juga bisa difungsikan untuk operasi yang sama. Contoh:

```
del profile["id"]
print(profile)
```

◉ Pengaksesan dictionary keys

Method `keys()` digunakan untuk mengakses semua keys dictionary, hasilnya adalah tipe data view objects `dict_keys`. Dari nilai tersebut bungkus menggunakan `list()` untuk mendapatkan nilainya dalam bentuk list.

```
profile = {
    "id": 2,
    "name": "mario",
    "hobbies": ("playing with luigi", "saving the mushroom kingdom"),
    "is_female": False,
}

print(list(profile.keys()))
# output → ['id', 'name', 'hobbies', 'is_female']
```

◉ Pengaksesan dictionary values

Method `values()` digunakan untuk mengakses semua value dictionary, hasilnya adalah tipe data view objects `dict_values`. Gunakan fungsi `list()` untuk mengkonversinya ke bentuk list.

```
profile = {
    "id": 2,
    "name": "mario",
    "hobbies": ("playing with luigi", "saving the mushroom kingdom"),
    "is_female": False,
```

◉ Method `items()` dictionary

Digunakan untuk mengakses semua item dictionary. Nilai baliknya bertipe view objects `dict_items` yang strukturnya cukup mirip seperti list berisi tuple.

Untuk mengkonversinya ke bentuk list, gunakan fungsi `list()`.

```
profile = {
    "id": 2,
    "name": "mario",
    "hobbies": ("playing with luigi", "saving the mushroom kingdom"),
    "is_female": False,
}

print(list(profile.items()))
# output → [('id', 2), ('name', 'mario'), ('hobbies', ('playing with
luigi', 'saving the mushroom kingdom')), ('is_female', False)]
```

◉ Copy dictionary

Method `copy()` digunakan untuk meng-*copy* dictionary, hasilnya data dictionary baru.

```
p1 = {
    "id": 2,
    "name": "mario",
    "is_female": False,
}
print(p1)
# output → {'id': 2, 'name': 'mario', 'is_female': False}

p2 = p1.copy()
print(p2)
```

Pada contoh di atas, statement `p1.copy()` menghasilkan data baru dengan isi sama seperti isi `p1`, data tersebut kemudian ditampung oleh variabel `p2`.

> *Operasi copy disini jenisnya adalah shallow copy.*
> *Lebih detailnya mengenai shallow copy vs deep copy dibahas pada chapter terpisah.*

◉ Mengosongkan isi dictionary

Method `clear()` berguna untuk menghapus isi dictionary.

```
profile = {
    "id": 2,
    "name": "mario",
    "is_female": False,
}
print("len:", len(profile), "data:", profile)
# output → len: 3 data: {'id': 2, 'name': 'mario', 'is_female': False}

profile.clear()
print("len:", len(profile), "data:", profile)
# output → len: 0 data: {}
```

Catatan chapter 🗒️

◉ Source code praktik

`github.com/novalagung/dasarpemrogramanpython-example/../dictionary`

◉ Chapter relevan lainnya

- OOP ➜ Class & Object

◉ Referensi

- https://docs.python.org/3/library/stdtypes.html#mapping-types-dict

A.17. Python String

String (atau `str`) merupakan kumpulan data `char` atau karakter yang tersimpan secara urut (*text sequence*). String di Python mengadopsi standar Unicode dengan *default encoding* adalah `UTF-8`.

A.17.1. Pengenalan String

Python mendesain tipe data string dalam bentuk yang sangat sederhana dan mudah digunakan. Untuk membuat string cukup tulis saja text yang diinginkan dengan diapit tanda petik satu atau petik dua. Contoh:

```
text = "hello python"
print(text)
# output → hello python

text = 'hello python'
print(text)
# output → hello python
```

◉ Multiline string

Untuk string *multiline* atau lebih dari satu baris, cara penulisannya bisa dengan:

- Menggunakan karakter spesial `\n`:

```
text = "a multiline string\nin python"

print(text)
```

- Atau menggunakan tanda `"""` atau `'''` untuk mengapit text. Contoh:

```
text = """a multiline string
in python"""

print(text)
# output ↓
#
# a multiline string
# in python
```

> *Teknik penulisan string menggunakan tanda `"""` atau `'''` selain digunakan untuk membuat multiline string, juga bisa dimanfaatkan untuk hal lain yaitu pembuatan docstring.*
>
> *Lebih detailnya mengenai docstring dibahas di chapter DocString.*

◉ Escape character

Python mengenal *escape character* umum yang ada di banyak bahasa pemrograman, contohnya seperti `\"` digunakan untuk menuliskan karakter `"` (pada string yang dibuat menggunakan literal `" "`). Penambahan karakter `\` adalah penting agar karakter `"` terdeteksi sebagai penanda string.

Sebagai contoh, dua statement berikut adalah ekuivalen:

```
text = 'this is a "string" in python'
print(text)
# output → this is a "string" in python

text = "this is a \"string\" in python"
```

A.17.2. String *special characters*

Di atas telah dicontohkan bagaimana cara menulis karakter *newline* atau baris baru menggunakan `\n`, dan karakter petik dua menggunakan `\"`. Dua karakter tersebut adalah contoh dari *special characters*.

Python mengenal banyak special characters yang masing-masing memiliki kegunaan yang cukup spesifik. Agar lebih jelas silakan lihat tabel berikut:

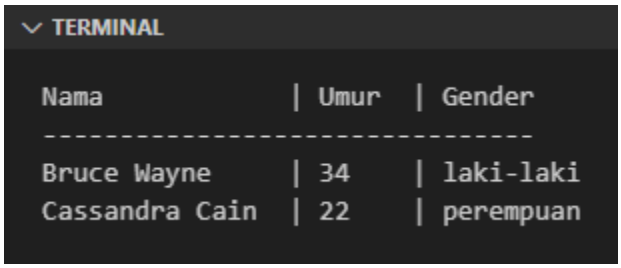
Special character	Kegunaan
\\	karakter backslash (\)
\'	tanda petik satu (')
\"	tanda kutip (petik dua) (")
\a	bunyi *beeb* (ASCII BEL)
\b	backspace (ASCII BS)
\f	page separator / formfeed (ASCII FF)
\n	karakter baris baru linefeed (ASCII LF)
\r	karakter baris baru carriage return (ASCII CR)
\t	horizontal tab (ASCII TAB)

Special character	Kegunaan
\v	vertical tab (ASCII VT)
\{oktal}	nilai oktal, contoh: `\122`, `\004`, `\024`
\x{hex}	nilai heksadesimal, contoh: `\xA4`, `\x5B`

Tambahan contoh penggunaan salah satu special character `\t` (horizontal tab):

```
print("Nama\t\t| Umur\t| Gender")
print("-------------------------------")
print("Bruce Wayne\t| 34\t| laki-laki")
print("Cassandra Cain\t| 22\t| perempuan")
```

Program di atas menghasilkan output berikut:



> *Syntax `0xC548` adalah salah satu penulisan numerik berbasis hexadecimal. Pembahasan detailnya ada di chapter Number.*

A.17.3. String formatting

String formatting adalah teknik untuk mem-format string agar menghasilkan text sesuai dengan format yang diinginkan.

Cara termudah melakukan string formatting adalah dengan menggunakan **f-strings** (atau formatted string literals). Tulis string seperti biasa tapi diawali dengan huruf `f` atau `F` sebelum penulisan `" "`.

Pada contoh berikut, sebuah string dibuat dimana dua bagian string didalamnya datanya bersumber dari variabel string lain.

```
name = "Aiden Pearce"
occupation = "IT support"

text = f"hello, my name is {name}, I'm an {occupation}"
print(text)
# output → hello, my name is Aiden Pearce, I'm an IT support
```

Penjelasan:

- String dibuat dengan metode f-strings, dimana struktur text adalah `hello, my name is {name}, I'm an {occupation}`.
- Text `{name}` di dalam string di-replace oleh nilai variable `name`, yang pada konteks ini nilainya `Aiden Pearce`.
- Text `{occupation}` di dalam string di-replace oleh nilai variable `occupation`, yang pada konteks ini nilainya `IT support`.
- f-strings di atas menghasilkan text `hello, my name is Aiden Pearce, I'm an IT support`.

> *Pada penerapan metode **f-strings**, isi dari `{}` tidak harus data string, tetapi tipe data lainnya juga bisa digunakan asalkan printable atau bisa di-print.*

Selain menggunakan metode di atas, ada beberapa alternatif cara lain yang bisa digunakan, diantaranya:

```
text = "hello, my name is {name}, I'm an {occupation}".format(name =
name, occupation = occupation)
print(text)
# output → hello, my name is Aiden Pearce, I'm an IT support

text = "hello, my name is {0}, I'm an {1}".format(name, occupation)
print(text)
# output → hello, my name is Aiden Pearce, I'm an IT support

text = "hello, my name is {}, I'm an {}".format(name, occupation)
print(text)
# output → hello, my name is Aiden Pearce, I'm an IT support
```

Semua metode string formatting yang telah dipelajari menghasilkan nilai balik yang sama, yaitu `hello, my name is Aiden Pearce, I'm an IT support`. Mana yang lebih baik? Silakan pilih saja metode yang sesuai selera.

A.17.4. Penggabungan string & split string

◉ Penggabungan string (concatenation)

Ada beberapa metode yang bisa digunakan untuk *string concatenation* atau operasi penggabungan string.

- Menggunakan teknik penulisan string literal sebaris.

  Caranya dengan langsung tulis saja semua string-nya menggunakan separator karakter spasi.

```
text = "hello " "python"
print(text)
# output → hello python
```

- Menggunakan operator `+`.

  Operator `+` jika diterapkan pada string menghasilkan penggabungan string.

```
text_one = "hello"
text_two = "python"
text = text_one + " " + text_two

print(text)
# output → hello python
```

  Untuk data non-string jika ingin digabung harus dibungkus dengan fungsi `str()` terlebih dahulu. Fungsi `str()` digunakan untuk mengkonversi segala jenis data ke bentuk string.

```
text = "hello"
number = 123
yes = True

message = text + " " + str(number) + " " + str(yes)

print(message)
```

- Menggunakan method `join()` milik string.

  Pada penerapannya, karakter pembatas atau *separator* ditulis terlebih dahulu, kemudian di-*chain* dengan method join dengan isi argument adalah list yang ingin digabung.

  ```
  text = " ".join(["hello", "python"])
  print(text)
  # output → hello python
  ```

◉ Split string

Split string adalah operasi pemisahan karakter string menjadi list dengan element berisi string. Operasi ini merupakan kebalikan dari penggabungan string.

Cara penggunaannya adalah dengan mengakses method `split()` langsung dari data string. Pemanggilan method tersebut perlu disertai argument berisi separator atau karakter pemisah string.

Pada contoh berikut, string `hello python 12345` di split menggunakan karakter spasi (` `).

```
text = "hello python 12345"
res = text.split(" ")
print(res)
# output → ['hello', 'python', '12345']
```

> *Parameter method `split()` boleh untuk tidak diisi apapun, dengan efek*

> *adalah string menjadi di-split menggunakan separator* `\n`, `\r`, `\t`, `\f` *atau spasi (* ` ` *).*

A.17.5. Operasi sequence pada string

String masih termasuk kategori tipe data sequence, yang artinya bisa digunakan pada operasi standar sequence, contoh seperti perulangan, pengaksesan elemen, dan slicing.

◉ Mengecek jumlah karakter string

Fungsi `len()` ketika digunakan pada tipe data string mengembalikan informasi jumlah karakter string.

```
text = "hello python"

print("text:", text)
# output → hello python

print("length:", len(text))
# output → 12
```

◉ Mengakses element string

Setiap elemen string bisa diakses menggunakan index. Penulisan notasi pengaksesannya sama seperti pada tipe data sequence lainnya, yaitu menggunakan `string[index]`.

```
text = "hello python"
print(text[0])
```

Selain via index, keyword perulangan `for` bisa dimanfaatkan untuk mengiterasi elemen string. Contoh:

```
for c in text:
    print(c)
```

Contoh lain menggunakan `range()`:

```
for i in range(0, len(text)):
    print(text[i])
```

Output:



DANGER

Pengaksesan elemen menggunakan index di-luar kapasitas data akan menghasilkan error.

Sebagai contoh, string `text = "hello"`, jika diakses index ke-5-nya

misalnya (`text[5]`) hasilnya adalah error.

◉ *Slicing* string

Teknik slicing bisa diterapkan pada data string. Contoh:

```
text = "hello python"

print(text[1:5])
# output → ello

print(text[7:])
# output → ython

print(text[:4])
# output → hell
```

> *Pembahasan detail mengenai slice ada di chapter Slice*

A.17.6. Operasi *character & case*

Tipe data string memiliki beberapa method yang berguna untuk keperluan operasi string yang berhubungan dengan *character & case*

◉ Pengecekan karakter alfabet dan angka

- Method `isalpha()` digunakan untuk mengecek apakah string berisi karakter alfabet atau tidak. Nilai kembaliannya `True` jika semua karakter dalam string adalah alfabet.

```
print("abcdef".isalpha())
# output → True, karena abcdef adalah alfabet

print("abc123".isalpha())
# output → False, karena ada karakter 123 yang bukan merupakan alfabet

print("موز".isalpha())
# output → True, karena موز adalah abjad arabic

print("㐀㐁㐂".isalpha())
# output → True, karena 㐀㐁㐂 adalah karakter jepang
```

- Method `isdigit()` digunakan untuk mengecek apakah string berisi karakter digit atau tidak. Nilai kembaliannya `True` jika semua karakter dalam string adalah angka numerik (termasuk pangkat).

```
print("123456".isdigit())
# output → True, karena 123456 adalah digit

print("123abc".isdigit())
# output → False, karena ada karakter abc yang bukan merupakan digit

print('2⅓'.isdigit())
# output → False, karena bilangan pecahan memiliki karakter `/` yang
tidak termasuk dalam kategori digit

print('4²'.isdigit())
# output → True, karena 4² adalah bilangan pangkat

print('٢٨'.isdigit())
# output → True, karena ٢٨ adalah digit arabic

print('𝟜'.isdigit())
# output → True, karena 𝟜 adalah digit
```

- Method `isdecimal()` digunakan untuk mengecek apakah string berisi

karakter desimal atau tidak. Nilai kembaliannya `True` jika semua karakter dalam string adalah angka numerik desimal.

```
print("123456".isdecimal())
# output → True, karena 123456 adalah angka desimal

print("123abc".isdecimal())
# output → False, karena ada karakter abc yang bukan merupakan angka desimal

print('2⅓'.isdecimal())
# output → False, karena bilangan pecahan memiliki karakter `/` yang tidak termasuk dalam kategori angka desimal

print('4²'.isdecimal())
# output → False, karena bilangan pangkat yang tidak termasuk dalam kategori angka desimal

print('٢٨'.isdecimal())
# output → True, karena ٢٨ adalah angka desimal arabic

print('𝟜'.isdecimal())
# output → True, karena 𝟜 adalah angka desimal
```

- Method `isnumeric()` digunakan untuk mengecek apakah string berisi karakter desimal atau tidak. Nilai kembaliannya `True` jika semua karakter dalam string adalah angka numerik (termasuk pecahan, pangkat, dan angka numerik lainnya).

```
print("123456".isnumeric())
# output → True, karena 123456 adalah angka numerik

print("123abc".isnumeric())
# output → False, karena ada karakter abc yang bukan merupakan numerik
```

- Method `isalnum()` digunakan untuk mengecek apakah string berisi setidaknya karakter alfabet atau digit, atau tidak keduanya. Nilai kembaliannya `True` jika semua karakter dalam string adalah alfabet atau angka numerik.

```
print("123abc".isalnum())
# output → True, karena 123 adalah digit dan abc adalah alfabet

print("12345⅛".isalnum())
# output → True, karena 12345⅛ adalah digit

print("abcdef".isalnum())
# output → True, karena abcdef adalah alfabet

print("abc 12".isalnum())
# output → False, karena ada karakter spasi yang bukan merupakan
karakter digit ataupun alfabet

print("موز".isalnum())
# output → True, karena موز adalah abjad arabic

print("你好吗".isalnum())
# output → True, karena 你好吗 adalah karakter jepang
```

◉ Pengecekan karakter *whitespace*

Method `isspace()` digunakan untuk mengecek apakah string berisi karakter *whitespace*.

```
print(" ".isspace())
# output → True, karena string berisi karakter spasi

print("\n".isspace())
# output → True, karena string berisi karakter newline
```

◉ Pengecekan karakter *case*

- Method `islower()` digunakan untuk mengecek apakah semua karakter string adalah ditulis dalam huruf kecil (*lower case*), jika kondisi tersebut terpenuhi maka nilai kembaliannya adalah `True`.

```
print("hello python".islower())
# output → True

print("Hello Python".islower())
# output → False

print("HELLO PYTHON".islower())
# output → False
```

- Method `istitle()` digunakan untuk mengecek apakah kata dalam string adalah ditulis dengan awalan huruf besar (*title case*), jika kondisi tersebut terpenuhi maka nilai kembaliannya adalah `True`.

```
print("hello python".istitle())
# output → False

print("Hello Python".istitle())
# output → True

print("HELLO PYTHON".istitle())
# output → False
```

- Method `isupper()` digunakan untuk mengecek apakah semua karakter string adalah ditulis dalam huruf besar (*upper case*), jika kondisi tersebut terpenuhi maka nilai kembaliannya adalah `True`.

```
print("hello python".isupper())
# output → False

print("Hello Python".isupper())
# output → False

print("HELLO PYTHON".isupper())
# output → True
```

◉ Mengubah karakter *case*

Beberapa method yang bisa digunakan untuk mengubah *case* suatu string:

- Method `capitalize()` berfungsi untuk mengubah penulisan karakter pertama string menjadi huruf besar (*capitalize*).
- Method `title()` berfungsi untuk mengubah penulisan kata dalam string diawali dengan huruf besar (*title case*).
- Method `upper()` berfungsi untuk mengubah penulisan semua karakter string menjadi huruf besar (*upper case*).
- Method `lower()` berfungsi untuk mengubah penulisan semua karakter string menjadi huruf kecil (*lower case*).
- Method `swapcase()` berfungsi untuk membalik penulisan case karakter string. Untuk karakter yang awalnya huruf kecil menjadi huruf besar, dan sebaliknya.

```
print("hello python".capitalize())
# output → Hello python

print("hello python".title())
# output → Hello Python

print("hello python".upper())
```

A.17.7. Operasi pencarian string & substring

◉ Pengecekan string menggunakan keyword `in`

Keyword `in` bisa digunakan untuk mengecek apakah suatu string merupakan bagian dari string lain. Nilai balik statement adalah boolean. Contoh:

```
text = "hello world"
print("ello" in text)
# output → True
```

Teknik tersebut bisa dikombinasikan dengan seleksi kondisi `if`:

```
text = "hello world"
if "ello" in text:
    print(f"py is in {text}")
# output → py is in hello world
```

◉ Pengecekan substring

Ada beberapa Method yang bisa digunakan untuk keperluan pengecekan substring, apakah suatu string merupakan bagian dari string lain.

- Menggunakan method `startswith()` untuk mengecek apakah suatu string diawali dengan huruf/kata tertentu.

  ```
  print("hello world".startswith("hell"))
  # output → True
  ```

- Menggunakan method `endswith()` untuk mengecek apakah suatu string diakhiri dengan huruf/kata tertentu.

```
print("hello world".endswith("orld"))
# output → True

print("hello world".endswith("worl"))
# output → False
```

- Menggunakan method `count()` untuk mengecek apakah suatu string merupakan bagian dari string lain.

```
print("hello world".count("ello"))
# output → 1
```

Method ini mengembalikan jumlah huruf/kata yang ditemukan. Jika kebutuhannya adalah mencari tau apakah suatu substring ada atau tidak, maka gunakan operasi logika lebih dari 0 (atau `n > 0`).

```
print("hello world".count("ello") > 0)
# output → True
```

◉ Pencarian index substring

Method-method berikut sebenarnya kegunaannya mirip seperti method untuk pengecekan substring, perbedaannya adalah nilai balik pemanggilan method berupa index substring.

- Method `count()` mengembalikan jumlah substring yang ditemukan sesuai kata kunci yang dicari.

```
text = "hello world hello world"
print(text.count("ello"))
# output → 2
```

- Method `index()` mengembalikan index substring pertama yang ditemukan sesuai kata kunci yang dicari. Jika substring tidak ditemukan, method ini menghasilkan error.

```
text = "hello world hello world"
print(text.index("worl"))
# output → 6
```

- Method `rindex()` mengembalikan index substring pertama yang ditemukan sesuai kata kunci yang dicari dengan urutan pencarian adalah dari kanan. Jika substring tidak ditemukan, method ini menghasilkan error.

```
text = "hello world hello world"
print(text.rindex("worl"))
# output → 18
```

- Method `find()` mengembalikan index substring pertama yang ditemukan sesuai kata kunci yang dicari. Jika substring tidak ditemukan, method ini menghasilkan nilai `-1`.

```
text = "hello world hello world"
print(text.find("worl"))
# output → 6
```

- Method `rfind()` mengembalikan index substring pertama yang ditemukan sesuai kata kunci yang dicari dengan urutan pencarian adalah dari kanan. Jika substring tidak ditemukan, method ini menghasilkan nilai

`-1`.

```
text = "hello world hello world"
print(text.rfind("worl"))
# output → 18
```

A.17.8. Operasi string lainnya

◉ Replace substring

Method `replace()` digunakan untuk me-replace suatu substring dengan string lain. Contoh penggunaan:

```
str_old = "hello world"
str_new = str_old.replace("world", "python")
print(str_new)
# output → hello python
```

◉ Trim / strip

Metode trimming/stripping digunakan untuk menghapus *whitespace* yang diantaranya adalah baris baru dan juga spasi.

Sebelum kita mulai, coba perhatikan kode berikut. String `text` dideklarasikan menggunakan `""" """` yang dalam penerapannya tidak akan meng-*escape* whitespace.

```
text = """
hello python
"""
```

Bisa dilihat saat di print kelihatan *newline* atau baris barunya pada awal string dan juga akhir string.

Dengan menggunakan teknik trimming, whitespace bisa dihilangkan. Ada beberapa method yang bisa digunakan, diantaranya:

- Method `lstrip()` untuk trim *whitespace* karakter di awal atau sebelah kiri string.

```
text = """
hello python
"""

print(f"--{text.lstrip()}--")
# output ↓
#
# --hello python
# --
```

- Method `rstrip()` untuk trim *whitespace* karakter di akhir atau sebelah kanan string.

```
text = """
hello python
"""

print(f"--{text.rstrip()}--")
# output ↓
#
# --
# hello python--
```

- Method `strip()` untuk trim *whitespace* karakter di awal dan akhir string.

```
text = """
hello python
"""

print(f"--{text.strip()}--")
# output → --hello python--
```

◉ Join string

Method `join()` berguna untuk menggabungkan list berisi element string. String yang digunakan untuk memanggil method ini menjadi *separator* operasi join.

```
data = ["hello", "world", "abcdef"]
res = "-".join(data)
print(res)
# output → hello-world-abcdef
```

◉ Konversi data ke string

Ada beberapa metode konversi tipe data ke string, diantaranya:

- Menggunakan fungsi `str()`.

  Fungsi ini bisa digunakan untuk mengkonversi data bertipe apapun ke bentuk string. Contoh penerapan:

```
number = 24
string1 = str(number)
print(string1)
# output → 24
```

- Menggunakan teknik string formatting. Contoh:

```
number = 24
string1 = f"{number}"
print(string1)
# output ➜ 24

items = [1, 2, 3, 4]
string2 = f"{items}"
print(string2)
# output ➜ [1, 2, 3, 4]

obj = {
    "name": "AMD Ryzen 5600g",
    "type": "processor",
    "igpu": True,
}
string3 = f"{obj}"
print(string3)
# output ➜ {'name': 'AMD Ryzen 5600g', 'type': 'processor', 'igpu':
True}
```

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../string
```

◉ Chapter relevan lainnya

- String ➜ Unicode

- Slice

◉ TBA

- Bytes
- Konversi tipe data ke string

◉ Referensi

- https://docs.python.org/3/library/string.html
- https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
- https://docs.python.org/3/reference/lexical_analysis.html#f-strings

A.18. Python Unicode String

Python mengadopsi aturan standar Unicode dalam pengelolaan karakter pada string. Benefitnya Python mendukung dan mengenali berbagai macam jenis karakter, termasuk diantaranya adalah huruf Arab, Jepang, emoji, symbol, dan banyak jenis karakter lainnya.

> *Unicode sendiri adalah suatu aturan standar untuk encoding text yang di-maintain oleh Unicode Consortium. Standarisasi ini diciptakan untuk mendukung semua jenis penulisan yang ada di bumi.*

Pada chapter ini kita akan membahas tentang bagaimana implementasi Unicode di Python.

A.18.1. Pengenalan Unicode String

Dalam dunia per-Unicode-an, ada yang disebut dengan **code point** yaitu suatu angka numerik (bisa desimal maupun hexadecimal) yang merepresentasikan karakter tertentu. Jadi bisa diibaratkan *identifier* dari suatu karakter. Semua karakter ada *code point*-nya, termasuk huruf A, B, C, maupun karakter lainnya (angka, tulisan romawi, symbol, dll).

Cara penulisan karakter unicode sendiri bisa dengan langsung menuliskan karakternya, atau bisa juga dengan menuliskan *code point* dalam notasi tertentu.

- Contoh penulisan text dengan langsung menuliskan karakternya:

```
message = "◇◇◇◇◇ 😀"
print(message)
# output → ◇◇◇◇◇ 😀
```

- Menggunakan notasi special character `\uXXXX`, dimana `XXXX` diisi dengan *code point* dalam encoding 16-bit.

```
message = "\uC548\uB155\uD558\uC138\uC694"
print(message)
# output → ◇◇◇◇◇
```

  - *Code point* 16-bit `C548` merepresentasikan karakter ◇
  - *Code point* 16-bit `B155` merepresentasikan karakter ◇
  - *Code point* 16-bit `D558` merepresentasikan karakter ◇
  - *Code point* 16-bit `C548` merepresentasikan karakter ◇
  - *Code point* 16-bit `C694` merepresentasikan karakter ◇

  Untuk memunculkan emoji menggunakan kode encoding 16-bit butuh tambahan effort karena code point emoji tidak cukup jika direpresentasikan oleh *code point* yang lebarnya hanya 16-bit.

- Menggunakan notasi special character `\UXXXXXXXX`, dimana `XXXXXXXX` diisi *code point* dalam encoding 32-bit.

```
message = "\U0000C548\U0000B155\U0000D558\U0000C138\U0000C694 \U0001F600"
print(message)
```

    - *Code point* 32-bit `0000C548` merepresentasikan karakter
    - *Code point* 32-bit `0000B155` merepresentasikan karakter
    - *Code point* 32-bit `0000D558` merepresentasikan karakter
    - *Code point* 32-bit `0000C138` merepresentasikan karakter
    - *Code point* 32-bit `0000C694` merepresentasikan karakter
    - *Code point* 32-bit `0001F600` merepresentasikan emoji 😀

- Atau menggunakan notasi special character `\N{NAME}`, dimana `NAME` diisi dengan nama karakter unicode dalam huruf besar.

```
message = "\N{HANGUL SYLLABLE AN}\N{HANGUL SYLLABLE NYEONG} \N{GRINNING FACE}"
print(message)
# output → 😀
```

    - Nama karakter Unicode `HANGUL SYLLABLE AN` merepresentasikan karakter
    - Nama karakter Unicode `HANGUL SYLLABLE NYEONG` merepresentasikan karakter
    - Nama karakter Unicode `GRINNING FACE` merepresentasikan emoji 😀

> *Salah satu website yang berguna untuk mencari informasi nama dan code point karakter Unicode: https://www.compart.com/en/unicode/*

A.18.2. Fungsi utilitas pada *Unicode*

◉ Fungsi `ord()`

Fungsi `ord()` digunakan untuk mengambil nilai code point dari suatu karakter. Nilai baliknya adalah numerik berbasis desimal.

```
text = "N"
codePoint = ord(text)
print(f'code point of {text} in decimal: {codePoint}')
# output → code point of N in decimal: 78

text = ""
codePoint = ord(text)
print(f'code point of {text} in decimal: {codePoint}')
# output → code point of  in decimal: 50504
```

Untuk menampilkan code point dalam notasi heksadesimal, cukup bungkus menggunakan fungsi `hex()`.

```
text = ""
codePoint = ord(text)

print(f'code point of {text} in decimal: {codePoint}')
# output → code point of  in decimal: 50504
```

Bisa dilihat dari program di atas, unicode code point dari karakter 🏴 dalam bentuk heksadesimal adalah `c548`. Jika dicek pada praktek sebelumnya, kode heksadesimal yang sama kita gunakan juga dalam penulisan karakter unicode menggunakan notasi `\uXXXX` (yaitu `\uc548`).

◉ Fungsi `chr()`

Fungsi `chr()` adalah kebalikan dari fungsi `ord()`, kegunaannya adalah untuk menampilkan string sesuai code point.

Pada contoh dibawah ini fungsi `chr()` digunakan untuk memunculkan karakter dengan code point desimal `50504` dan juga heksadesimal `C548`, yang keduanya adalah merepresentasikan karakter yang sama, yaitu 🏴.

```
codePoint = chr(50504)
print(codePoint)
# output → 🏴

codePoint = chr(0xC548)
print(codePoint)
# output → 🏴
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../unicode
```

◉ Chapter relevan lainnya

- String

◉ Referensi

- https://docs.python.org/3/howto/unicode.html#:~:text=Python's%20string%20type%20uses%20the,character%20its%20own%20unique%20code.
- https://docs.python.org/3/howto/unicode.html?highlight=unicode%20howto#the-string-type

A.19. Number/Bilangan di Python

Sedikit mengulang tentang pembahasan chapter Tipe Data ➜ numerik, telah dijelaskan bahwa Python mengenal 3 jenis tipe data numerik, yaitu `int`, `float`, dan `complex`.

Pada chapter ini kita akan belajar lebih dalam tentang ketiganya.

A.19.1. Integer

Bilangan bulat direpresentasikan oleh tipe data `int` (kependekan dari *integer*). Cara deklarasi nilai bertipe data ini adalah menggunakan literal integer dimana angka ditulis langsung. Contoh:

```
angka1 = 24
angka2 = 13
total = angka1 + angka2
print(f"angka: {total}")
# output ➜ angka: 37
```

INFO

Ada yang unik dengan deklarasi bilangan bulat di Python. Diperbolehkan untuk menambahkan karakter underscore (`_`) di sela-sela angka. Misalnya:

```
angka3 = 100_2_345_123

print(f"angka3: {angka3}")
# output ➜ angka3: 1002345123
```

Variabel `angka3` di atas nilainya adalah sama dengan literal `1002345123`.

Literal integer *default*-nya adalah berbasis 10, contohnya seperti `24` dan `13` di atas keduanya adalah berbasis 10. Dan umumnya bahasa pemrograman lain juga sama.

A.19.2. Hexadecimal, Octal, Binary

Selain basis 10, bilangan bulat bisa dituliskan menggunakan basis lain, misalnya heksadesimal/oktal/biner, caranya dengan memanfaatkan *prefix* atau suatu awalan saat penulisan literalnya.

- Prefix literal untuk heksadesimal: `0x`
- Prefix literal untuk oktal: `0o`
- Prefix literal untuk biner: `0b`

```
angka = 140
angka_heksadesimal = 0x8c
angka_oktal = 0o214
angka_biner = 0b10001100

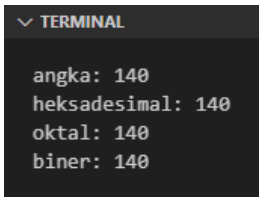
print(f"angka: {angka}")
# output → angka: 140

print(f"heksadesimal: {angka_heksadesimal}")
# output → heksadesimal: 140

print(f"oktal: {angka_oktal}")
# output → oktal: 140

print(f"biner: {angka_biner}")
# output → biner: 140
```

Nilai numerik (tanpa melihat basis deklarasinya) ketika di-print pasti dimunculkan dalam basis 10. Python otomatis meng-handle proses konversi antar basisnya. Pembuktiannya bisa dilihat pada output program di atas.



Dari perbandingan source code dan output, terlihat bahwa angka `8c` heksadesimal adalah sama dengan `214` oktal dan `10001100` biner.

Sedangkan untuk memunculkan angka-angka tersebut sesuai basisnya caranya adalah dengan menggunakan metode string formatting, dengan menambahkan suffix dalam penulisan variabel. Contoh:

```
angka = 140
angka_heksadesimal = 0x8c
angka_oktal = 0o214
angka_biner = 0b10001100

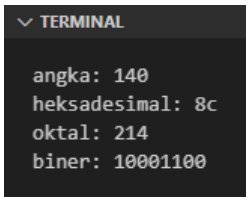
print(f"angka: {angka:d}")
# output → angka: 140

print(f"heksadesimal: {angka_heksadesimal:x}")
# output → heksadesimal: 8c

print(f"oktal: {angka_oktal:o}")
# output → oktal: 214

print(f"biner: {angka_biner:b}")
```

Output program:



Perbedaan lengkap tentang prefix dan suffix tiap basis bilangan bisa dicek pada tabel berikut:

Nama	Basis	Deklarasi		String formatting	
		Prefix	**Contoh**	***Suffix***	**Contoh**
Decimal	10	-	`angka1 = 24` `angka2 = 13`	`d` atau tanpa suffix	`print(f"angka1: {angka1}")` `print(f"angka2: {angka2:d}")`
Hexadecimal	16	`0x`	`hex1 = 0x8c` `hex2 = 0xff00c0`	`x`	`print(f"hex1: {hex1:x}")` `print(f"hex2: {hex2:x}")`
Octal	8	`0o`	`oct1 = 0o214` `oct2 = 0o605`	`o`	`print(f"oct1: {oct1:o}")` `print(f"oct2: {oct2:o}")`
Binary	2	`0b`	`bin1 = 0b1010` `bin2 = 0b110`	`b`	`print(f"bin1: {bin1:b}")` `print(f"bin2: {bin2:b}")`

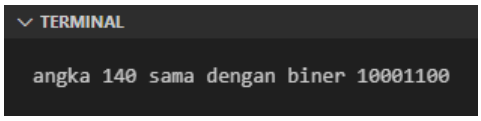
> *Lebih detailnya mengenai string formatting dibahas pada chapter String Formatting*

◉ Operasi perbandingan antar basis

Nilai bilangan integer meskipun dideklarasikan dengan basis biner, heksadesimal, atau oktal, nilai tersebut disimpan di variabel oleh Python dalam satu tipe data, yaitu `int`. Dari sifat tersebut, maka operasi logika perbandingan bisa dilakukan tanpa melihat basis numerik-nya, karena kesemuanya pasti bertipe `int`.

```
angka = 140
```

Output program:



◉ Print nilai numerik dalam basis tertentu menggunakan suffix

Angka numerik bisa di-print dalam basis apapun tanpa melihat deklarasinya menggunakan basis apa. Contohnya bisa dilihat pada program berikut, nilai oktal `214` di-print dalam 4 basis berbeda dengan memanfaatkan suffix tiap-tiap basis.

```
angka_oktal = 0o214

print(f"angka: {angka_oktal:d}")
# output → angka 140

print(f"heksadesimal: {angka_oktal:x}")
# output → heksadesimal: 8c

print(f"oktal: {angka_oktal:o}")
# output → oktal: 214

print(f"biner: {angka_oktal:b}")
# output → biner: 10001100
```

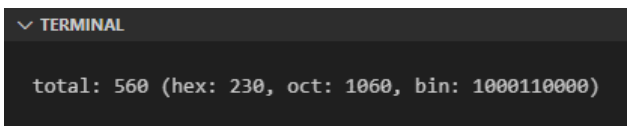
◉ Operasi aritmatika antar basis

Operasi aritmatika, apapun itu, juga bisa dilakukan antar basis. Contoh:

```
angka = 140
angka_heksadesimal = 0x8c
angka_oktal = 0o214
angka_biner = 0b10001100

total = angka + angka_heksadesimal + angka_oktal + angka_biner
print(f"total: {total} (hex: {total:x}, oct: {total:o}, bin: {total:b})")
```

Output program:



◉ Print nilai numerik dalam basis tertentu menggunakan fungsi

- Fungsi `oct()` digunakan untuk memunculkan nilai numerik dalam basis oktal dalam tipe data string.

```
int1 = oct(140)
print(f"int1: {int1}")
# output → int1: 0o214

int2 = oct(0x8c)
print(f"int2: {int2}")
# output → int2: 0o214
```

- Fungsi `hex()` digunakan untuk memunculkan nilai numerik dalam basis heksadesimal dalam tipe data string.

```
int3 = hex(140)
print(f"int3: {int3}")
# output → int3: 0x8c

int4 = hex(0b10001100)
print(f"int4: {int4}")
# output → int4: 0x8c
```

- Fungsi `bin()` digunakan untuk memunculkan nilai numerik dalam basis biner dalam tipe data string.

```
int5 = bin(140)
print(f"int5: {int5}")
# output → int5: 0b10001100

int6 = bin(0o214)
print(f"int6: {int6}")
# output → int6: 0b10001100
```

◉ Fungsi `int()`

Fungsi `int()` digunakan untuk mengkonversi data string berisi angka numerik berbasis apapun (selama basisnya 0 hingga 36) ke tipe data integer.

```
int1 = int("0b10001100", base=2)
print(f"int1: {int1}")
# output → int1: 140

int2 = int("0x8c", base=16)
print(f"int2: {int2}")
```

A.19.3. Floating point (*float*)

Bilangan *float* adalah bilangan yang memiliki angka dibelakang koma (atau titik untuk sistem angka luar negeri), misalnya angka `3.14` (yang di negara kita biasa ditulis dengan `3,14`).

> *Umumnya bilangan ini dikenal dengan nama **bilangan desimal**. Namun penulis tidak menggunakan istilah ini karena kata desimal pada chapter ini tidak selalu berarti bilangan dengan nilai dibelakang koma.*
>
> *Penulis memilih menggunakan istilah bilangan float.*

Untuk mendeklarasikan bilangan float, langsung saja tulis angka yang diinginkan dengan penanda dibelakang koma adalah tanda titik. Misalnya:

```
angka_float = 3.141592653589
print(f"angka float: {angka_float}")
# output → angka float: 3.141592653589
```

Khusus untuk bilangan float yang nilai belakang komanya adalah `0` bisa dituliskan dengan tanpa menuliskan angka `0`-nya. Contoh:

```
angka_float = 3.
print(f"angka float: {angka_float}")
# output → angka float: 3.0
```

◉ Pembulatan / *rounding*

Pembulatan nilai di belakang koma dilakukan menggunakan fungsi `round()`. Panggil fungsi tersebut, sisipkan data float yang ingin dibulatkan sebagai argument pertama fungsi dan jumlah digit belakang koma sebagai argument ke-dua.

```
pi = 3.141592653589

n1 = round(pi, 2)
print(f"n1: {n1}")
# output → n1: 3.14

n2 = round(pi, 5)
print(f"n2: {n2}")
# output → n2: 3.14159
```

Selain fungsi `round()` ada juga 2 fungsi milik module `math` yang cukup berguna untuk keperluan

pembulatan ke-bawah atau ke-atas.

- Pembulatan ke-bawah.

```
import math

n3 = math.floor(pi)
print(f"n3: {n3}")
# output → n3: 3
```

- Pembulatan ke-atas

```
import math

n4 = math.ceil(pi)
print(f"n4: {n4}")
# output → n4: 4
```

Kedua fungsi di atas menghasilkan nilai balik bertipe `int`, tidak seperti fungsi `round()` yang mengembalikan nilai float.

◉ Pembulatan float dengan string formatting

Fungsi `round()`, `math.floor()`, dan `math.ceil()` menerima data float sebagai argument pemanggilan fungsi dan mengembalikan nilai baru setelah dibulatkan.

Jika pembulatan hanya diperlukan saat printing saja, lebih efektif menggunakan metode string formatting. Caranya, tulis variabel dalam string formatting lalu tambahkan suffix `:.{n}f` dimana `n` diisi dengan jumlah digit belakang koma. Sebagai contoh, suffix `:.2f` menghasilkan string berisi data float dengan 2 digit dibelakang koma.

Contoh versi lebih lengkap:

```
angka_float = -3.141592653589

print(f"angka float: {angka_float:.2f}")
# output → angka float: -3.14

print(f"angka float: {angka_float:.3f}")
# output → angka float: -3.142

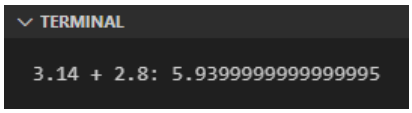
print(f"angka float: {angka_float:.4f}")
# output → angka float: -3.1416
```

◉ Karakteristik *floating point*

Hampir di semua bahasa pemrograman yang ada, tipe data float (atau sejenisnya) memiliki satu sifat unik dimana angka belakang koma tidak tersimpan secara pasti informasinya digitnya.

Agar lebih jelas, silakan run program berikut:

```
n = 3.14 + 2.8
print(f"3.14 + 2.8: {n}")
```



Ajaib bukan? Operasi aritmatika `3.14 + 2.8` menghasilkan output `5.9399999999999995`.

Namun tidak usah khawatir, ini bukan error. Di belakang layar, komputer memang selalu menyimpan informasi angka belakang koma float secara tidak pasti (tidak *fixed*).

Untuk menampilkan angka fixed-nya, gunakan suffix `:f`. Contoh:

```
n = 3.14 + 2.8

print(f"3.14 + 2.8: {n:f}")
# output → 3.14 + 2.8: 5.940000
```

Manfaatkan suffix `:{n}f` untuk menampilkan jumlah digit belakang koma (`n`) sesuai keinginan. Misalnya:

```
n = 3.14 + 2.8
print(f"3.14 + 2.8: {n:.2f}")
# output → 3.14 + 2.8: 5.94
```

> *Lebih detailnya mengenai string formatting dibahas pada chapter String Formatting*

◉ Konversi tipe data via fungsi `float()`

Fungsi `float()` digunakan untuk mengkonversi suatu nilai menjadi float.

```
number = 278885
float_num1 = float(number)
```

Fungsi ini cukup berguna untuk dipergunakan dalam kebutuhan konversi tipe data, misalnya dari string ke float.

```
text = '278885.666'
float_num2 = float(text)
print(f"float_num2: {float_num2}")
# output → float_num: 278885.666
```

◉ Notasi float *exponential*

Deklarasi nilai float bisa ditulis menggunakan literal float dengan notasi exponsensial, yaitu `{f}e{n}` atau `{f}e+{n}` dimana literal tersebut menghasilkan angka `f * (10 ^ n)`.

Agar lebih jelas, langsung ke praktek saja.

```
float1 = 2e0
print(f"float1: {float1}")
# output → float1: 2.0

float2 = 577e2
print(f"float2: {float2}")
# output → float2: 57700.0

float3 = 68277e+6
print(f"float3: {float3}")
# output → float3: 68277000000.0
```

Penjelasan:

- Notasi `2e0` artinya adalah `2.0 * (10 ^ 0)`. Nilai tersebut ekuivalen dengan `2.0`
- Notasi `577e2` artinya adalah `577.0 * (10 ^ 2)`. Nilai tersebut ekuivalen dengan `57700.0`
- Notasi `68277e+6` artinya adalah `68277.0 * (10 ^ 6)`. Nilai tersebut ekuivalen dengan `68277000000.0`

Nilai `n` setelah huruf `e` jika diisi dengan nilai negatif menghasilkan output dengan formula `f / (10 ^ n)`. Contoh:

```
float4 = 6e-3
print(f"float4: {float4}")
# output → float4: 0.006
```

A.19.4. Bilangan *complex*

Bilangan *complex* adalah bilangan yang isinya merupakan kombinasi bilangan real dan bilangan imajiner,

contohnya seperti `120+3j`.

Informasi bilangan real pada *complex number* bisa dimunculkan menggunakan property `real` sedangkan informasi bilangan imajinernya menggunakan property `imag`. Contoh:

```
angka_complex = 120+3j
print(f"angka complex: {angka_complex}")
# output → angka complex: (120+3j)

r = angka_complex.real
print(f"angka real: {r}")
# output → angka real: 120.0

i = angka_complex.imag
print(f"angka imajiner: {i}")
# output → angka imajiner: 3.0
```

◉ Fungsi `complex()`

Fungsi `complex()` adalah digunakan sebagai alternatif cara membuat bilangan kompleks.

Sebagai contoh, bilangan `120+3j` jika dituliskan menggunakan fungsi `complex()` maka penulisannya seperti berikut:

```
angka_complex = complex(120, 3)
print(f"angka complex: {angka_complex}")
# output → angka complex: (120+3j)
```

◉ Operasi aritmatika bilangan *complex*

Seperti wajarnya suatu bilangan, nilai *complex* bisa dimasukan dalam operasi matematika standar, misalnya:

```
cmp1 = 120-2j
cmp2 = -19+4j

res = cmp1 + cmp2
print(f"angka complex: {res}")
# output → angka complex: (101+2j)

res = cmp1 + cmp2 + 23
print(f"angka complex: {res}")
# output → angka complex: (124+2j)

res = (cmp1 + cmp2 + 23) / 0.5
print(f"angka complex: {res}")
```

Penjelasan:

- Operasi antar bilangan kompleks akan melakukan perhitungan terhadap bilangan real dan juga bilangan imajinernya.
- Operasi antara bilangan kompleks vs. bilangan real, menghasilkan dua operasi aritmatika:
  - Menghitung bilangan real bilangan complex vs bilangan real
  - Dan juga menghitung bilangan imajiner vs bilangan real

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../number-bilangan
```

◉ Chapter relevan lainnya

- Variabel
- Tipe Data
- String ➜ formatting

◉ TBA

- nan
- inf

◉ Referensi

- https://pythondev.readthedocs.io/numbers.html
- https://note.nkmk.me/en/python-nan-usage/
- https://note.nkmk.me/en/python-inf-usage/

A.20. Python Slice (Data Sequence Slicing)

Pada chapter ini kita akan belajar tentang penerapan teknik slice pada data sequence.

A.20.1. Pengenalan slice

Teknik slice atau slicing digunakan untuk mengakses sekumpulan element/item dari data sequence sesuai dengan index yang diinginkan. Data sequence sendiri adalah klasifikasi tipe data yang berisi kumpulan data terurut atau sekuensial. Yang termasuk dalam tipe data sequence adalah list, range, tuple, dan string.

Operasi slice mengembalikan data bertipe sama seperti data aslinya, sedangkan isi sesuai dengan index yang ditentukan.

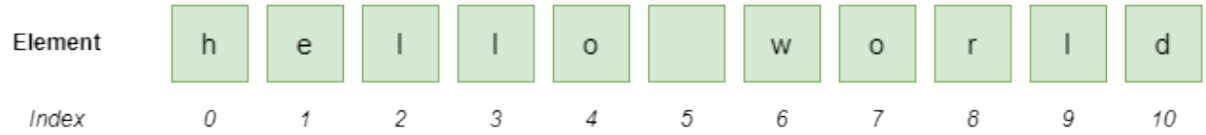
Salah satu penerapan slice adalah dengan memanfaatkan notasi `data[start:end]` atau `data[start:end:step]`.

- `start` adalah index awal slicing. Misalkan index `start` adalah `2` maka slicing dimulai dari element index ke-2.
- `end` adalah index akhir slicing. Misalkan index `end` adalah `5` maka slicing berakhir **sebelum** element index ke-5 (yang berarti element ke-4).
- `step` by default nilainya `1`, kegunaannya untuk menentukan apakah element yang dikembalikan adalah setiap `step` index.

Lanjut praktek. Pada contoh berikut disiapkan variabel `data_str` berisi string `hello world` yang kemudian akan di-slice datanya.

```
data_str = "hello world"
print(data_str)
# output → hello world
```

Variabel `data_str` visualisasinya dalam bentuk *sequence* kurang lebih seperti ini. Lebar element data adalah `11` dengan index awal `0` dan index akhir `10`.



Ok, sekarang kita coba slice `data_str`:

- Slicing element index ke-0 hingga ke-2, maka notasinya adalah `data_str[0:3]`. Perlu diketahui bahwa `end` diisi dengan nilai `index-1`, jadi jika ingin mengambil element hingga index ke-2 maka nilai `end` adalah `3`.

  ```
  data_str = "hello world"

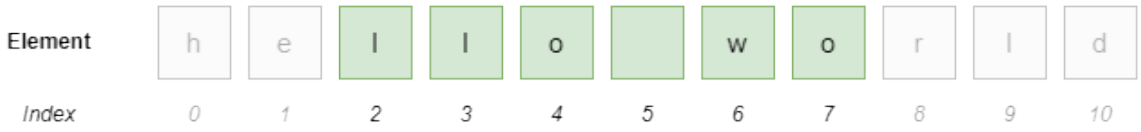
  slice1 = data_str[0:3]
  print(slice1)
  # output → hel
  ```



- Slicing element index ke-2 hingga ke-7, maka notasinya adalah

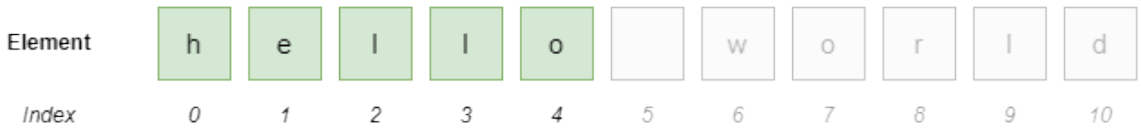
`data_str[2:8]`.

```
slice2 = data_str[2:8]
print(slice2)
# output → llo wo
```



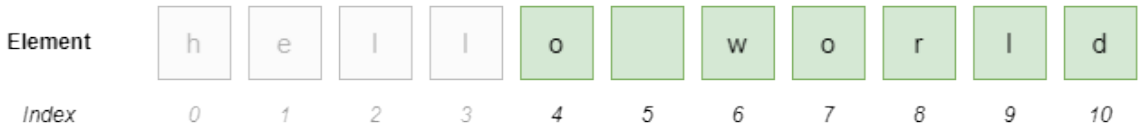
- Slicing element hingga index ke-4, maka notasinya adalah `data_str[:5]`. Nilai `start` jika tidak diisi maka default-nya adalah `0`. Notasi tersebut adalah ekuivalen dengan `data_str[0:5]`.

```
slice3 = data_str[:5]
print(slice3)
# output → hello
```



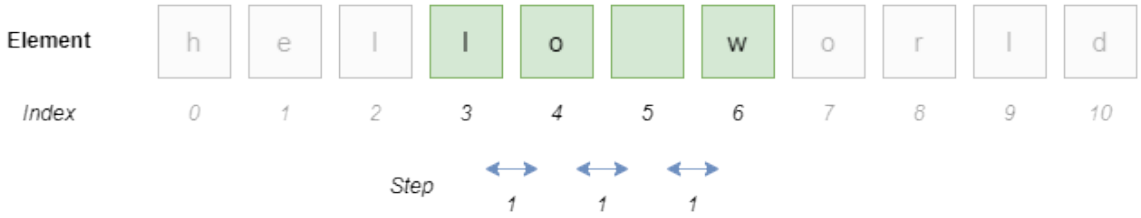
- Slicing element dimulai index ke-4, maka notasinya adalah `data_str[4:]`. Nilai `end` jika tidak diisi maka default-nya adalah nilai jumlah element data (ekuivalen dengan notasi `data_str[4:len(data_str)]`).

```
slice4 = data_str[4:]
```



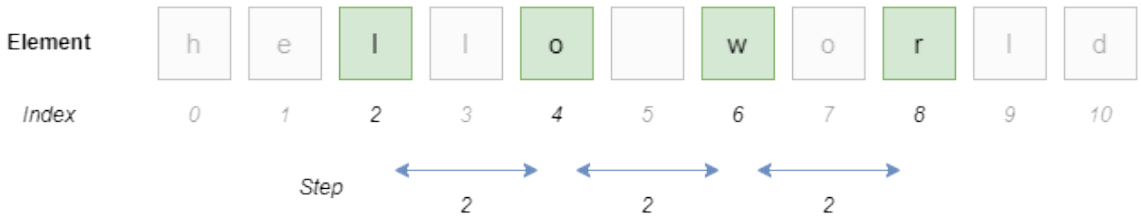
- Slicing element dimulai index ke-3 hingga ke-6 dengan element yang dikembalikan adalah setiap 1 element, maka notasinya adalah `data_str[3:7:1]`.

```
slice5 = data_str[3:7:1]
print(slice5)
# output → lo w
```



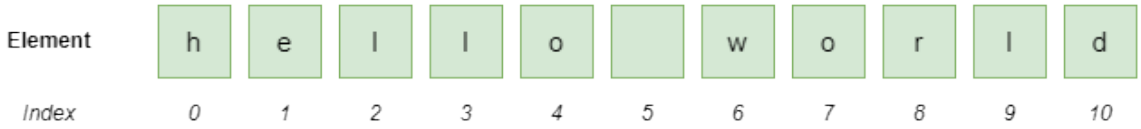
- Slicing element dimulai index ke-2 hingga ke-8 dengan ketentuan element yang dikembalikan adalah setiap 2 element, notasinya: `data_str[2:9:2]`.

```
slice6 = data_str[2:9:2]
print(slice6)
# output → lowr
```



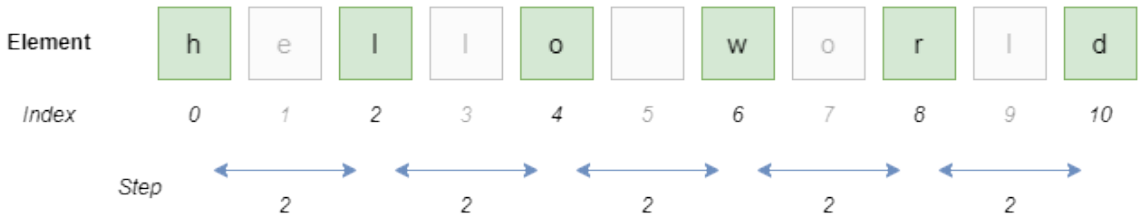
- Slicing seluruh element bisa dilakukan dengan notasi `data_str[:]`. Notasi tersebut adalah ekuivalen dengan `data_str[0:len(data_str)]`.

```
slice7 = data_str[:]
print(slice7)
# output → hello world
```



- Slicing seluruh element dengan ketentuan element yang dikembalikan adalah setiap 2 element, ditulis dengan notasi `data_str[::2]`. Notasi tersebut adalah ekuivalen dengan `data_str[0:len(data_str):2]`.

```
slice8 = data_str[::2]
print(slice8)
# output → hlowrd
```



◉ Tentang slicing seluruh element

Slicing seluruh element bisa dilakukan dengan notasi `data[0:len(data)]` atau `data[0:len(data):1]`. Sebagai contoh, 3 statement printing tuple berikut memunculkan output yang sama meskipun data tuple yang ditampilkan adalah dari variabel yang berbeda.

```
data_tuple = (1, 3, 5, 7, 9, 11, 13, 14)
print(data_tuple)
# output → (1, 3, 5, 7, 9, 11, 13, 14)

tuple1 = data_tuple[0:len(data_tuple)]
print(tuple1)
# output → (1, 3, 5, 7, 9, 11, 13, 14)

tuple2 = data_tuple[0:len(data_tuple):1]
print(tuple2)
# output → (1, 3, 5, 7, 9, 11, 13, 14)
```

Ok, lalu kenapa harus menggunakan teknik ini? padahal operasi assignment data tuple ke variabel baru jauh lebih mudah, misalnya:

```
tuple3 = data_tuple
print(tuple3)
# output → (1, 3, 5, 7, 9, 11, 13, 14)
```

Statement assignment `tuple3` di atas isinya adalah sama dengan data hasil operasi slicing `tuple1` dan `tuple2`, namun *reference*-nya adalah berbeda.

> *Pembahasan detail mengenai reference ada di chapter Object ID & Reference*

A.20.2. Fungsi `slice()`

Notasi penulisan slice bisa disimpan pada suatu variabel dengan memanfaatkan fungsi `slice()`. Nilai `start`, `end`, dan `step` dijadikan argument pemanggilan fungsi tersebut dengan notasi `slice(start, end)` atau `slice(start, end, step)`.

Pada contoh berikut, perhatikan bagaimana perbedaan slicing pada `list1`, `list2`, dan `list3`:

```
data_list = [2, 4, 6, 7, 9, 11, 13]
print(data_list)
# output → [2, 4, 6, 7, 9, 11, 13]

list1 = data_list[2:6:1]
print(list1)
# output → [6, 7, 9, 11]

list2 = data_list[slice(2, 6, 1)]
print(list2)
# output → [6, 7, 9, 11]

sl = slice(2, 6)
list3 = data_list[sl]
print(list3)
# output → [6, 7, 9, 11]
```

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../slice
```

◉ Chapter relevan lainnya

- List
- Tuple
- String
- Object ID & Reference

◉ TBA

- Negative index slicing

◉ Referensi

- https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range
- https://python-reference.readthedocs.io/en/latest/docs/functions/slice.html
- https://stackoverflow.com/questions/509211/how-slicing-in-python-works

A.21. Python Object ID & Reference

Pada chapter ini kita akan belajar tentang beberapa hal yang berhubungan dengan object/data dan reference, diantaranya:

- Apa itu *identifier* data (object ID)
- Bagaimana Python mengelola data
- Apa yang terjadi sewaktu data di-*assign* ke variabel lain
- Dan juga peran ID dalam data reference dan operasi slicing

A.21.1. Object ID

Di Python, semua object atau data memiliki identifier, yaitu angka unik yang merepresentasikan data tersebut. Sebagai contoh, pada kode berikut nilai numerik `24` tersimpan pada variabel `number`, ID nya adalah `140728206353928`.

```
number = 24

print("data:", number)
# output → data: 24

identifier = id(number)
print("id:", identifier)
# output → id: 140728206353928
```

Output program di atas jika di-run:



Object ID dialokasikan oleh Python saat program dijalankan, dan nilainya bisa saja berbeda setiap eksekusi program.

◉ Fungsi `id()`

Fungsi `id()` digunakan untuk melihat ID suatu data. Cara penggunaannya cukup mudah, tulis fungsi lalu sisipkan data yang ingin dicek ID-nya sebagai parameter pemanggilan fungsi.

A.21.2. *Reference* / alamat memori data

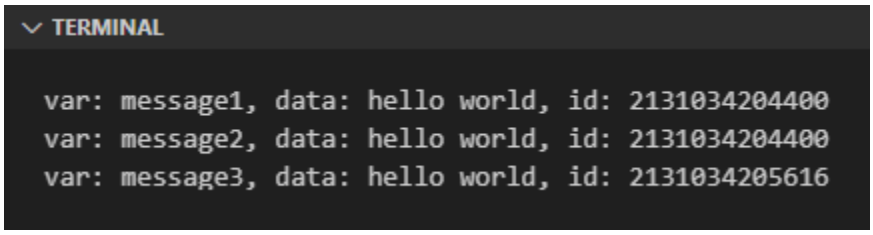
Perlu diketahui bahwa Identifier merupakan metadata informasi yang menempel pada data atau object, **bukan menempel ke variabel**. Data yang sama jika di-assign ke banyak variabel, maka pengecekan ID pada semua variabel tersebut mengembalikan ID yang sama.

> *Reference pada konteks programming artinya adalah referensi suatu data ke alamat memori.*

Coba pelajari kode berikut. Variabel `message1` berisi string `hello world`. String tersebut kemudian di-assign ke `message2`. Selain itu ada juga variabel `message3` berisi string yang sama persis tapi dari deklarasi literal berbeda.

```
message1 = "hello world"
message2 = message1
```

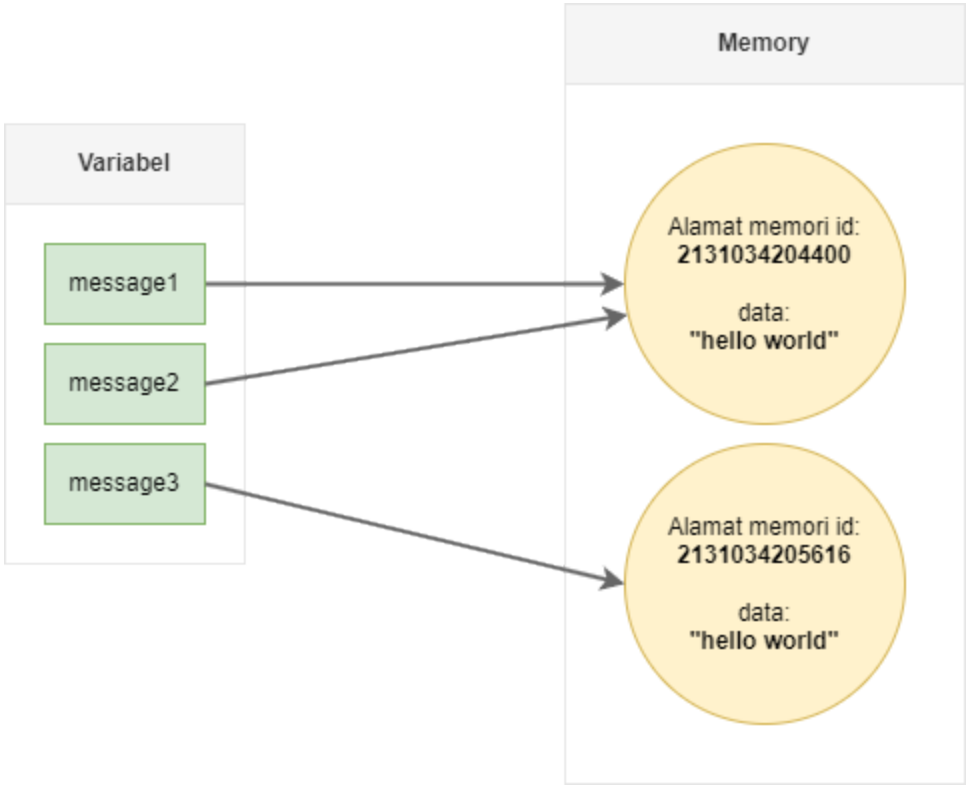
Jalankan program, lalu perhatikan `id`-nya:



Penjelasan program:

- Ketiga variabel di atas berisi data string yang sama persis, yaitu `hello world`.
- Identifier data string pada variabel `message1` dan `message2` adalah sama. Hal ini wajar karena memang variabel `message2` mendapatkan data dari `message1`. Yang terjadi di belakang layar, kedua variabel tersebut menampung nilai yang tersimpan di alamat memori yang sama (*reference*-nya sama).
- Identifier data string pada variabel `message3` adalah berbeda dibandingkan `message1` maupun `message2`, hal ini karena meskipun isi string ketiga variabel sama, dua diantaranya adalah tersimpan di alamat memory yang berbeda.

Ilustrasi dalam bentuk grafiknya kurang lebih seperti ini:



Variabel hanya merupakan media untuk pengaksesan data. Data sendiri tersimpan-nya adalah di memory. Sangat mungkin ada situasi dimana satu data direpresentasikan oleh lebih dari 1 variabel. Contohnya seperti `message1` dan `message2`.

A.21.3. Operasi logika via keyword `is`

Kita sudah cukup sering menggunakan operator `==` dan operator logika lainnya untuk membandingkan dua buah nilai. Dalam penerapannya, operator-operator tersebut akan membandingkan isi data, **bukan identifier**-nya.

Pada kode berikut ini, 3 variabel yang telah dibuat sebelumnya digunakan pada statement perbandingan.

```
message1 = "hello world"
message2 = message1
message3 = "hello world"

print(f"message1 ({id(message1)}) == message2 ({id(message2)}) ➞
{message1 == message2}")
# output ➞ message1 (2131034204400) == message2 (2131034204400) ➞ True

print(f"message1 ({id(message1)}) == message3 ({id(message3)}) ➞
{message1 == message3}")
# output ➞ message1 (2131034204400) == message3 (2131034205616) ➞ True

print(f"message2 ({id(message2)}) == message3 ({id(message3)}) ➞
{message2 == message3}")
# output ➞ message2 (2131034204400) == message3 (2131034205616) ➞ True
```

Hasil dari ke-3 statement perbandingan adalah `True`, karena memang isi data-nya adalah sama, yaitu string `hello world`.

Selanjutnya coba bandingkan dengan satement operator perbandingan menggunakan keyword `is`. Keyword `is` akan melakukan pengecekan apakah identifier suatu data adalah sama dengan yang dibandingkan (yang di-cek adalah identifier-nya, bukan isi datanya).

```
message1 = "hello world"
message2 = message1
message3 = "hello world"

print(f"message1 ({id(message1)}) is message2 ({id(message2)}) ➞
{message1 is message2}")
# output ➞ message1 (2131034204400) is message2 (2131034204400) ➞ True

print(f"message1 ({id(message1)}) is message3 ({id(message3)}) ➞
{message1 is message3}")
# output ➞ message1 (2131034204400) is message3 (2131034205616) ➞ False
```

Hasilnya:

- Statement `message1 is message2` menghasilkan `True` karena kedua variabel tersebut merepresentasikan satu data yang sama (tersimpan di alamat memory yang sama).
- Statement perbandingan lainnya menghasilkan `False` karena identifier data adalah berbeda meskipun isi data adalah sama.

◉ Lebih dalam mengenai korelasi operasi assignment dan object ID

Mari kita modifikasi lagi kode sebelumnya agar lebih terlihat jelas efek dari operasi assignment ke object ID.

Pada kode berikut, kita coba tampilkan hasil operasi perbandingan menggunakan keyword `is`. Kemudian nilai variabel `message2` diubah dan dibandingkan ulang. Setelah itu, nilai `message3` diubah untuk diisi dengan nilai dari `message2`.

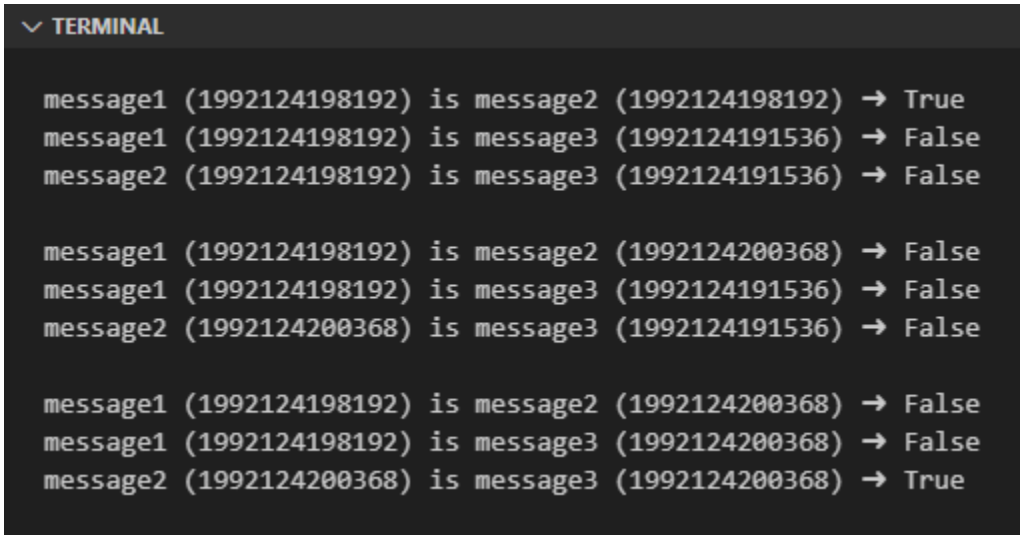
```
message1 = "hello world"
message2 = message1
message3 = "hello world"

print(f"message1 ({id(message1)}) is message2 ({id(message2)}) →
{message1 is message2}")
print(f"message1 ({id(message1)}) is message3 ({id(message3)}) →
{message1 is message3}")
print(f"message2 ({id(message2)}) is message3 ({id(message3)}) →
{message2 is message3}")

message2 = "hello world"

print(f"message1 ({id(message1)}) is message2 ({id(message2)}) →
{message1 is message2}")
```

Output program:

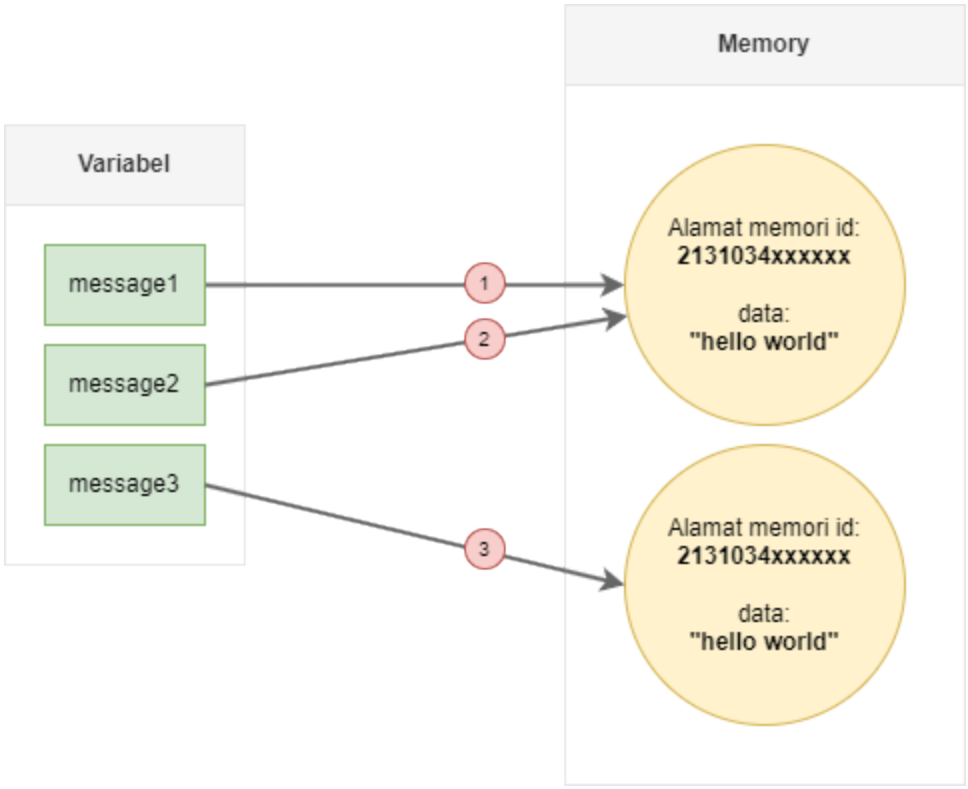


Bisa dilihat, pada bagian akhir, statement `message2 is message3` menghasilkan nilai `True` karena pada baris tersebut isi data `message3` sudah diganti dengan data dari `message2`, menjadikan kedua variabel menampung satu data yang sama, dan tersimpan di alamat memory yang sama.

Ilustrasi perubahan data pada program di atas dalam bentuk grafik bisa dilihat pada penjelasan berikut:

- Fase 1:

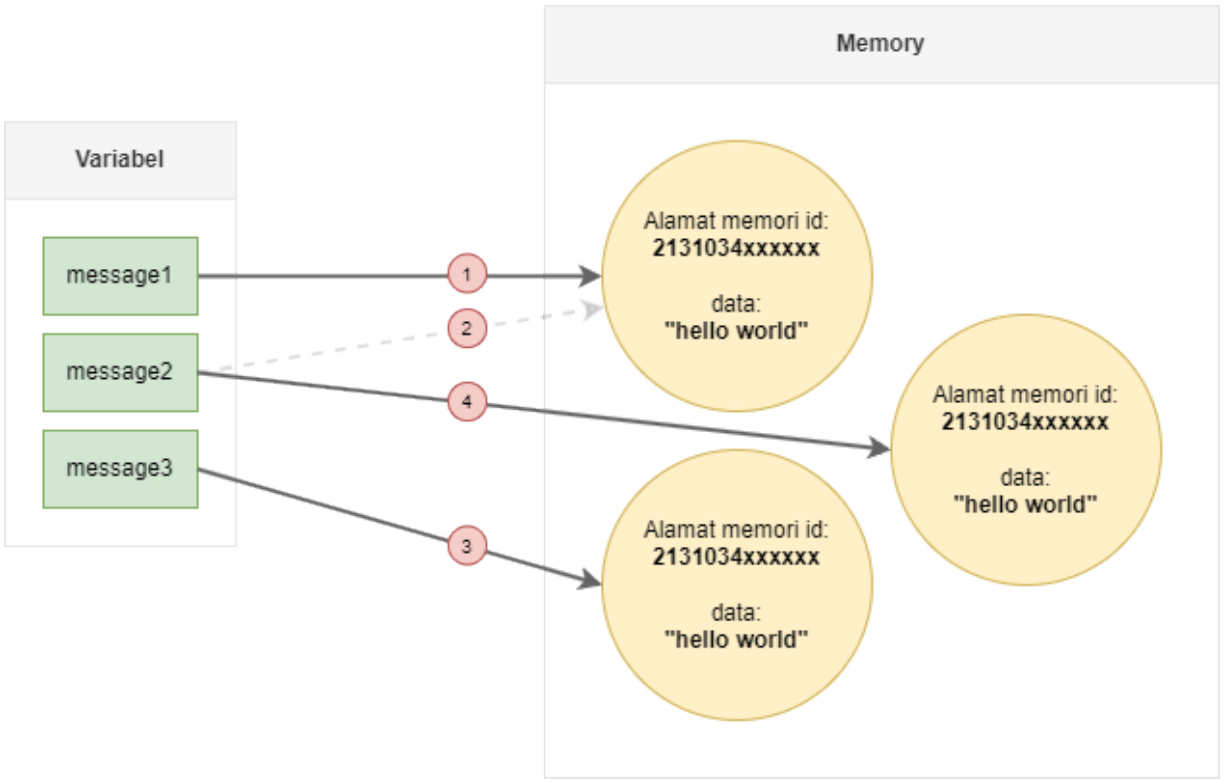
```
message1 = "hello world"    # statement 1
message2 = message1         # statement 2
message3 = "hello world"    # statement 3
```



- Fase 2:

```
message1 = "hello world"    # statement 1
message2 = message1         # statement 2
message3 = "hello world"    # statement 3

message2 = "hello world"    # statement 4
```

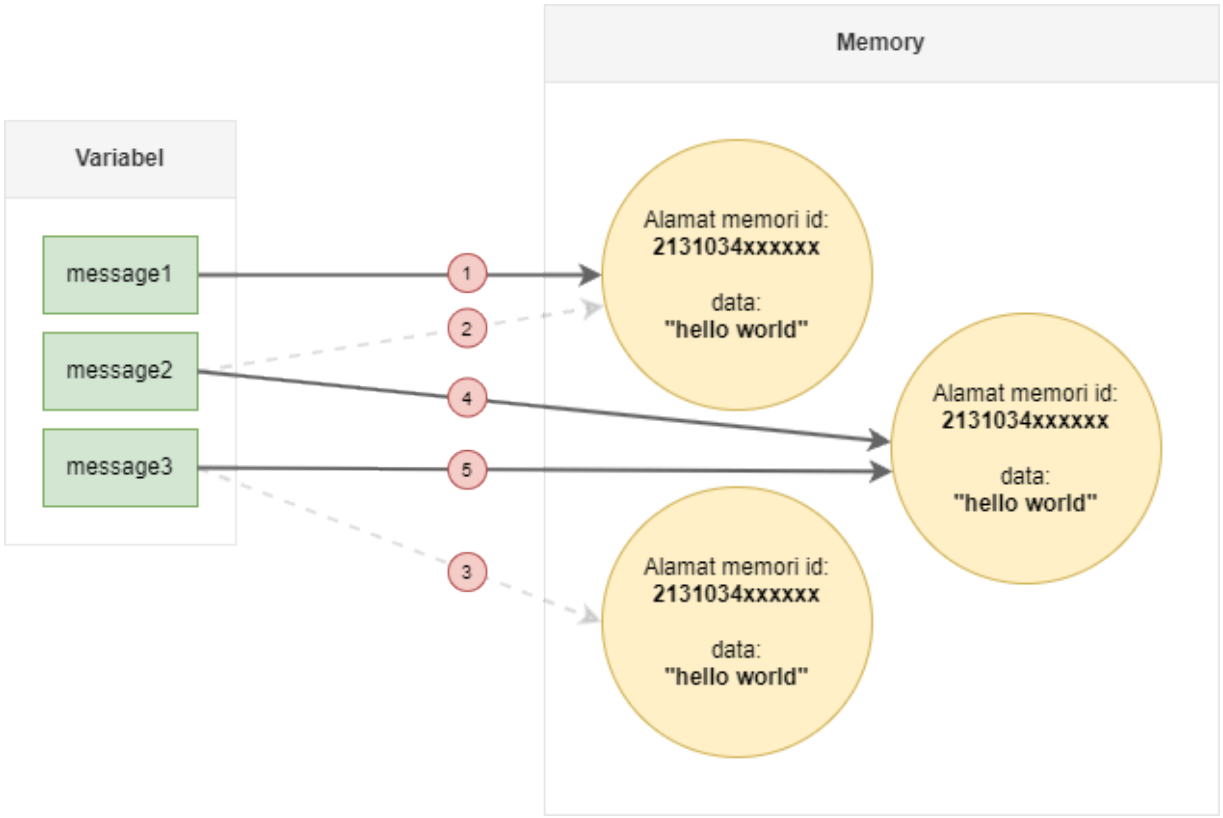


- Fase 3:

```
message1 = "hello world"    # statement 1
message2 = message1         # statement 2
message3 = "hello world"    # statement 3

message2 = "hello world"    # statement 4

message3 = message2         # statement 5
```



A.21.4. Reference data sequence

Data sequence (entah itu list, string, ataupun lainnya) kalau dilihat dari segi manajemen memory-nya adalah berbeda dibandingkan dengan bagaimana data dalam array di-manage di bahasa lain.

List di Python tersimpan pada satu alamat memory. Tidak seperti slice/array pada bahasa Go (misalnya), yang setiap element-nya merupakan *reference*.

Perhatikan kode berikut, variabel `numbers1` berikut di-assign ke variabel bernama `numbers2`, maka identifier kedua variabel adalah sama karena mengarah ke satu data yang sama.

```
numbers1 = [1, 2, 3, 4]
print("numbers1", id(numbers1), numbers1)
# output → numbers1 2269649131136 [1, 2, 3, 4]

numbers2 = numbers1
print("numbers1", id(numbers1), numbers1)
# output → numbers1 2269649131136 [1, 2, 3, 4]

print("numbers2", id(numbers2), numbers2)
# output → numbers2 2269649131136 [1, 2, 3, 4]
```

Perlu diingat bahwa data sequence bukan data *atomic* seperti `int` yang isinya sangat spesifik, yaitu nilai numerik. Data sequence merupakan data kolektif dengan isi banyak element. Data sequence isi-isi element-nya bisa dimutasi atau diubah tanpa men-*trigger* alokasi alamat memory baru (identifier-nya adalah tetap).

Sebagai contoh, pada program berikut, variabel `numbers1` dan `numbers2` reference-nya adalah sama. Apa yang akan terjadi ketika ada penambahan element baru di salah satu variabel?

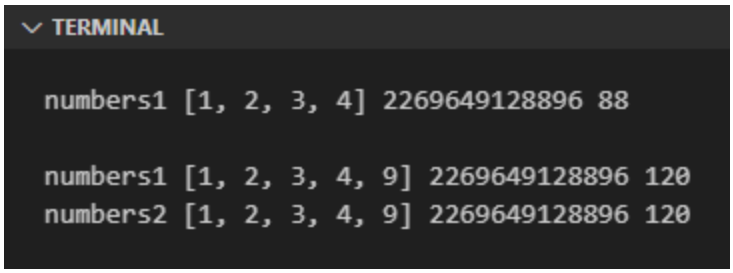
```
import sys

numbers1 = [1, 2, 3, 4]
print("numbers1", numbers1, id(numbers1), sys.getsizeof(numbers1))

numbers2 = numbers1
numbers2.append(9)

print("numbers1", numbers1, id(numbers1), sys.getsizeof(numbers1))
print("numbers2", numbers1, id(numbers2), sys.getsizeof(numbers2))
```

Output program:



Dari output eksekusi program terlihat bahwa data `numbers1` ikut berubah setelah `numbers2` diubah lewat penambahan element baru (via method `append()`). perubahan di kedua variabel terjadi karena memang keduanya merepresentasikan satu data yang sama, reference-nya adalah sama.

Terlihat juga ID kedua variabel juga tetap meskipun setelah isi element-nya di-ubah.

◉ Fungsi `sys.getsizeof()`

Fungsi `getsizeof()` tersedia dalam module `sys`, kegunaannya untuk melihat ukuran data dalam *byte*.

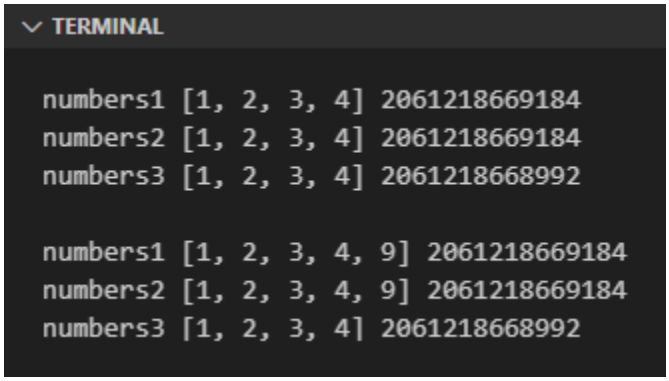
A.21.5. Reference pada data hasil slicing

Bagaimana dengan slicing, apakah ada efeknya ke object ID dan *reference* data? Yap, ada. Coba saja test program berikut:

```
numbers1 = [1, 2, 3, 4]
numbers2 = numbers1
numbers3 = numbers1[:]

print("numbers1", numbers1, id(numbers1)) # statement 1
print("numbers2", numbers2, id(numbers2)) # statement 2
print("numbers3", numbers3, id(numbers3)) # statement 3
```

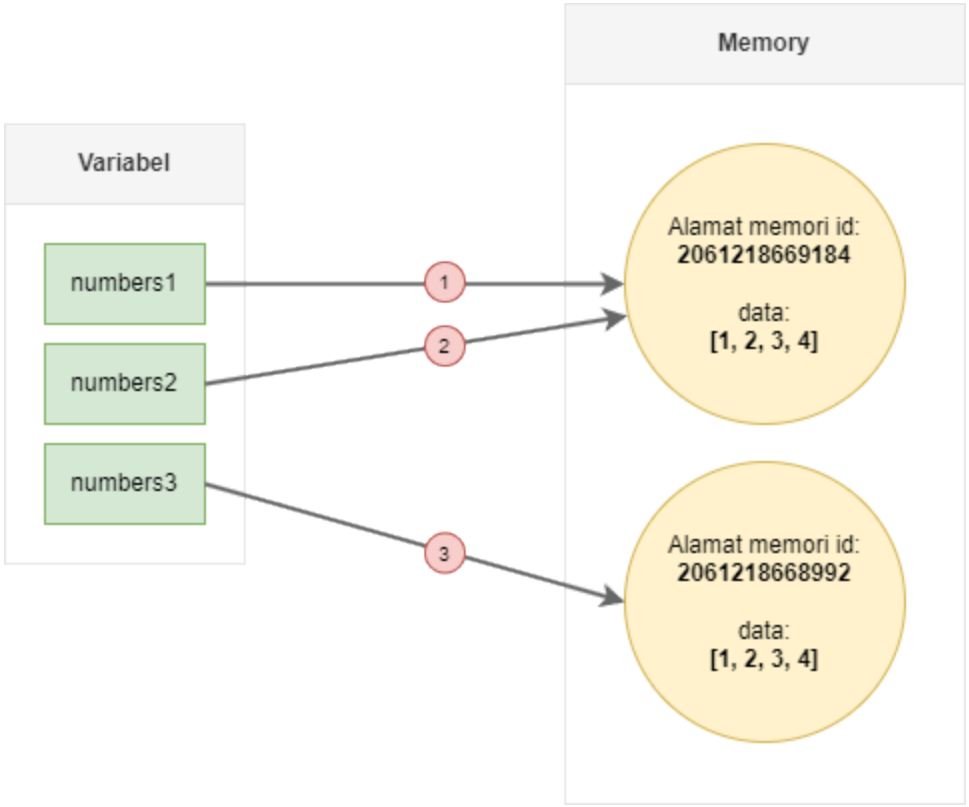
Kemudian lihat hasilnya:



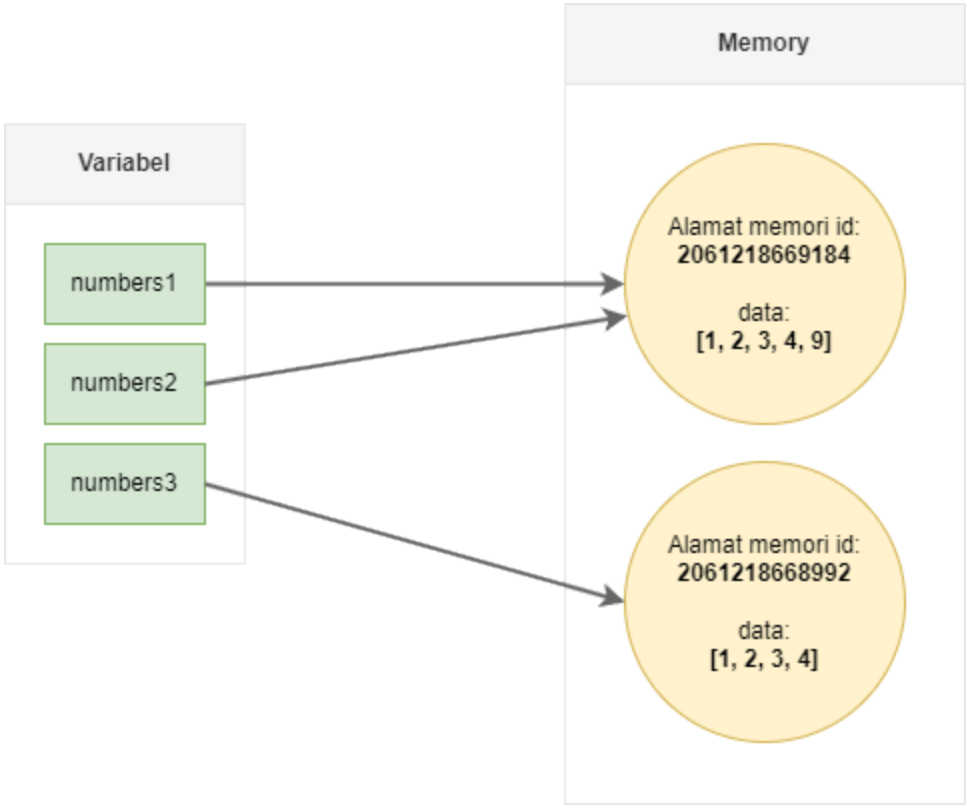
Penjelasan:

- Ketika suatu data sequence di assign dari satu variabel ke variabel lain, maka keduanya memiliki reference yang sama.
- Namun, jika assignment tersebut merupakan hasil operasi slice, maka data hasil slicing merupakan data baru yang tersimpan di alamat memory baru. Jadi ID-nya sudah pasti berbeda.

Ilustrasi yang terjadi pada saat `statement 1, 2, 3` dieksekusi:



Lalu setelah angka `9` di-append ke `numbers2`:



A.21.6. Catatan tambahan tentang object ID

Ada hal unik/spesial yang berhubungan dengan object ID yang wajib untuk diketahui, diantaranya:

◉ Object ID data numerik

Python meng-*cache* informasi data numerik integer `-5` hingga `256`, karena alasan ini terkadang ID suatu data numerik integer adalah sama (meskipun tidak selalu).

```
n1 = 12
n2 = 12
```

◉ Object ID data string

Python meng-*cache* informasi data string `a` hingga `z`, karena alasan ini terkadang ID suatu data string adalah sama (meskipun tidak selalu).

```
str1 = 'Indonesia'
str2 = "Indonesia"

print(f"id str1: {id(str1)}, id str2: {id(str2)}")
# output → id str1: 133983722110320, id str2: 133983722110320

print(f"str1 == str2: {str1 == str2}")
# output → str1 == str2: True

print(f"str1 is str2: {str1 is str2}")
# output → str1 is str2: True
```

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../object-id-reference
```

◉ Chapter relevan lainnya

- Slice

◉ TBA

- Hashable

◉ Referensi

- https://stackoverflow.com/questions/45335809/python-pass-by-reference-and-slice-assignment
- https://stackoverflow.com/a/15172182/1467988

A.22. Python Function / Fungsi

Penerapan fungsi di Python cukup mudah dan pada chapter ini kita akan memulai untuk mempelajarinya.

O iya, chapter ini merupakan chapter pembuka pembahasan topik fungsi. Ada banyak hal yang perlu dipelajari, oleh karena itu penulis memutuskan untuk memecah chapter menjadi beberapa bagian.

A.22.1. Pengenalan Fungsi

Function atau fungsi adalah kode program yang terisolasi dalam satu blok kode, yang bisa dipanggil sewaktu-waktu. Fungsi memiliki beberapa atribut seperti nama fungsi, isi fungsi, parameter/argument, dan nilai balik.

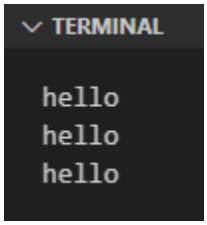
Pembuatan fungsi dilakukan dengan keyword `def` diikuti dengan nama fungsi, lalu di bawahnya ditulis body/isi fungsi. Sebagai contoh pada kode berikut fungsi `say_hello()` dideklarasikan dengan isi adalah sebuah statement yang menampilkan text `hello`.

```
def say_hello():
    print("hello")
```

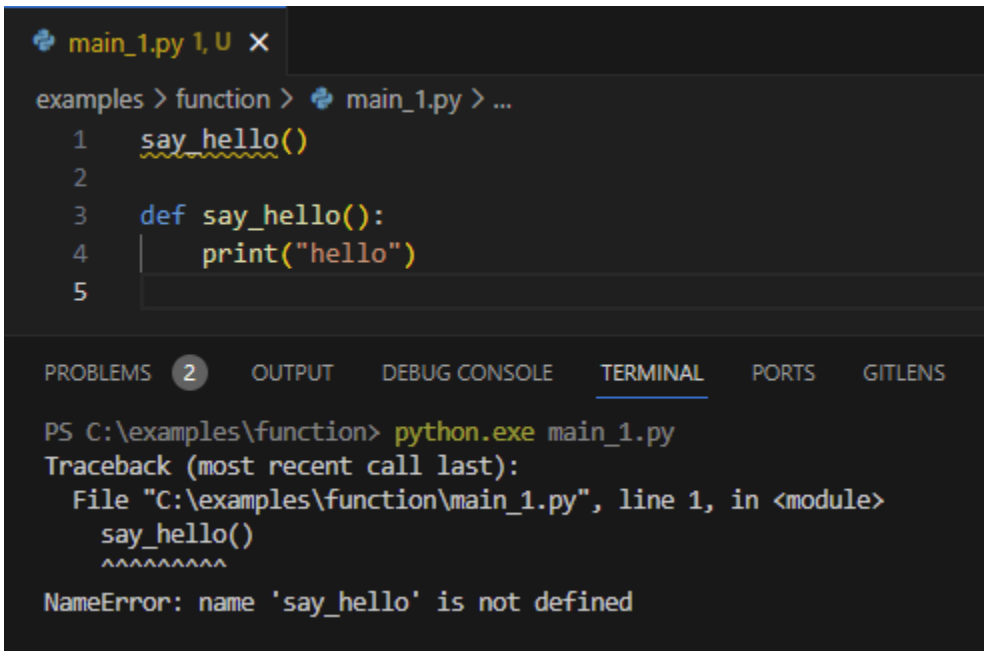
Setelah di deklarasikan, fungsi bisa dipanggil berkali-kali. Misalnya pada contoh berikut fungsi `say_hello()` dipanggil 3x.

```
def say_hello():
```

Output program sewaktu di-run:



Suatu fungsi hanya bisa diakses atau dipanggil setelah fungsi tersebut dideklarasikan (statement pemanggilan fungsi harus dibawah statement deklarasi fungsi). Jika fungsi dipaksa digunakan sebelum dideklarasikan hasilnya error.



Pada contoh di atas, selain `say_hello()` sebenarnya ada satu buah fungsi lagi yang digunakan pada contoh, yaitu `print()`. Fungsi `print()` dideklarasikan dalam Python Standard Library (stdlib). Sewaktu program dijalankan fungsi-fungsi dalam stdlib otomatis ter-*import* dan bisa digunakan.

> *Lebih detailnya mengenai Python Standard Library dibahas pada chapter*

> *Python Standard Library*

Untuk tambahan latihan, buat satu fungsi lagi, lalu isi dengan banyak statement. Misalnya:

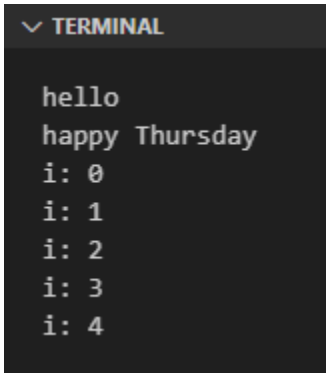
```
def print_something():
    print("hello")

    today = "Thursday"
    print(f"happy {today}")

    for i in range(5):
        print(f"i: {i}")

print_something()
```

Output program:



Penulisan isi fungsi (statement-statement dalam fungsi) harus disertai dengan *indentation* yang benar. Isi statement posisinya tidak boleh sejejar dengan blok deklarasi fungsi (secara vertikal). Isi fungsi harus lebih menjorok ke kanan.

Sebagai contoh, penulisan statement berikut adalah tidak valid dan menghasilkan error sewaktu di-run:

```
def print_something():
print("hello")

today = "Thursday"
print(f"happy {today}")

for i in range(5):
    print(f"i: {i}")
```

A.22.2. Parameter dan argument fungsi

Fungsi bisa memiliki parameter. Dengan adanya parameter, suatu nilai bisa di-sisipkan ke dalam fungsi secara dinamis saat pemanggilannya.

Parameter sendiri merupakan istilah untuk variabel yang menempel pada fungsi, yang mengharuskan kita untuk menyisipkan nilai pada parameter tersebut saat pemanggilan fungsi.

Contoh:

```
def calculate_circle_area(r):
    area = 3.14 * (r ** 2)
    print("area of circle:", area)

calculate_circle_area(788)
# output → area of circle: 1949764.1600000001
```

Penjelasan:

- Fungsi `calculate_circle_area()` dideklarasikan memiliki parameter bernama `r`.
- Notasi penulisan parameter fungsi ada diantara penulisan kurung `()` milik

blok deklarasi fungsi.

- Tugas fungsi `calculate_circle_area()` adalah menghitung luas lingkaran dengan nilai jari-jari didapat dari parameter `r`. Nilai luas lingkaran kemudian di-print.
- Setelah blok deklarasi fungsi, ada statement pemanggilan fungsi `calculate_circle_area()`. Nilai numerik `788` digunakan sebagai argument parameter `r` pemanggilan fungsi tersebut.

> **INFO**
>
> Catatan:
>
> - Parameter adalah istilah untuk variabel yang menempel di fungsi.
> - Argument adalah isitilah untuk nilai yang disisipkan saat pemanggilan fungsi (yang ditampung oleh parameter).
>
> Dewasa ini, kedua istilah tersebut dimaknai sama, jadi tidak usah bingung.

Parameter *by default* bisa menerima segala jenis tipe data. Untuk memaksa suatu parameter agar hanya bisa menerima data tertentu, maka tulis tipe data yang diinginkan dengan notasi penulisan sama seperti deklarasi variabel.

Perhatikan contoh berikut agar lebih jelas. Fungsi `calculate_circle_area()` di atas di-*refactor* menjadi fungsi dengan 2 parameter yaitu `message` bertipe string dan `r` bertipe `int`.

```
def calculate_circle_area(message: str, r: int):
    area = 3.14 * (r ** 2)
    print(message, area)

calculate_circle_area("area of circle:", 788)
```

Fungsi bisa tidak memiliki parameter, satu parameter, atau bisa lebih dari satu, tidak ada batasan.

> *Python memiliki **args** dan **kwargs**, pembahasan detailnya ada di chapter Function ➜ Args & Kwargs*

O iya, argument fungsi bisa dituliskan secara horizontal maupun vertikal. Misalnya:

- Penulisan argument secara horizontal

```
calculate_circle_area("area of circle:", 788)
```

- Penulisan argument secara vertikal

```
calculate_circle_area(
    "area of circle:",
    788
)
```

Penulisan argument secara vertikal umumnya cukup berguna pada situasi dimana fungsi yang dipanggil memiliki cukup banyak parameter yang harus diisi.

A.22.3. Naming convention fungsi & parameter

Mengacu ke dokumentasi PEP 8 – Style Guide for Python Code, nama fungsi dianjurkan untuk ditulis menggunakan `snake_case`.

```
def say_hello():
    print("hello")
```

Sedangkan aturan penulisan nama parameter/argument adalah sama seperti nama variabel, yaitu menggunakan `snake_case` juga. Misalnya:

```
def say_hello(the_message):
    print(the_message)
```

A.22.4. Nilai balik fungsi (*return value*)

Fungsi bisa memiliki *return value* atau nilai balik. Data apapun bisa dijadikan sebagai nilai balik fungsi, caranya dengan dengan memanfaatkan keyword `return`, tulis keyword tersebut di dalam isi fungsi diikuti dengan data yang ingin dikembalikan.

Mari coba praktekan, coba jalankan kode berikut:

```
def calculate_circle_area(r: int):
    area = 3.14 * (r ** 2)
    return area

def calculate_circle_circumference(r: int):
    return 2 * 3.14 * r

area = calculate_circle_area(788)
print(f"area: {area:.2f}")
# output → area: 1949764.16

circumference = calculate_circle_circumference(788)
print(f"circumference: {circumference:.2f}")
# output → circumference: 4948.64
```

Penjelasan:

- Notasi penulisan parameter fungsi ada dalam kurung `()` milik blok deklarasi fungsi.
- Fungsi `calculate_circle_area()` dideklarasikan memiliki parameter bernama `r`.
    - Tugas fungsi ini adalah menghitung luas lingkaran dengan nilai jari-jari didapat dari parameter `r`. Hasil perhitungan disimpan di variabel `area`.
    - Di akhir isi fungsi, nilai variabel `area` dikembalikan menggunakan keyword `return`.
- Fungsi `calculate_circle_circumference()` mirip seperti fungsi sebelumnya, hanya saja fungsi ini memiliki tugas yang berbeda yaitu untuk menghitung keliling lingkaran.
    - Fungsi ini melakukan perhitungan `2 * 3.14 * r` kemudian hasilnya dijadikan nilai balik.
- Setelah blok deklarasi fungsi, ada statement pemanggilan fungsi `calculate_circle_area()`. Nilai `788` digunakan sebagai argument parameter `r` pemanggilan fungsi tersebut.
- Kemudian ada lagi statement pemanggilan fungsi `calculate_circle_circumference()`. Nilai `788` digunakan sebagai argument parameter `r` pemanggilan fungsi tersebut.
- Nilai balik kedua pemanggilan fungsi di atas masing-masing di-print.

O iya, fungsi yang didalamnya tidak memiliki statement `return` sebenarnya juga mengembalikan nilai balik, yaitu `None`.

> *Pembahasan detail mengenai tipe data `None` ada di chapter None*

A.22.5. Tipe data nilai balik fungsi (*return type*)

Python mendukung *type hinting* yaitu penentuan tipe data nilai balik fungsi yang ditulis secara eksplisit. Penerapannya bisa dilihat pada kode berikut dimana fungsi `calculate_circle_area()` dan `calculate_circle_circumference()` tipe data nilai baliknya ditulis secara jelas.

```
def calculate_circle_area(r: int) -> float:
    area = 3.14 * (r ** 2)
    return area

def calculate_circle_circumference(r: int) -> float:
    return 2 * 3.14 * r

area = calculate_circle_area(788)
print(f"area: {area:.2f}")
# output → area: 1949764.16

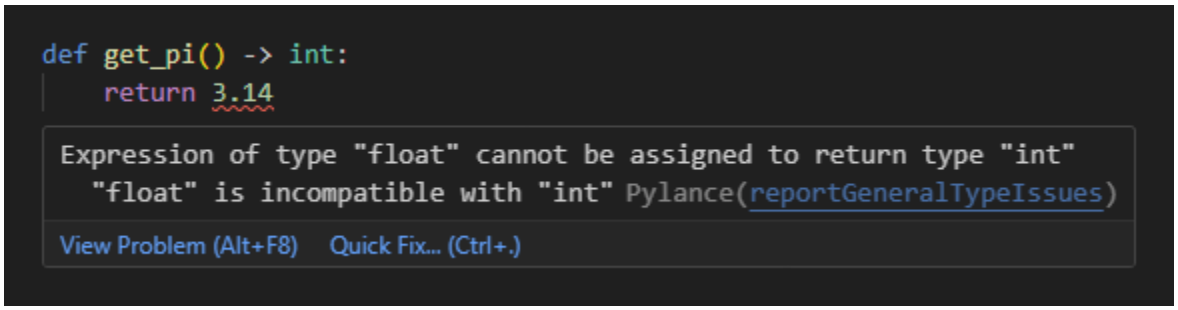
circumference = calculate_circle_circumference(788)
print(f"circumference: {circumference:.2f}")
# output → circumference: 4948.64
```

Notasi penulisan *type hinting* adalah dengan menuliskan tanda `->` setelah deklarasi fungsi diikuti dengan tipe datanya.

```
def <function_name>() -> <data_type>:
    return <data>
```

Penerapan *type hinting* mewajibkan fungsi untuk selalu memiliki nilai balik berupa data bertipe sesuai.

Jika nilai balik tipe datanya berbeda dengan yang ditulis di deklarasi fungsi, warning akan muncul.



Khusus untuk tipe data nilai balik `None` tidak wajib diikuti statement `return`. Contoh:

- Fungsi dengan return type `None` diikuti statement `return`:

```
def say_hello() -> None:
    print("hello world")
    return None
```

- Fungsi dengan return type `None` tanpa diikuti statement `return`:

```
def say_hello() -> None:
    print("hello world")
```

- Fungsi dengan tanpa return type maupun return statement:

```
def say_hello():
    print("hello world")
```

A.22.6. Keyword `pass`

Keyword `pass` secara fungsional umumnya tidak terlalu berguna, kecuali untuk beberapa situasi. Misalnya untuk dipergunakan sebagai isi pada fungsi yang masih belum selesai dikerjakan. Daripada fungsi isinya kosong dan akan menghasilkan error kalau di-run, lebih baik diisi `pass`.

Sebagai contoh, penulis berencana membuat fungsi bernama `transpose_matrix()`, namun fungsi tersebut tidak akan di-*coding* sekarang karena suatu alasan. Jadi yang penulis lakukan adalah mendeklarasikan fungsi tersebut, kemudian diisi hanya statement `pass`.

```
# need to complete sometime later
def transpose_matrix(matrix):
    pass
```

Dari blok kode di atas, nantinya engineer akan tau bahwa fungsi tersebut akan dibuat tapi belum selesai pengerjaannya.

Catatan chapter 🗒️

◉ Source code praktik

`github.com/novalagung/dasarpemrogramanpython-example/../function`

◉ Chapter relevan lainnya

- Function ➜ Positional, Optional, Keyword Arguments
- Function ➜ Args & Kwargs
- Function ➜ Closure
- Function ➜ Lambda

◉ Referensi

- https://docs.python.org/3/tutorial/controlflow.html#defining-functions
- https://peps.python.org/pep-3102/
- https://peps.python.org/pep-0484/

A.23. Python Function Argument (Positional, Optional, Keyword Argument)

Pada chapter ini kita akan belajar tentang apa itu positional argument, optional argument, dan keyword arguments, serta bagaimana penerapannya di Python.

A.23.1. Positional argument

Positional argument adalah istilah untuk urutan parameter/argument fungsi. Pengisian argument saat pemanggilan fungsi harus urut sesuai dengan deklarasi parameternya.

Silakan perhatikan kode berikut:

```
def create_sorcerer(name, age, race, era):
    return {
        "name": name,
        "age": age,
        "race": race,
        "era": era,
    }

obj1 = create_sorcerer("Sukuna", 1000, "incarnation", "heian")
print(obj1)
# output → {'name': 'Sukuna', 'age': 1000, 'race': 'incarnation', 'era':
'heian'}
```

Coba lakukan sedikit experiment dengan mengubah urutan pengisian data contohnya seperti ini. Hasilnya: program tidak error, namun data yang dihasilkan adalah tidak sesuai harapan.

```
obj4 = create_sorcerer("400 year ago", 400, "human", "Hajime Kashimo")
print(obj3)
# output → {'name': '400 year ago', 'age': 400, 'race': 'human', 'era':
'Hajime Kashimo'}
```

Saat pemanggilan fungsi dengan argument, pastikan untuk selalu menyisipkan argument sesuai dengan parameter yang dideklarasikan. Gunakan penamaan parameter yang sesuai agar lebih mudah untuk mengetahui parameter harus diisi dengan data apa.

A.23.2. Keyword argument

Keyword argument atau *named argument* adalah metode pengisian argument pemanggilan fungsi disertai nama parameter yang ditulis secara jelas (*eksplisit*).

Pada kode berikut dibuat 3 buah statement pemanggilan fungsi `create_sorcerer()`. Ketiganya memiliki perbedaan satu sama lain pada bagian bagaimana argument disisipkan ke fungsi.

```
obj5 = create_sorcerer("Sukuna", 1000, "incarnation", "heian")
print(obj5)
# output → {'name': 'Sukuna', 'age': 1000, 'race': 'incarnation', 'era':
'heian'}

obj6 = create_sorcerer(name="Kenjaku", age=1000, race="human", era="1000+
year ago")
print(obj6)
```

Penjelasan:

- Pada statement `obj5`, fungsi dipanggil dengan nilai argument disisipkan seperti biasa.
- Pada statement `obj6`, fungsi dipanggil dengan nilai argument disisipkan disertai nama parameter.
- Pada statement `obj7`, argument pertama dan ke-2 ditulis tanpa nama parameter, sedangkan argument ke-3 dan ke-4 ditulis disertai nama parameternya.

Kombinasi penulisan argument seperti pada statement `obj7` adalah diperbolehkan, dengan catatan: untuk argument yang tidak disertai nama parameter harus diletakkan di kiri sebelum penulisan argument parameter lainnya yang mengadopsi metode *keyword argument*.

Salah satu benefit dari penerapan *keyword argument*: pada argument pemanggilan fungsi yang disertai nama parameter, urutan penulisan argument boleh di-ubah. Contohnya seperti ini:

```
obj8 = create_sorcerer(era="1000+ year ago", age=1000, name="Kenjaku",
race="human")
print(obj8)
# output → {'name': 'Kenjaku', 'age': 1000, 'race': 'human', 'era':
'1000+ year ago'}

obj9 = create_sorcerer("Hajime Kashimo", 400, era="400 year ago",
race="human")
print(obj9)
# output → {'name': 'Hajime Kashimo', 'age': 400, 'race': 'human', 'era':
'400 year ago'}
```

Pada statement `obj8` semua argument pemanggilan fungsi ditulis menggunakan metode *keyword argument* dan urutannya diubah total.

Sewaktu di-print, hasilnya tetap valid. Sedangkan pada statement `obj9`, hanya argument parameter `era` dan `race` yang ditulis menggunakan metode *keyword argument* dengan urutan diubah. Sisanya (yaitu `name` dan `age`) ditulis menggunakan metode *positional argument* secara urut.

Kesimpulannya:

- Penulisan argument pemanggilan fungsi *by default* harus urut (sesuai dengan aturan *positional argument*), dengan pengecualian jika argument ditulis menggunakan *keyword argument* maka boleh diubah urutannya.
- Jika suatu pemanggilan fungsi pada bagian penulisan argument-nya menerapkan kombinasi *positional argument* dan *keyword argument* maka untuk argument yang ditulis tanpa keyword harus berada di bagian kiri dan dituliskan secara urut.

A.23.3. Optional argument

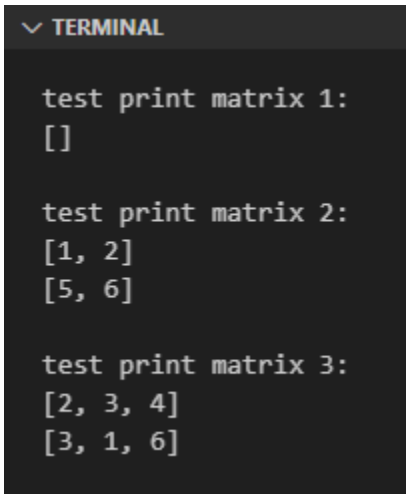
Suatu parameter bisa ditentukan nilai *default*-nya saat deklarasi fungsi. Efeknya, saat pemanggilan fungsi diperbolehkan untuk tidak mengisi nilai argument karena nilai *default* sudah ditentukan.

Sebagai contoh, pada fungsi `print_matrix()` berikut, parameter `matrix` di-set nilai *default*-nya adalah list kosong `[]`. Fungsi `print_matrix()` dipanggil 2x, pemanggilan pertama dengan tanpa argument, dan yang kedua dengan argument matrix `[[1, 2], [5, 6]]`.

```
def print_matrix(matrix=[]):
    if len(matrix) == 0:
        print("[]")

    for el in matrix:
```

Silakan run program di atas, dan perhatikan outputnya. Error tidak muncul saat eksekusi statement `print_matrix()` pertama yang padahal tidak ada data yang disisipkan saat pemanggilan fungsi. Hal ini karena fungsi tersebut pada parameter `matrix` sudah ada nilai *default*-nya.



A.23.4. Kombinasi positional argument, keyword argument, dan optional argument

Parameter fungsi bisa berisi nilai default (seperti pada contoh sebelumnya) atau tidak, atau bisa juga kombinasi keduanya.

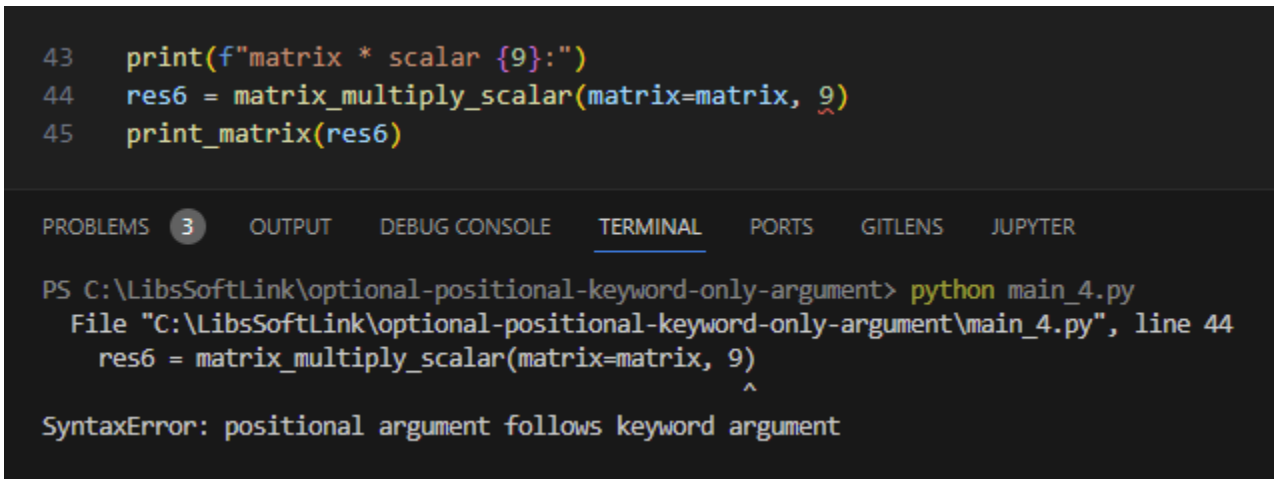
Kode program berikut adalah contoh pengaplikasiannya. Fungsi `matrix_multiply_scalar()` memiliki 2 buah parameter yaitu `matrix` yang tidak memiliki *default value* dan `scalar` yang *default value*-nya adalah `1`.

```
def matrix_multiply_scalar(matrix, scalar = 1):
    res = []
    for row in matrix:
        res.append([cell * scalar for cell in row])
```

Pada kode di atas fungsi `matrix_multiply_scalar()` dipanggil beberapa kali:

- Pemanggilan ke-1: nilai parameter `scalar` tidak diisi, efeknya maka *default value* digunakan.
- Pemanggilan ke-2: nilai parameter `scalar` ditentukan adalah `3`.
- Pemanggilan ke-3: nilai parameter `scalar` ditentukan adalah `2` menggunakan metode *keyword argument* diterapkan.
- Pemanggilan ke-4: nilai parameter `matrix` dan `scalar` dituliskan menggunakan metode *keyword argument* diterapkan.
- Pemanggilan ke-5: nilai parameter `matrix` dan `scalar` dituliskan menggunakan metode *keyword argument* diterapkan dengan posisi penulisan argument diubah.

Argument pemanggilan fungsi yang ditulis menggunakan metode *keyword argument* harus selalu diposisikan di sebelah kanan, setelah penulisan argument yang menggunakan metode *positional argument*. Jika dipaksa ditulis terbalik, maka menghasilkan error. Contohnya seperti pada gambar berikut:



Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-
example/../positional-optional-keyword-argument
```

◉ Chapter relevan lainnya

- Function
- Function ➜ Args & Kwargs
- Function ➜ Closure
- Function ➜ Lambda

◉ Referensi

- https://docs.python.org/3/tutorial/controlflow.html#defining-functions

A.24. Python Args & Kwargs

Pada chapter ini kita akan belajar tentang penerapan args dan kwargs pada fungsi di Python.

A.24.1. Pengenalan Args

Args (atau yang umumnya ditulis sebagai ***args**) merupakan notasi penulisan parameter spesial dengan kapabilitas bisa menampung banyak *positional argument* untuk ditampung dalam 1 parameter saja.

Agar lebih jelas tentang kegunaan args, mari pelajari terlebih dahulu kode berikut:

```
def sum_then_print(n1, n2, n3, n4, n5):
    total = n1 + n2 + n3 + n4 + n5
    print(total)

sum_then_print(2, 3, 4, 5, 4)
# output → 18
```

Fungsi `sum_then_print()` menerima 5 buah argument numerik yang dari nilai tersebut kemudian dihitung totalnya lalu ditampilkan.

Fungsi tersebut memiliki limitasi yaitu hanya bisa menerima 5 buah data numerik. Untuk membuatnya bisa menampung sejumlah data, solusinya bisa dengan cukup menggunakan 1 parameter saja dengan data argument yang disisipkan harus dalam tipe data sequence seperti list, atau solusi alternatif

lainnya bisa dengan menggunakan ***args** yang di bawah ini dibahas.

Implementasi args cukup mudah, pada deklarasi fungsi tulis saja parameter dengan nama apapun bebas, tetapi pada penulisannya diawali karakter asterisk atau *, contohnya seperti parameter `numbers` berikut:

```
def sum_then_print(*numbers):
    total = 0
    for n in numbers:
        total = total + n
    print(total)

sum_then_print(2, 3, 4, 5, 4)
# output → 18
```

Fungsi di atas parameter `numbers`-nya ditulis menggunakan notasi ***args**, maka parameter tersebut akan menampung semua argument yang disisipkan saat pemanggilan fungsi. Nilai argument disimpan oleh parameter `numbers` dalam bentuk **tuple**. Variabel `numbers` di-iterasi nilainya lalu dihitung totalnya.

◉ Args untuk argument dengan tipe data bervariasi

Metode ***args** ini mampu menampung segala jenis argument tanpa meghiraukan tipe datanya. Contohnya bisa dilihat pada program berikut ini:

```
def print_data(*params):
    print(f"type: {type(params)}, data: {params}")
    for i in range(len(params)):
        print(f"param {i}: {params[i]}")

print_data("hello python", 123, [5, True, ("yesn't")], {"iwak", "peyek"})
# output ↓
```

◉ Kombinasi positional argument dan args

Args sebenarnya tidak benar-benar menangkap semua argument pemanggilan fungsi, melainkan hanya argument yang ditulis sesuai posisi parameter hingga posisi setelahnya. Misalnya, sebuah fungsi memiliki 2 parameter dimana parameter pertama menampung string dan parameter kedua adalah ***args**, maka pada contoh ini parameter ***args** hanya menampung argument ke-2 dan setelahnya. Contoh:

```
def sum_then_print(message, *numbers):
    total = 0
    for n in numbers:
        total = total + n
    print(f"{message} {total}")

sum_then_print("total nilai:", 2, 3, 4, 5, 4)
# output → total nilai: 18
```

Bisa dilihat, pada kode di atas parameter `message` menampung argument ke-1 yaitu string `total nilai:`, dan parameter `numbers` menampung argument ke-2 hingga seterusnya (yaitu data `2`, `3`, `4`, `5`, `4`).

Perlu diketahui dalam penerapan kombinasi positional argument dan args, positional argument harus selalu ditulis sebelum parameter ***args**.

◉ Kombinasi positional argument, args, dan keyword argument

Keyword argument bisa digunakan bebarengan dengan positional argument dan ***args**, dengan syarat harus dituliskan di akhir setelah ***args**.

```python
def sum_then_print(message, *numbers, suffix_message):
    total = 0
    for n in numbers:
        total = total + n
    print(f"{message} {total} {suffix_message}")

sum_then_print("total nilai:", 2, 3, 4, 5, 4, suffix_message="selesai!")
# output → total nilai: 18 selesai!
```

A.24.2. Pengenalan Kwargs

Kwargs (atau yang umumnya ditulis sebagai ****kwargs** atau **keyword arguments**) merupakan notasi penulisan parameter spesial dengan kapabilitas bisa menampung banyak *keyword argument* pemanggilan fungsi dalam 1 parameter saja.

```python
def print_data(**data):
    print(f"type: {type(data)}")
    print(f"data: {data}")

    for key in data:
        print(f"param: {key}, value: {data[key]}")

print_data(phone="nokia 3310", discontinue=False, year=2000,
networks=["GSM", "TDMA"])
# output ↓
#
# type: <class 'dict'>
# data: {'phone': 'nokia 3310', 'discontinue': False, 'year': 2000,
'networks': ['GSM', 'TDMA']}
#
# param: phone, value: nokia 3310
# param: discontinue, value: False
# param: year, value: 2000
```

Argument yang ditampung oleh parameter ****kwargs** datanya tersimpan dalam bentuk dictionary dengan key adalah nama parameter dan value adalah nilai argument.

◉ Kombinasi positional argument dan kwargs

Kwargs sebenarnya hanya menampung semua argument mulai dari argument ke-`n` hingga seterusnya dimana `n` adalah nomor/posisi ****kwargs** ditulis.

Contohnya pada kode berikut, parameter `data` hanya akan menampung argument nomor ke-3 hingga seterusnya. Argument pertama ditampung oleh parameter `message` sedangkan argument ke-2 oleh parameter `number`.

```
def print_data(message, number, **data):
    print(f"message: {message}")
    print(f"number: {number}")
    print()
    for key in data:
        print(f"param: {key}, value: {data[key]}")

print_data("sesuk prei", 2023, phone="nokia 3315", networks=["GSM",
"TDMA"])
# output ↓
#
# message: sesuk prei
# number: 2023
#
# param: phone, value: nokia 3310
# param: networks, value: ['GSM', 'TDMA']
```

Dalam penerapannya, positional argument harus selalu ditulis sebelum parameter ****kwargs**.

◉ Kombinasi positional argument, args dan kwargs

Kombinasi antara positional argument, ***args**, dan ****kwargs** juga bisa dilakukan dengan ketentuan semua positional argument ditulis terlebih dahulu, kemudian diikuti ***args**, lalu ****kwargs**.

Contoh penerapannya:

```
def print_all(message, *params, **others):
    print(f"message: {message}")
    print(f"params: {params}")
    print(f"others: {others}")

print_all("hello world", 1, True, ("yesn't", "nope"), name="nokia 3310",
discontinued=True, year_released=2000)
# output ↓
#
# message: hello world
# params: (1, True, ("yesn't", 'nope'))
# others: {'name': 'nokia 3310', 'discontinued': True, 'year_released':
2000}
```

Python secara cerdas mengidentifikasi argument mana yang akan disimpan pada positional parameter, ***args**, dan ****kwargs**. Pada kode di atas, mapping antara arguments dengan parameter adalah seperti ini:

- Argument `hello world` ditampung parameter `message`.
- Argument `1`, `True`,dan `("yesn't", "nope")` ditampung parameter `params`.
- Keyword argument `name="nokia 3310"`, `discontinued=True`, dan `year_released=2000` ditampung parameter `others`.

◉ Kombinasi positional argument, args, keyword

argument, dan kwargs

Keyword argument bisa dituliskan diantara ***args**, dan ****kwargs**, diluar itu menghasilkan error.

```
def print_all(message, *params, say_something, **others):
    print(f"message: {message}")
    print(f"params: {params}")
    print(f"say_something: {say_something}")
    print(f"others: {others}")

print_all("hello world", 1, True, ("yesn't", "nope"), say_something="how are you", name="nokia 3310", discontinued=True, year_released=2000)
# output ↓
#
# message: hello world
# params: (1, True, ("yesn't", 'nope'))
# say_something: how are you
# others: {'name': 'nokia 3310', 'discontinued': True, 'year_released': 2000}
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../args-kwargs
```

◉ Chapter relevan lainnya

- Function

- Function ➜ Positional, Optional, Keyword Arguments
- Function ➜ Closure
- Function ➜ Lambda

◉ Referensi

- https://docs.python.org/3/tutorial/controlflow.html#arbitrary-argument-lists

A.25. Python Closure

Closure adalah istilah umum dalam programming untuk deklarasi fungsi yang berada di dalam fungsi (*nested function*). Pada chapter ini kita akan mempelajari cara implementasinya.

A.25.1. Pengenalan Closure

Di Python, fungsi bisa dideklarasikan di-dalam suatu fungsi. Penerapannya cukup berguna pada kasus dimana ada blok kode yang perlu di-eksekusi lebih dari satu kali tetapi eksekusinya hanya di dalam fungsi tertentu, atau eksekusinya setelah pemanggilan fungsi tertentu.

Permisalan ada fungsi `inner()` yang dideklarasikan di dalam fungsi `outer()`, maka:

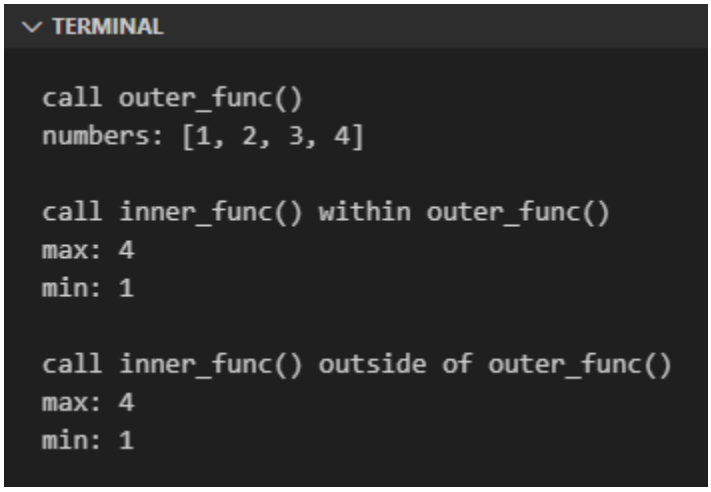
- Fungsi `inner()` bisa diakses dari dalam fungsi `outer()`
- Fungsi `inner()` juga bisa diakses dari luar fungsi `outer()` asalkan fungsi `inner()` tersebut dijadikan sebagai nilai balik fungsi `outer()` (untuk kemudian ditampung ke dalam variabel lalu dieksekusi)

Program berikut berisi contoh praktis tentang fungsi `inner()` dan `outer()`. Silakan pelajari dan praktekan.

```
def outer_func(numbers = []):
    print(f"numbers: {numbers}")

    def inner_func():
        print(f"max: {max(numbers)}")
        print(f"min: {min(numbers)}")
```

Output program:



Program di atas jika di-breakdown sesuai urutan eksekusi statement-nya kurang lebih seperti ini:

- Tahap 1: eksekusi statement `print("call outer_func()")`
- Tahap 2: eksekusi statement `print(f"numbers: {numbers}")`
- Tahap 3: eksekusi statement `print("call inner_func() within outer_func()")`
- Tahap 4: eksekusi statement `inner_func()`
  - Tahap 4.A. eksekusi statement `print(f"max: {max(numbers)}")`
  - Tahap 4.B. eksekusi statement `print(f"min: {min(numbers)}")`
- Tahap 5: eksekusi statement `print("call inner_func() outside of outer_func()")`
- Tahap 6: eksekusi statement `inner_func()` via `f()` dari luar fungsi `outer_func()`
  - Tahap 6.A. eksekusi statement `print(f"max: {max(numbers)}")`
  - Tahap 6.B. eksekusi statement `print(f"min: {min(numbers)}")`

Jika di-*flatten* semua statement-nya maka programnya menjadi seperti ini:

```
print("call outer_func()")
numbers = [1, 2, 3, 4]
print(f"numbers: {numbers}")

print("call inner_func() within outer_func()")
print(f"max: {max(numbers)}")
print(f"min: {min(numbers)}")

print("call inner_func() outside of outer_func()")
print(f"max: {max(numbers)}")
print(f"min: {min(numbers)}")
```

◉ Fungsi `min()` & `max()`

Kedua fungsi ini digunakan untuk menghitung agregasi data numerik.

- Fungsi `min()` untuk pencarian nilai minimum dari data list yang berisi elemen data numerik.
  Contoh `min([3, 4, 1, 2, 3, 4])` menghasilkan data `1`.

- Fungsi `max()` untuk pencarian nilai maksimum dari data list yang berisi elemen data numerik.
  Contoh `max([3, 4, 1, 2, 3, 4])` menghasilkan data `4`.

A.25.2. Menampung fungsi dalam variabel

Pada contoh sebelumnya, fungsi `inner_func()` ditampung ke variabel bernama `f` via nilai balik pemanggilan fungsi `outer_func()`. Dari sini terlihat bahwa closure bisa disimpan ke variabel.

Tidak hanya closure, fungsi biasa-pun juga bisa disimpan dalam variabel,

contohnya ada pada fungsi `print_all()` berikut yang disimpan pada variabel `display` untuk kemudian di-eksekusi.

```
def print_all(message, *numbers, **others):
    print(f"message: {message}")
    print(f"numbers: {numbers}")
    print(f"others: {others}")

display = print_all
display("hello world", 1, 2, 3, 4, name="nokia 3310", discontinued=True,
year_released=2000)
# output ↓
#
# message: hello world
# numbers: (1, 2, 3, 4)
# others: {'name': 'nokia 3310', 'discontinued': True, 'year_released':
2000}
```

A.25.3. Fungsi sebagai argument parameter

Selain disimpan dalam variabel, fungsi/closure bisa juga dijadikan sebagai nilai argument suatu parameter fungsi. Metode seperti ini cukup sering digunakan terutama pada operasi data sequence atau agregasi data numerik.

Contoh penerapan fungsi/closure sebagai argument pemanggilan fungsi bisa dilihat pada kode berikut ini. Silakan coba dan pelajari, penjelasannya ada dibawah kode.

```
def aggregate(message, numbers, f):
    res = f(numbers)
    print(f"{message} is {res}")
```

Fungsi `aggregate()` dideklarasikan memiliki 3 buah parameter yaitu `message`, `numbers`, dan `f` dimana `f` adalah akan diisi dengan fungsi/ closure. Di dalam fungsi `aggregate()`, fungsi `f` dipanggil dengan disisipkan argument yaitu `numbers` dalam pemanggilannya.

Ada juga fungsi `sum()` dideklarasikan dengan tugas untuk menghitung total dari data list numerik `numbers`. Dan fungsi `avg()` untuk nilai rata-rata dari data `numbers`.

Kemudian di bawahnya ada 4 buah statement pemanggilan fungsi `aggregate()`:

- Pemanggilan ke-1 adalah perhitungan nilai total `numbers`. Fungsi `sum` yang telah dideklarasikan sebelumnya dijadikan sebagai argument pemanggilan fungsi `aggregate()` untuk ditampung di parameter `f`.
- Pemanggilan ke-2 adalah perhitungan nilai rata-rata dimana fungsi `avg` yang telah dideklarasikan dijadikan sebagai argument pemanggilan fungsi `aggregate()`..
- Pemanggilan ke-3 adalah perhitungan nilai maksimum. Fungsi `max` yang merupakan fungsi bawaan Python digunakan sebagai argument pemanggilan fungsi `aggregate()`.
- Pemanggilan ke-1 adalah perhitungan nilai minimum. Fungsi `min` yang merupakan fungsi bawaan Python digunakan sebagai argument pemanggilan fungsi `aggregate()`.

Dari contoh terlihat bagaimana contoh penerapan closure sebagai nilai argument parameter fungsi. Fungsi atau closure bisa digunakan sebagai nilai argument, dengan catatan skema parameter-nya harus disesuaikan dengan kebutuhan.

Di dalam fungsi `aggregate()`, closure `f` diharapkan untuk memiliki

parameter yang bisa menampung data list `numbers`. Selama fungsi/closure memenuhi kriteria ini maka penggunaannya tidak menghasilkan error.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../args-kwargs
```

◉ Chapter relevan lainnya

- Function
- Function ➜ Positional, Optional, Keyword Arguments
- Function ➜ Args & Kwargs
- Function ➜ Lambda

◉ Referensi

- https://docs.python.org/3/library/stdtypes.html#functions

A.26. Python Lambda

Pada chapter ini kita akan belajar tentang *anonymous function* atau fungsi tanpa nama yang biasa disebut dengan **lambda**.

A.26.1. Pengenalan Lambda

Lambda adalah fungsi yang tidak memiliki nama. Lambda umumnya disimpan ke suatu variabel atau dieksekusi langsung. Lambda bisa memiliki parameter dan mengembalikan nilai balik, seperti fungsi pada umumnya.

Perbedaan signifikan antara lambda dengan fungsi/closure adalah pada lambda isinya hanya boleh 1 baris satement. Jika ada lebih dari 1 baris silakan gunakan fungsi saja.

Untuk mempermudah pemahaman kita tentang lambda, silakan pelajari kode berikut. Ada dua blok fungsi dibuat, satu berbentuk fungsi biasa dan satunya lagi adalah lambda. Keduanya memiliki tugas sama persis yaitu menampilkan pesan `hello python`.

```
def say_hello1():
    print("hello python")

say_hello1()
# output → hello python

say_hello2 = lambda : print("hello python")

say_hello2()
# output → hello python
```

Bisa dilihat bagaimana perbedaan penulisan syntax fungsi menggunakan

lambda dibandingkan dengan fungsi biasa. Lambda ditulis menggunakan keyword `lambda`, diikuti tanda titik 2 `:` lalu statement satu baris. Lambda perlu ditampung ke sebuah variabel (misalnya `say_hello2()`), setelahnya variabel tersebut digunakan untuk mengeksekusi lambda dengan cara memanggilnya seperti fungsi.

A.26.2. lambda return value

Lambda selalu mengembalikan nilai balik atau *return value*. Jika isi lambda adalah suatu data atau operasi yang menghasilkan data, maka data tersebut otomatis jadi nilai balik. Contoh:

```
def get_hello_message1():
    return "hello python"

res = get_hello_message1()
print(res)
# output → hello python

get_hello_message2 = lambda : "hello python"

res = get_hello_message2()
print(res)
# output → hello python
```

Pada kode di atas lambda `get_hello_message2()` mengembalikan nilai balik bertipe string.

Lalu bagaimana dengan lambda `say_hello2()` yang telah dipraktekan di atas, apakah juga mengembalikan nilai balik? Iya, lambda tersebut juga ada return value-nya. Namun, karena isi lambda `say_hello2()` adalah pemanggilan fungsi `print()` maka nilai balik lambda adalah data `None`.

> *Pembahasan detail mengenai tipe data `None` ada di chapter None*

A.26.3. Lambda argument/parameter

Sama seperti fungsi, lambda bisa memiliki parameter baik itu jenisnya parameter positional, optional, ataupun keyword argument.

Sebagai contoh, lihat perbandingan fungsi `get_full_name1()` dengan lambda `get_full_name2()` pada kode berikut. Parameter di lambda dituliskan diantara keyword `lambda` dan tanda titik 2 `:`. Jika ada lebih dari 1 parameter, gunakan tanda koma `,` sebagai separator.

```
def get_full_name1(first_name, last_name):
    return f"{first_name} {last_name}"

get_full_name2 = lambda first_name, last_name : f"{first_name}
{last_name}"

res = get_full_name1("Darion", "Mograine")
print(res)
# output → Darion Mograine

res = get_full_name2("Sally", "Whitemane")
print(res)
# output → Sally Whitemane
```

Untuk penerapan optional argument dan keyword argument, contohnya bisa dilihat pada kode berikut:

```
get_full_name3 = lambda first_name, last_name = "" : f"{first_name}
{last_name}".strip()
```

A.26.4. Lambda dengan parameter *args & **kwargs

Penerapan ***args** dan ****kwargs** pada parameter lambda tidak berbeda dengan penerapannya di fungsi biasa. Sebagai perbandingan Silakan pelajari 2 contoh berikut yang masing-masing berisi contoh penulisan lambda vs versi fungsi biasa.

- Contoh lambda dengan parameter ***args**:

```
# %% A.26.4. Lambda *args dan **kwargs

def debug1(separator, *params):
    res = []
    for i in range(len(params)):
        res.append(f"param {i}: {params[i]}")
    return separator.join(res)

debug2 = lambda separator, *params : separator.join([f"param {i}:
{params[i]}" for i in range(len(params))])

res = debug1(", ", "Darion Mograine", ["Highlord", "Horseman of the
Ebon Blade", "Ashbringer"], True)
print(res)
# output → param 0: Darion Mograine, param 1: ['Highlord', 'Horseman
of the Ebon Blade', 'Ashbringer'], param 2: True

res = debug2(", ", "Darion Mograine", ["Highlord", "Horseman of the
Ebon Blade", "Ashbringer"], True)
print(res)
# output → param 0: Darion Mograine, param 1: ['Highlord', 'Horseman
of the Ebon Blade', 'Ashbringer'], param 2: True
```

- Contoh lambda dengan parameter ****kwargs**:

```python
def debug3(separator, **params):
    res = []
    for key in params:
        res.append(f"{key}: {params[key]}")
    return separator.join(res)

debug4 = lambda separator, **params : separator.join([f"{key}:
{params[key]}" for key in params])

res = debug3(
    ", ",
    name="Darion Mograine",
    occupations=["Highlord", "Horseman of the Ebon Blade",
"Ashbringer"],
    active=True
)
print(res)
# output → name: Darion Mograine, occupations: ['Highlord', 'Horseman
of the Ebon Blade', 'Ashbringer'], active: True

res = debug4(
    ", ",
    name="Darion Mograine",
    occupations=["Highlord", "Horseman of the Ebon Blade",
"Ashbringer"],
    active=True
)
print(res)
# output → name: Darion Mograine, occupations: ['Highlord', 'Horseman
of the Ebon Blade', 'Ashbringer'], active: True
```

A.26.5. Isi lambda: statement 1 baris

Lambda secara penulisan bisa dibilang lebih praktis dibanding fungsi, namun limitasinya yang hanya bisa berisi statement 1 baris saja terkadang menjadi masalah, terutama untuk mengakomodir operasi komplex yang umumnya

membutuhkan lebih dari 1 baris kode.

Namun Python dari segi bahasa adalah cukup flexibel, banyak API yang memungkinkan kita selaku programmer untuk bisa menuliskan kode yang cukup kompleks tapi dalam 1 baris saja. Pada contoh berikut, operasi transpose matrix bisa dilakukan hanya dalam 1 baris dengan menerapkan list comprehension.

```
def transpose_matrix1(m):
    tm = []
    for i in range(len(m[0])):
        tr = []
        for row in m:
            tr.append(row[i])
        tm.append(tr)
    return tm

transpose_matrix2 = lambda m : [[row[i] for row in matrix] for i in
range(len(m[0]))]

matrix = [[1, 2], [3, 4], [5, 6]]

res = transpose_matrix1(matrix)
print(res)
# output → [[1, 3, 5], [2, 4, 6]]

res = transpose_matrix2(matrix)
print(res)
# output → [[1, 3, 5], [2, 4, 6]]
```

> *Pembahasan detail mengenai list comprehension ada di chapter List Comprehension*

A.26.6. Lambda dengan parameter

fungsi/lambda

Lambda, closure, fungsi, ketiganya bisa digunakan sebagai argument suatu pemanggilan fungsi dengan cara implementasi juga sama, yaitu cukup tulis saja lambda sebagai argument baik secara langsung maupun lewat variabel terlebih dahulu.

Contoh penerapannya bisa dilihat pada kode berikut. Lambda `aggregate()` dibuat dengan desain parameter ke-3 yaitu `f` bisa menampung nilai berupa fungsi/closure/lambda.

```
aggregate = lambda message, numbers, f: print(f"{message} is
{f(numbers)}")

numbers = [24, 67, 22, 98, 3, 50]

def average1(numbers):
    return sum(numbers) / len(numbers)

aggregate("average", numbers, average1)
# output → average is 44.0

average2 = lambda numbers : sum(numbers) / len(numbers)
aggregate("average", numbers, average2)
# output → average is 44.0

aggregate("average", numbers, lambda numbers : sum(numbers) /
len(numbers))
# output → average is 44.0
```

Lambda `aggregate()` dipanggil 3x yang pada pemanggilan ke-2 dan ke-3-nya, argument parameter ke-3 diisi dengan lambda.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../lambda
```

◉ Chapter relevan lainnya

- Function
- Function ➜ Positional, Optional, Keyword Arguments
- Function ➜ Args & Kwargs
- Function ➜ Closure

◉ Referensi

- https://docs.python.org/3/reference/expressions.html#lambda

A.27. Python Modules

Pada chapter ini, kita akan menjelajahi konsep module beserta implementasinya di Python.

A.27.1. Pengenalan Modules

Module di Python merupakan istilah untuk file yang berisi kode-kode python (seperti deklarasi variabel, fungsi, class, dan lainnya). Kode-kode tersebut diisolasi sesuai dengan tugasnya. Contoh:

- Module `numbers` berisi fungsi-fungsi untuk keperluan operasi angka
- Module `random` yang isinya kode untuk generate data random

Dengan adanya module kode menjadi lebih modular, rapi, dan mudah dikelola.

Pembuatan module di Python sangat mudah karena dilakukan dengan cukup membuat file. Nama file yang digunakan adalah menjadi nama module. Misalnya, file `calculate.py`, maka nama module adalah `calculate`.

Module dapat di-import di module lain, atau digunakan sebagai entry point eksekusi program (sebutannya main module). Misalnya di praktek-praktek pada chapter sebelumnya kita sering menggunakan command `python.exe main.py` untuk menjalankan program Python. Command tersebut menjadikan module `main` (file `main.py`) sebagai entrypoint eksekusi program.

Ok, sekarang mari kita coba praktekan penerapan module sebagai dependency (module yang di-import di module lain).

Buat program baru dengan isi kode di bawah ini. File `my_program.py` kita fungsikan sebagai entrypoint program, sedangkan module `calculate` sebagai

dependency yang di-import di `my_program`.

Project structure

```
belajar-module/
├── calculate.py
└── my_program.py
```

Selanjutnya tulis isi kode file `calculate`:

File calculate.py

```
note = "module calculate contains mathematic functions"

def calc_hypotenuse(a, b):
    return sqrt(pow(a) + pow(b))

def pow(n, p = 2):
    return n ** p

def sqrt(x):
    n = 1
    for _ in range(10):
        n = (n + x/n) * 0.5
    return n
```

Module `calculate` berisi 1 buah variabel dan 3 buah fungsi:

- Variabel `note` berisi string
- Fungsi `calc_hypotenuse()` untuk menghitung nilai hipotenusa dari `a` dan `b`
- Fungsi `pow()` untuk meperingkas operasi pangkat
- Fungsi `sqrt()` untuk mencari akar kuadrat

Kesemua unit di atas di-import ke `my_program` untuk kemudian digunakan dalam perhitungan pencarian nilai hipotenusa.

File my_program.py

```
a = 10
b = 15

import calculate

print(calculate.note)

res = calculate.calc_hypotenuse(a, b)
print("hypotenuse:", res)

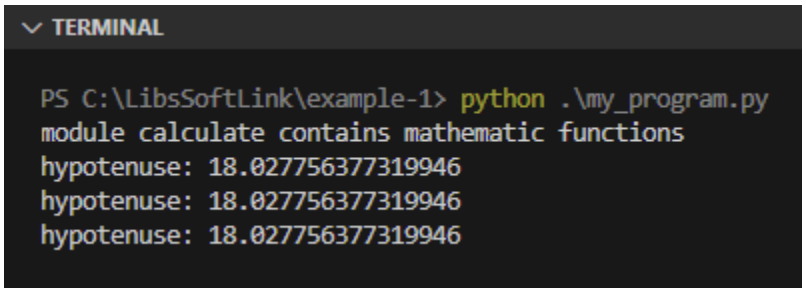
res = calculate.sqrt(a**2 + b**2)
print("hypotenuse:", res)

res = calculate.sqrt(calculate.pow(a) + calculate.pow(b))
print("hypotenuse:", res)
```

Coba jalankan program menggunakan command berikut agar module `my_program` menjadi entrypoint eksekusi program.

```
python my_program.py
```

Output:



Implementasi module di Python cukup mudah bukan?

Keyword `import` digunakan untuk meng-import suatu module atau class. Pada contoh di atas module `calculate` di-import ke `my_program.py` untuk kemudian digunakan fungsi-fungsi didalamnya.

Pengaksesan variabel/konstanta dari suatu module menggunakan notasi `<module>.<variable/constant>`, contohnya `calculate.note`.

Sedangkan pengaksesan fungsi menggunakan notasi `<module>.<function>()`, contohnya seperti `calculate.calc_hypotenuse()`, `calculate.sqrt()`, dan `calculate.pow()`.

A.27.2. Behaviour import module

Ketika suatu module di-import, semua unit di dalamnya dapat diakses dari file peng-import. Contohnya bisa dilihat pada kode yang sudah ditulis, variabel `note` dan fungsi `calc_hypotenuse()` yang berada di module `calculate`, keduanya bisa langsung digunakan.

Jika dalam module ada statement yang sifatnya bukan deklarasi variabel atau fungsi, misalnya seperti statement print, maka statement tersebut akan langsung dieksekusi saat module ter-import.

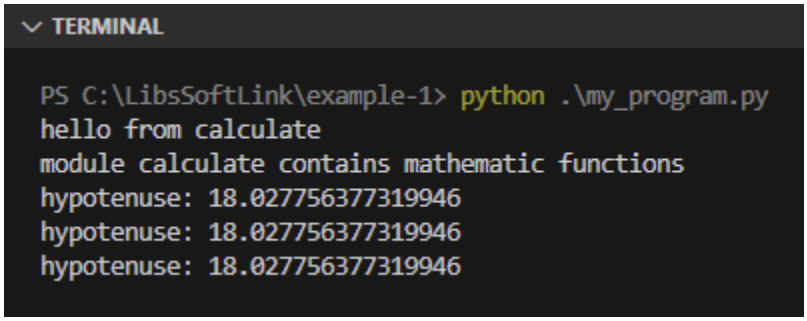
Mari coba praktekan. Tambahkan statement berikut di file `calculate.py`:

File calculate.py

```
print("hello from calculate")

note = "module calculate contains mathematic functions"
```

Jalankan program, lihat hasilnya:



A.27.3. *Naming convention* module

Mengacu ke dokumentasi PEP 8 – Style Guide for Python Code, nama module dianjurkan untuk ditulis dalam huruf kecil (lowercase) dengan underscode sebagai pembatas antar kata.

A.27.4. Keyword `from` dan `import`

Ada dua bentuk penerapan keyword `import`. Pertama, yaitu untuk meng-import module, contohya seperti pada praktek di atas (`import calculate`). Kedua, adalah mengkombinasikan keyword tersebut dengan keyword `from` untuk meng-import langsung fungsi atau item lainnya dari suatu module. Contoh:

File my_program.py

```
a = 10
b = 15

from calculate import note
from calculate import calc_hypotenuse
from calculate import sqrt
```

Dua versi berbeda `my_program.py` di atas adalah ekuivalen, keduanya melakukan operasi yang sama persis dan menghasilkan output yang sama pula.

Pada contoh ke-2 (program yang baru saja ditulis), variabel `note`, fungsi `calc_hypotenuse()` dan `sqrt()` di-import secara langsung via statement `from calculate import <function>`.

Untuk penulisannya bisa dituliskan satu per satu statement import-nya, atau bisa cukup sebaris saja (cara ini hanya berlaku untuk import item yang bersumber dari module yang sama).

```
from calculate import note
from calculate import calc_hypotenuse
from calculate import sqrt

# vs

from calculate import note, calc_hypotenuse, sqrt
```

◉ Fungsi `pow()`

Fungsi `pow()` merupakan fungsi bawaan Python Standard Library yang bisa langsung digunakan tanpa perlu meng-import apapun.

Cara penggunaan fungsi `pow()` adalah dengan langsung menulisnya dalam skema `pow(a, b)`. Fungsi ini menghasilkan operasi matematika `a pangkat b`.

Pada kode di atas, fungsi `pow()` milik module `calculate` sengaja tidak di-import agar tidak meng-override atau menimpa fungsi `pow()` bawaan Python.

A.27.5. Statement `from <module> import *`

Statement `from <module> import *` digunakan untuk meng-import semua unit yang ada dalam module `<module>`. Contoh penerapannya:

File my_program.py

```
a = 10
b = 15

from calculate import *

print(note)

res = calc_hypotenuse(a, b)
print("hypotenuse:", res)

res = sqrt(a**2 + b**2)
print("hypotenuse:", res)

res = sqrt(pow(a, 2) + pow(b, 2))
print("hypotenuse:", res)
```

A.27.6. Keyword `as`

Module maupun fungsi bisa di-import dengan diberi nama alias. Biasanya teknik ini digunakan pada situasi dimana module yang di-import namanya cukup panjang, maka digunakan alias agar lebih pendek. Pembuatan alias sendiri dilakukan via keyword `as`.

Penerapannya bisa dilihat pada contoh berikut:

File my_program.py

```
a = 10
b = 15

import calculate as calc
from calculate import calc_hypotenuse as hptns, sqrt

print(calc.note)

res = hptns(a, b)
print("hypotenuse:", res)

res = sqrt(a**2 + b**2)
print("hypotenuse:", res)

res = sqrt(calc.pow(a) + calc.pow(b))
print("hypotenuse:", res)
```

Penjelasan:

- Statement `import calculate as calc` meng-import module `calculate` dengan alias `calc`. Nantinya fungsi-fungsi dalam module tersebut bisa diakses via `calc.<function>()`.
- Statement `from calculate import calc_hypotenuse as hptns, sqrt` meng-import:
  - Fungsi `calc_hypotenuse()` dari module `calculate` dengan alias `hptns()`.
  - Fungsi `sqrt()` dari module `calculate`.

A.27.7. Urutan pencarian module

Ketika suatu module di-import, Python akan melakukan pencarian file module di beberapa tempat berikut secara berurutan:

1. Pertama, Python akan mencari module di folder yang sama dimana statement `import` digunakan.
2. Jika pencarian pertama tidak menemukan hasil, maka Python lanjut mencari file module ke folder dimana environment variable `PYTHONPATH` di-set.
3. Jika pencarian kedua juga tidak menemukan hasil, Python melanjutkan pencarian di folder dimana Python di-install.
4. Jika pencarian ketiga juga tidak sukses (file module tidak ditemukan), maka eksekusi program menghasilkan error.

A.27.8. File module dalam folder

Bagaimana jika suatu file module (misalnya `calculate.py`) berada di dalam suatu sub-folder dalam folder program, apakah cara import-nya sama? Sebenarnya sama namun ada sedikit perbedaan yang harus diketahui. Selengkapnya akan kita bahas pada chapter selanjutnya, yaitu Packages.

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../modules
```

◉ Chapter relevan lainnya

- Packages
- Special Names

◉ Referensi

- https://docs.python.org/3/tutorial/modules.html

A.28. Python Packages

Pada chapter ini, kita akan membahas cara Python mengelola module melalui packages. Dengan package, module bisa diakses menggunakan notasi pengaksesan module, contohnya seperti `calculate.calc_hypotenuse()`.

A.28.1. Pengenalan Packages

Mari kita mulai dengan sedikit mengulang pembahasan yang ada di chapter sebelumnya. Di Python, module direpresentasikan oleh file, dan agar bisa mengakses item yang ada dalam module tersebut kita perlu meng-import-nya terlebih dahulu.

Package adalah cara mengelola module dengan menempatkannya dalam suatu folder. Sederhananya: **module adalah file, dan package adalah folder**.

Untuk mendemonstrasikan konsep ini, mari kita praktekan. Buat project baru dengan struktur seperti berikut:

Project structure

```
belajar-package/
├── libs/
│   ├── calculate.py
│   └── common/
│       ├── message.py
│       └── number.py
└── my_program.py
```

Program yang dibuat masih sama seperti yang ada di praktek sebelumnya.

Perbedaannya, kali ini fungsi `sqrt()` dan `pow()` ditempatkan dalam path `libs/common/number.py`, dan ada juga satu fungsi baru dibuat di `libs/common/message.py`.

File: libs/common/number.py

```
note = "module libs.common.number contains numerical functions"

def pow(n, p = 2):
    return n ** p

def sqrt(x):
    n = 1
    for _ in range(10):
        n = (n + x/n) * 0.5
    return n
```

File: libs/common/message.py

```
note = "module libs.common.message contains messaging/printing functions"

def print_hypotenuse(v):
    print("hypotenuse:", v)
```

Selanjutnya, isi file `calculate.py` dengan deklarasi variabel `note` dan fungsi `calc_hypotenuse()`.

File: libs/calculate.py

```
import libs.common.number

note = "module libs.calculate contains mathematic functions"
```

Terakhir, di file `my_program.py` (file entrypoint eksekusi program), import module `libs/common/number.py`, `libs/common/message.py`, dan `libs/calculate.py` lalu panggil fungsi yang ada di masing-masing module. Contoh:

File: my_program.py

```
a = 10
b = 15

import libs.calculate
import libs.common.number
import libs.common.message

print(libs.calculate.note)
print(libs.common.number.note)
print(libs.common.message.note)

res = libs.calculate.calc_hypotenuse(a, b)
libs.common.message.print_hypotenuse(res)

res = libs.common.number.sqrt(a**2 + b**2)
libs.common.message.print_hypotenuse(res)

res = libs.common.number.sqrt(libs.common.number.pow(a) +
libs.common.number.pow(b))
libs.common.message.print_hypotenuse(res)
```

Bisa dilihat bagaimana peran package dalam operasi import module dan pengaksesan unit dalam module.

- Module yang berada dalam package di import menggunakan notasi `<package>.<module>`. Contoh:
    - `import libs.calculate` di file `my_program.py`
- Jika ada package di dalam package, maka ditulis semua subfoldernya,

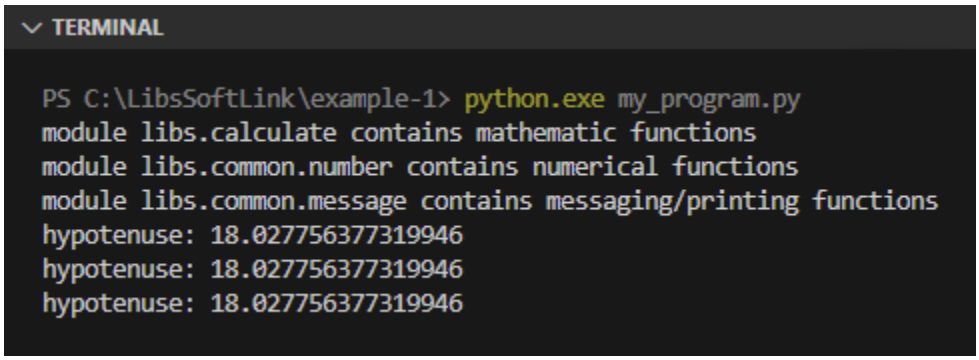
seperti: `<package>.<package>.<module>`. Contoh:

  - `import libs.common.number` di file `libs/calculate.py`
  - `import libs.common.number` di file `my_program.py`
  - `import libs.common.message` di file `my_program.py`

- Pengaksesan unit dalam module diwajibkan dengan ditulis *namespace* nya secara utuh, jadi nama package juga ditulis. Contohnya bisa dilihat pada beberapa statement seperti:

  - `libs.common.number.sqrt()` di file `libs/calculate.py`
  - `libs.common.number.pow()` di file `libs/calculate.py`
  - `libs.calculate.calc_hypotenuse()` di file `my_program.py`
  - `libs.common.message.print_hypotenuse()` di file `my_program.py`

Ok, selanjutnya coba jalankan program dan lihat hasilnya:



A.28.2. *Naming convention* package

Berdasarkan dokumentasi PEP 8 – Style Guide for Python Code, disarankan untuk menulis nama package dengan huruf kecil (lowercase) dan dianjurkan untuk menghindari penggunaan underscore.

A.28.3. Metode import module package

Seperti halnya module biasa, module dalam package bisa di-import dengan beberapa cara:

◉ Alias module via keyword `as`

Alias cukup berguna untuk mempersingkat penulisan module saat memanggil item didalamnya. Keyword `as` digunakan untuk pemberian nama alias module.

File: libs/calculate.py

```
import libs.common.number as num

note = "module libs.calculate contains mathematic functions"

def calc_hypotenuse(a, b):
    return num.sqrt(num.pow(a) + num.pow(b))
```

File: my_program.py

```
a = 10
b = 15

import libs.calculate as calc
import libs.common.number as num
import libs.common.message as msg

print(calc.note)
print(num.note)
print(msg.note)
```

Dengan menggunakan alias, namespace tidak perlu lagi dituliskan secara penuh, contohnya `libs.common.number.note` cukup ditulis dengan `num.note`.

```
import libs.common.number
print(libs.common.number.note)

# vs

import libs.common.number as num
print(num.note)
```

◉ Import package via `from` & `import`

Kombinasi keyword `from` dan `import` dapat digunakan dengan ketentuan: setelah keyword `from` yang ditulis adalah namespace package, lalu diikuti oleh keyword `import` dan nama module. Contoh:

File: libs/calculate.py

```
from libs.common import number

note = "module libs.calculate contains mathematic functions"

def calc_hypotenuse(a, b):
    return number.sqrt(number.pow(a) + number.pow(b))
```

File: my_program.py

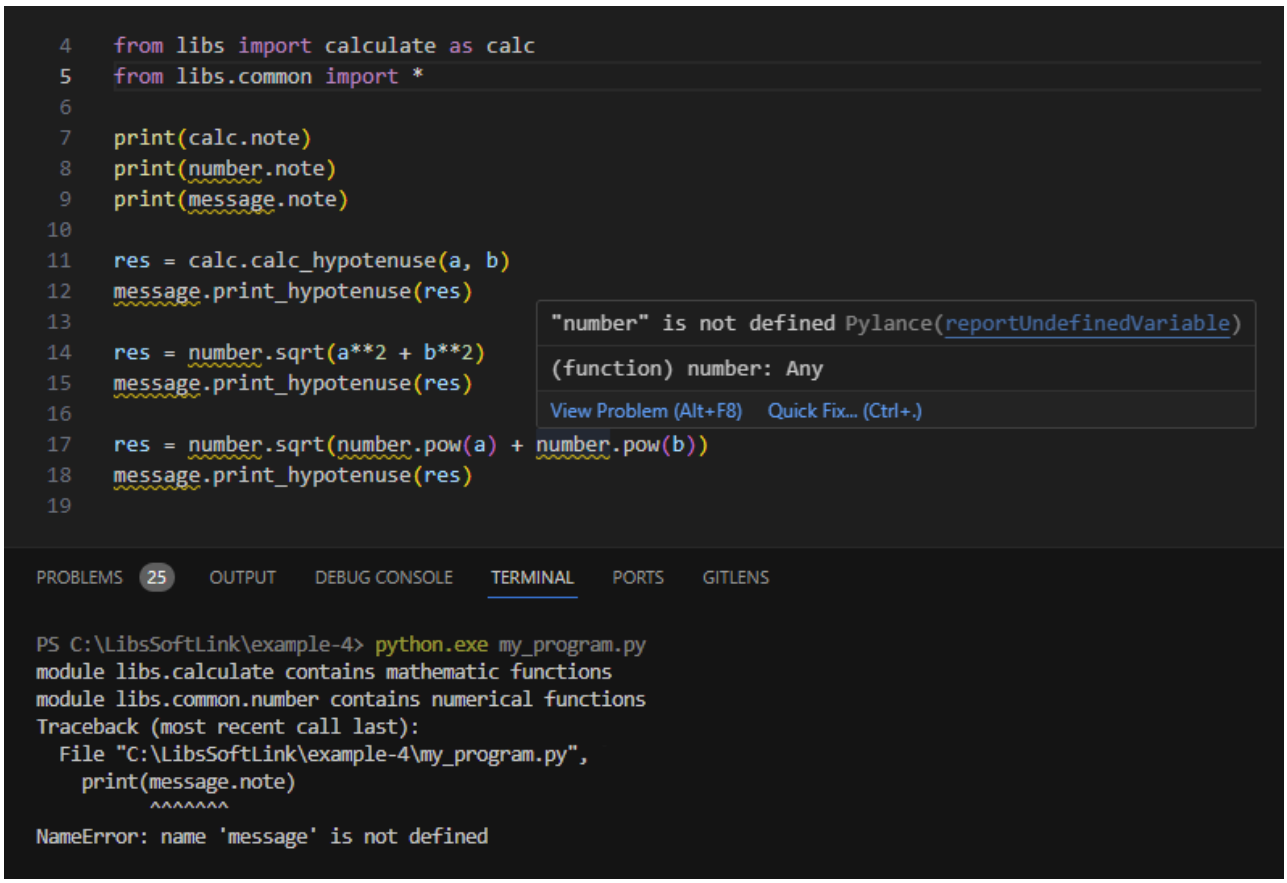
```
a = 10
b = 15
```

◉ Penggunaan `import *`

Ada beberapa hal yang perlu diketahui dalam penggunaan `import *`, namun sebelum membahasnya, silakan coba terlebih dahulu kode berikut. Silakan buka `my_program.py` lalu ubah statement import menjadi seperti ini:

```
from libs import calculate as calc
from libs.common import *
```

Hasilnya adalah error:



A.28.4. File `__init__.py`

Melanjutkan pembahasan sebelumnya dimana `import *` menghasilkan error, hal tersebut terjadi karena Python tidak bisa mendeteksi module apa saja yang bisa direpresentasikan dengan `*` saat meng-import suatu package.

Untuk mengatasi error, tambahkan file bernama `___init___.py` di setiap package/folder. Dengan ini maka struktur file program menjadi seperti ini:

Project structure

```
belajar-package/
├── libs/
│   ├── __init__.py
│   ├── calculate.py
│   └── common/
│       ├── __init__.py
│       ├── message.py
│       └── number.py
└── my_program.py
```

Isi file `__init__.py` dengan sebuah statement deklarasi variabel bernama `__all__`, dengan nilai adalah list nama module yang ada di dalam folder tersebut.

File: libs/common/__init__.py

```
__all__ = ["message", "number"]
```

File: libs/__init__.py

```
__all__ = ["calculate"]
```

Setelah penambahan di atas dilakukan, maka module yang berada dalam package bisa di-import menggunakan `import *`.

Coba sekarang test dengan mengaplikasikannya di program:

File: my_program.py

```
a = 10
b = 15

from libs import *
from libs.common import *

print(calculate.note)
print(number.note)
print(message.note)

res = calculate.calc_hypotenuse(a, b)
message.print_hypotenuse(res)

res = number.sqrt(a**2 + b**2)
message.print_hypotenuse(res)

res = number.sqrt(number.pow(a) + number.pow(b))
message.print_hypotenuse(res)
```

◉ Best practice file `__init__.py`

Sesuai penjelasan di dokumentasi Package, dianjurkan untuk selalu membuat file `__init__.py` di setiap package/folder untuk menghindari masalah saat pencarian module.

Meskipun module dalam package tidak digunakan via statement `import *`, dianjurkan untuk tetap membuat file tersebut. Isinya biarkan kosong saja tidak apa-apa.

◉ Special name `__all__`

Variabel yang diawali dan diakhir dengan karakter double underscore seperti `__all__` disebut sebagai variabel **special name**. Pembahasan lebih lanjut tentang special names ada di chapter Special Names.

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../packages
```

◉ Chapter relevan lainnya

- Modules
- Special names

◉ Referensi

- https://docs.python.org/3/tutorial/modules.html#packages

A.29. Python Variables Scope (Local vs. Global)

Pada chapter ini kita akan membahas tentang variable scope, yaitu kapan suatu variabel valid dan bisa digunakan di dalam block serta beberapa fungsi yang masih relevan dengan topik variabel scope.

A.29.1. Local vs. global variable

Global variable adalah variabel yang dideklarasikan di root (tidak di dalam suatu block fungsi). Sedangkan local variable adalah yang dideklarasikan di dalam suatu block dan hanya valid di block tersebut saja.

Contoh, sebuah file program Python bernama `main.py` di isi dengan kode berikut:

File main.py

```
name = "Cleaver Bmurglbrm"
print("greetings", name)

def greet():
    today = "Saturday"
    print("happy", today, name)

if name:
    greet()
    greet_message = "have a good day"
    print("hello", name, greet_message)

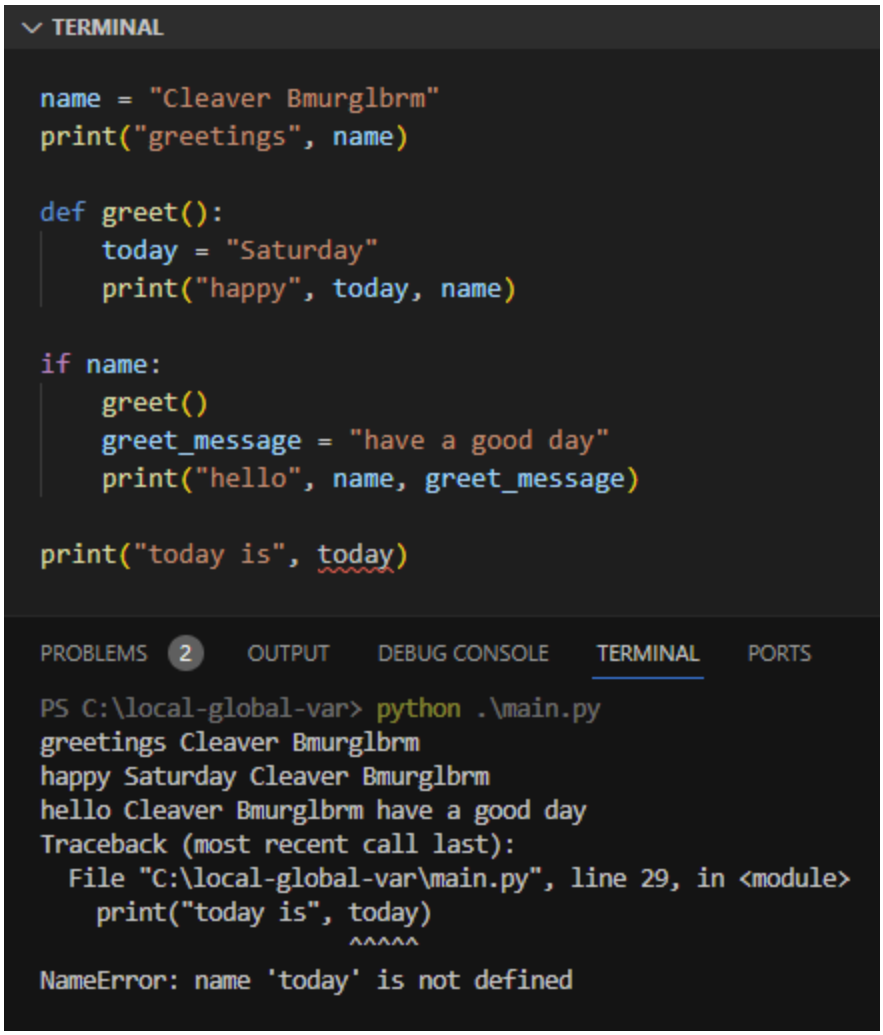
# output ↓
```

Variabel `name` disitu merupakan variabel global. Variabel jenis ini bisa diakses dari block manapun. Di contoh, variabel `name` diakses di dalam block fungsi `greet()` dan di block seleksi kondisi `if name:`.

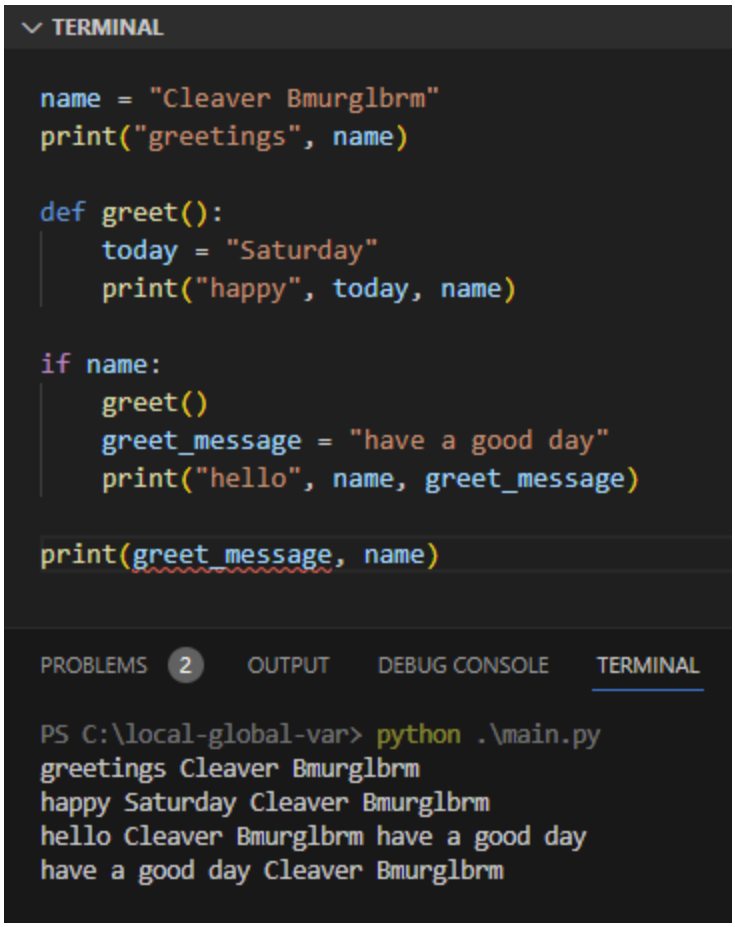
Berbeda dengan variabel `today` yang tempat deklarasinya ada di block fungsi `greet()`, membuatnya hanya valid untuk digunakan di block itu saja. Variabel `today` dikategorikan sebagai variabel local dengan scope adalah block fungsi `greet()`.

Selain itu ada juga variabel lain bernama `greet_message` yang dideklarasikan di block seleksi kondisi `if name:`.

Sekarang coba akses variabel `today` dari luar block, hasilnya pasti error. Keterangan errornya: variabel `today` tidak dikenali.



Bagaimana jika variabel `greet_message` diakses dari luar block seleksi kondisi, apakah juga error? Jawabannya, tidak!



Di editor terlihat variabel `greet_message` digaris-bawahi merah, menandakan ada error namun hanya di level linter saja. Error tersebut tidak membuat eksekusi program menjadi gagal.

Sampai sini bisa ditarik kesimpulan bahwa variabel global bisa diakses dari mana saja, sedangkan variabel local bisa diakses di dalam block dimana ia dideklarasikan. Pengaksesan variabel local diluar block-nya memiliki efek samping berikut:

- Jika variabel local dideklarasikan di dalam block fungsi/lambda/closure, maka pengaksesannya dari luar block menghasilkan error.
- Jika variabel local dideklarasikan di block seleksi kondisi atau lainnya, maka pengaksesaannya dari luar block diperbolehkan dan tidak membuat

eksekusi program menjadi error. Namun warning atau error di level linter muncul.

A.29.2. Local dan global variable dengan nama sama

Katakanlah ada variabel global dan variabel local yang namanya sama persis, di situasi seperti ini maka nilai variabel local pada block-nya adalah sesuai dengan saat deklarasinya dalam block. Di luar block, variabel tersebut nilainya kembali berisi nilai variabel global.

Agar lebih jelas coba praktekan kode berikut:

```
name = "Cleaver Bmurglbrm"

def greet():
    name = "Keymaster Urmgrgl"
    print("hello", name)

greet()
print("greetings", name)

# output ↓
#
# hello Keymaster Urmgrgl
# greetings Cleaver Bmurglbrm
```

Bisa dilihat dari output bahwa variabel `name` nilainya adalah `Keymaster Urmgrgl`, sesuai dengan isi variabel saat deklarasi di block fungsi `greet()`. Namun di luar block fungsi, nilainya kembali menjadi `Cleaver Bmurglbrm`.

Dari sini bisa diambil kesimpulan bahwa di luar block, perubahan nilai variabel local tidak berefek ke variabel global, meskipun namanya sama persis.

A.29.3. Keyword `global`

Untuk mengubah nilai suatu variabel global dari block, maka perlu adanya penggunaan keyword `global`. Keyword ini menandai variabel dalam block bahwa reference yang digunakan adalah variabel global. Efeknya, perubahan nilai pada variabel juga berpengaruh ke variabel global.

Contoh penerapan:

```
name = "Cleaver Bmurglbrm"

def greet():
    global name
    name = "Keymaster Urmgrgl"
    print("hello", name)

greet()
print("greetings", name)

# output ↓
#
# hello Keymaster Urmgrgl
# greetings Keymaster Urmgrgl
```

Cara penggunaan keyword `global` adalah dengan cukup menuliskannya di dalam block kemudian diikuti nama variabel. Dari output terlihat bahwa di luar block fungsi `greet()` nilai variabel `name` berubah.

A.29.4. Fungsi `globals()`

Fungsi `globals()` mengembalikan informasi semua variabel global yang bisa

diakses dari tempat dimana fungsi dipanggil. Nilai balik berbentuk dictionary dengan `key` adalah nama variabel dan `value` adalah nilai variabel.

Contoh penerapannya bisa dilihat pada kode berikut. Ada variabel bernama `my_var` yang nilainya diakses via nilai balik pemanggilan fungsi `globals()`

```
my_var = 12

def task_one():
    all_global_vars = globals()
    print(all_global_vars['my_var'])

task_one()
# output → 12
```

A.29.5. Fungsi `locals()`

Fungsi `locals()` mengembalikan informasi variabel yang dideklarasikan di block dimana fungsi tersebut dipanggil dengan nilai balik berbentuk dictionary.

Pada contoh berikut, variabel `my_var` dideklarasikan di luar block fungsi `task_two()`, sedangkan `another_var` didalamnya.

Di dalam `task_two()`, variabel `another_var` bisa diakses via nilai balik fungsi `locals()` karena masih satu block. Tidak seperti `my_var`, pengaksesannya via nilai balik fungsi `locals()` akan menghasilkan error karena variabel tersebut tidak dikenali.

```
my_var = 12

def task_two():
    another_var = "Hello Python"
```

Di contoh, ditambahkan block `try except` untuk menangkap error yang muncul saat statement `all_locals_vars['my_var']` dieksekusi.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../local-global-var
```

◉ TBA

- Variabel global pada module

◉ Referensi

- https://docs.python.org/3/faq/programming.html
- https://docs.python.org/3/library/functions.html#locals
- https://docs.python.org/3/library/functions.html#globals

A.30. Python Special Names

Chapter ini membahas tentang variabel spesial yang ada di Python (umumnya disebut special names).

Python memiliki variabel spesial yang bisa diakses secara global. Ciri khas special names adalah penulisannya diawali dan diakhiri dengan karakter `__`. Salah satunya adalah variabel `__all__` yang telah diulas di chapter sebelumnya.

Setiap special names memiliki kegunaan yang unik dan berbeda satu sama lain.

A.30.1. Variabel `__name__`

Variabel `__name__` adalah salah satu special names di Python. Isinya mencakup informasi nama modul atau string `__main__`, tergantung apakah variabel tersebut di-print dari file entrypoint eksekusi program atau di-import. File entrypoint yang dimaksud disini adalah file yang digunakan pada argument command `python <nama_file_program>`.

Agar lebih jelas, mari kita langsung praktekan. Silakan siapkan folder project baru dengan struktur file seperti ini:

Project structure example-1

```
example-1/
```

Folder project `example-1` ini berisi hanya 2 file, yaitu `calculate.py` dan `my_program.py`.

Selanjutnya, buka `calculate.py` dan tulis kode untuk mencari bilangan prima:

File: calculate.py

```
print("from calculate.py:", __name__)

def is_prime(num):
    print("from calculate.py is_prime(num):", __name__)

    flag = False

    if num == 1:
        print(num, "is not a prime number")
    elif num > 1:
        for i in range(2, num):
            if (num % i) == 0:
                flag = True
                break

        if flag:
            print(num, "is not a prime number")
        else:
            print(num, "is a prime number")
```

File `calculate.py` difungsikan sebagai module bernama `calculate`, yang nantinya di-import pada file `my_program.py`. Statement `print("from calculate.py:", __name__)` akan otomatis dieksekusi saat ter-import. Selain itu, ada juga fungsi `is_prime()` yang berisi kode pencarian bilangan prima.

> *Penjelasan detail mengenai module ada di chapter Modules*

Lanjut, buka file `my_program.py` dan isi dengan kode berikut:

File: my_program.py

```
print("from my_program.py:", __name__)

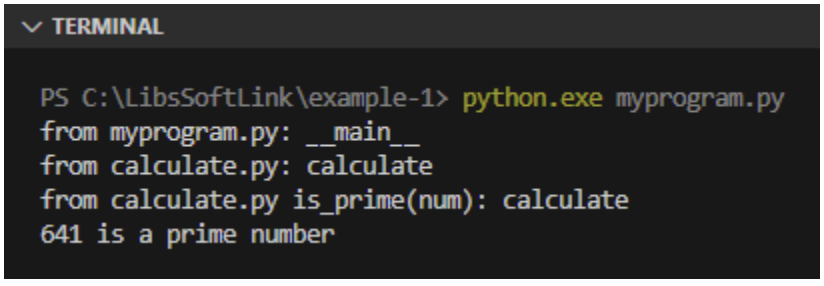
import calculate

import random
num = random.randint(0, 999)
calculate.is_prime(num)
```

File `my_program.py` nantinya kita gunakan sebagai entrypoint eksekusi program via perintah `python.exe my_program.py`. Isi file tersebut adalah 5 buah statement yang masing-masing penjelasannya bisa dilihat di bawah ini:

1. Statement `print("from my_program.py:", __name__)` dieksekusi
2. Module `calculate` di-import
3. Module `random` di-import
4. Fungsi `randint()` dalam module `random` dieksekusi
5. Fungsi `is_prime()` milik module `calculate` dieksekusi

Jalankan program menggunakan command `python my_program.py`, lalu lihat outputnya:



Dari contoh, dapat dilihat bahwa special names `__name__` jika di-print dari

`my_program.py` memiliki value string `"__main__"`. Hal ini karena file `my_program.py` adalah entrypoint program. Sedangkan pada module `calculate`, variabel yang sama menghasilkan output berbeda, yaitu `calculate` yang merupakan nama module file `calculate.py`.

> *Untuk mengetahui file mana yang merupakan file entrypoint eksekusi program, selain dengan mengecek nilai variabel `__name__` bisa juga dengan melihat argument eksekusi program. Misalnya `python my_program.py`, maka file entrypoint adalah `my_program.py`*

Variabel `__name__` biasanya dimanfaatkan sebagai kontrol entrypoint program, misalnya untuk membedakan statement yang akan dieksekusi ketika module digunakan sebagai entrypoint atau digunakan sebagai dependency atau module yang di-import di module lain.

Contoh pengaplikasian skenario yang disebutkan dapat ditemukan pada contoh program ke-2 berikut:

Project structure example-2

```
example-2/
|—— calculate.py
└—— main.py
```

File: calculate.py

```python
def is_prime(num):
    flag = False

    if num == 1:
        print(num, "is not a prime number")
```

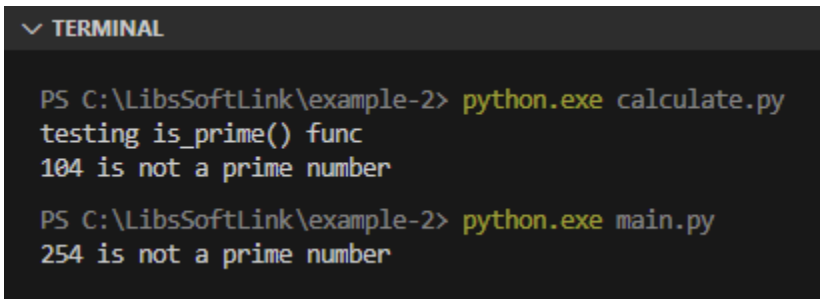
File: main.py

```
import calculate

import random
num = random.randint(0, 999)
calculate.is_prime(num)
```

Penjelasan:

- Module `calculate` jika di-run sebagai entrypoint, maka statement dalam blok seleksi kondisi `if __name__ == '__main__'` otomatis tereksekusi. Jika tidak digunakan sebagai entrypoint, maka tidak ada statement yang dieksekusi.
- Module `main` jika di-run sebagai entrypoint, maka module tersebut akan meng-import module `calculate` lalu memanggil fungsi `is_prime()` yang dideklarasikan didalamnya.



A.30.2. Variabel `__file__`

Variabel special name `__file__` berisi informasi path file di mana variabel tersebut ditulis atau digunakan. Ada dua cara untuk menggunakan variabel ini:

- Dengan mengaksesnya secara langsung

- Dengan mempergunakannya sebagai property dari module, misalnya: `random.__file__`

Contoh penerapannya bisa dilihat pada program berikut:

Project structure example-3

```
example-3/
├── calculate.py
└── main.py
```

File: calculate.py

```python
def is_prime(num):
    flag = False

    if num == 1:
        print(num, "is not a prime number")
    elif num > 1:
        for i in range(2, num):
            if (num % i) == 0:
                flag = True
                break

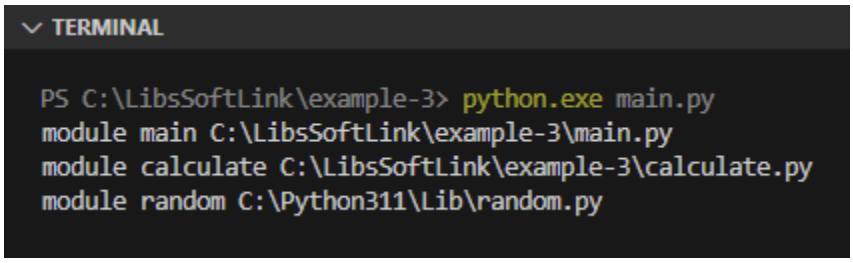
        if flag:
            print(num, "is not a prime number")
        else:
            print(num, "is a prime number")
```

File: main.py

```python
print("module main", __file__)

import calculate
```

Bisa dilihat pada gambar berikut bahwa output program adalah memunculkan nama module beserta lokasi di mana file module tersebut berada:



Penggunaan `__file__` akan menampilkan path file dimana variabel tersebut ditulis, sedangkan penggunaannya sebagai property module (misalnya `calculate.__file__`) menghasilkan informasi path module.

A.30.3. Variabel `__all__` & file `__init__.py`

Variabel `__all__` digunakan untuk menentukan module apa saja yang ter-import ketika statement `import *` digunakan. Variabel `__all__` wajib ditulis di file `__init__.py` yang ditempatkan dalam package.

> *Penjelasan detail mengenai `import *` ada pada chapter Packages.*

A.30.4. Attribute `__name__` milik class `type`

Kita telah menggunakan fungsi `type()` beberapa kali pada banyak chapter sebelum ini. Fungsi `type()` adalah fungsi yang mengembalikan data dengan tipe yaitu class `type`.

Class `type` memiliki attribute bernama `__name__` isinya informasi nama class. Contoh penerapan pengaksesan attribute ini:

```
data1 = "Noval Agung"
print(f"var: data1, data: {data1}, type: {type(data1).__name__}")
# output → var: data1, data: Noval Agung, type: str

data2 = 24 * 7
print(f"var: data2, data: {data2}, type: {type(data2).__name__}")
# output → var: data2, data: 168, type: int
```

A.30.5. Attribute `__class__` milik semua class / tipe data

Setiap tipe data memiliki akses ke attribute bernama `__class__`. Isi dari attribute ini adalah data yang sama hasil pemanggilan fungsi `type()` yaitu informasi tipe data atau class.

Pada kode sebelumnya, statement `type(data1)` menghasilkan nilai balik yang sama dengan statement `data1.__class__`. Dari nilai balik (yang bertipe `type`) tersebut bisa langsung dilakukan pengaksesan attribute `__name__`.

```
data1 = "Noval Agung"
print(f"var: data1, data: {data1}, type: {data1.__class__.__name__}")
# output → var: data1, data: Noval Agung, type: str

data2 = 24 * 7
print(f"var: data2, data: {data2}, type: {data2.__class__.__name__}")
# output → var: data2, data: 168, type: int
```

A.30.6. Attribute `__mro__` milik semua

class / tipe data

Class attribute `__mro__` berisi informasi hirarki class dalam tipe data tuple. Penjelasan lebih lanjut mengenai `__mro__` ada di chapter OOP ➜ Class Inheritance.

A.30.7. Package `__future__`

Package `__future__` berisi modules yang hanya tersedia di Python versi terbaru. Package ini biasa di-import pada program yang dijalankan menggunakan Python versi lama (misalnya 2.5), yang didalamnya ada penerapan kode yang hanya ada di versi Python terbaru.

Salah satu contoh adalah penggunaan operator `//` untuk operasi **floor division** atau pembagian dengan hasil dibulatkan. Operator tersebut hanya tersedia di Python 3.0+.

Agar bisa menggunakan operator tersebut di Python versi lama harus ada, perlu untuk meng-import module `division` dari package `__future__`. Tulis statement import di file program baris paling atas sendiri.

```
from __future__ import division

print(8 / 7)
# output ➜ 1.1428571428571428

print(8 // 7)
# output ➜ 1
```

A.30.8. Fungsi `__init__()`

Fungsi `__init__()` digunakan untuk membuat konstruktor pada suatu class. Penjelasan lebih lanjut mengenai `__init__()` ada di chapter OOP ➜ Class & Object.

A.30.9. Attribute `__doc__` milik semua class dan fungsi

Attribute `__doc__` digunakan untuk melihat informasi komentar docstring. Penjelasan lebih lanjut mengenai `__doc__` ada di chapter DocString.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../special-names
```

◉ Chapter relevan lainnya

- Modules
- Packages
- OOP ➜ Class & Object
- OOP ➜ Class Inheritance

◉ Referensi

- https://docs.python.org/3/tutorial/modules.html
- https://docs.python.org/3/tutorial/special-names.html
- https://stackoverflow.com/questions/7075082/what-is-future-in-python-used-for-and-how-when-to-use-it-and-how-it-works

A.31. Python None

Pada chapter ini kita akan belajar tentang object special bernama `None`.

A.31.1. Pengenalan `None`

`None` merupakan object bawaan Python yang umumnya digunakan untuk merepresentasikan nilai kosong atau *null*.

Ketika suatu variabel berisi data yang nilainya bisa kosong, umumnya sebelum variabel tersebut digunakan, dilakukan pengecekan terlebih dahulu menggunakan seleksi kondisi untuk memastikan apakah nilainya benar-benar kosong atau tidak.

Sebagai contoh, pada kode berikut, dipersiapkan sebuah fungsi bernama `inspect_data()`, tugasnya mengecek apakah variabel memiliki nilai atau tidak.

```
def inspect_data(data):
    if data is None:
        print("data is empty. like very empty")
    else:
        print(f"data: {data}, type: {type(data).__name__}")

data = 0
inspect_data(data)
# output → data: 0, type: int

data = ""
inspect_data(data)
# output → data: , type: str
```

Bisa dilihat pada program di atas output tiap statement adalah berbeda-beda sesuai tipe datanya.

- Ketika variabel `data` berisi `0` maka variabel tersebut tidak benar-benar kosong, melainkan berisi angka `0`.
- Karakteristik yang sama juga berlaku ketika variabel berisi string kosong `""`, meskipun ketika di-print tidak muncul apa-apa, variabel tersebut sebenarnya berisi tipe data string namun tanpa isi (string dengan panjang nol). Maka variabel tersebut sebenarnya tidak benar-benar kosong.
- Barulah ketika variabel isinya data `None` maka dianggap benar-benar kosong.

> *Kode di atas berisi penerapan salah satu special name, yaitu attribute `__name__` milik class `type`.*
>
> *Pembahasan detail mengenai special name ada di chapter Special Names ➜ Attribute **name** milik class type*

A.31.2. Penggunaan operator `is` terhadap `None`

Ketika memeriksa apakah sebuah objek bernilai `None` atau tidak, disarankan untuk menggunakan operator `is` dibanding `==`, karena sifat dari operator `==` memanggil special method `__eq__()` dari objek yang diperiksa dan pada praktiknya tidak semua class menggunakan implementasi default method `__eq__()`. Dari sini maka potensi penggunaan operator `==` menghasilkan nilai yg berbeda dibanding yang diharapkan adalah ada (meskipun sangat kecil).

Serta, operator `is` lebih cepat dibanding `==` untuk pengecekan data `None`.

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../none
```

◉ Chapter relevan lainnya

- Tipe Data
- Special Names

◉ Referensi

- https://docs.python.org/3/c-api/none.html
- https://stackoverflow.com/questions/3257919/what-is-the-difference-between-is-none-and-none
- https://stackoverflow.com/questions/26595/is-there-any-difference-between-foo-is-none-and-foo-none/26611#26611

A.32. Python Pack Unpack tuple, list, set, dict

Python mengenal teknik packing dan unpacking, dimana teknik ini umum diterapkan sesuai kebutuhan pada beberapa jenis tipe data kolektif seperti tuple, list, set, dan dictionary. Pada chapter ini kita akan mempelajari cara penggunaannya, beserta peran penggunaan tanda `*` dan `**` pada operasi packing dan unpacking.

A.32.1. Unpacking element tuple, list, set

Unpacking (yang dalam Bahasa Indonesia berarti bongkar muatan) adalah teknik pendistribusian elemen tipe data kolektif ke banyak variabel.

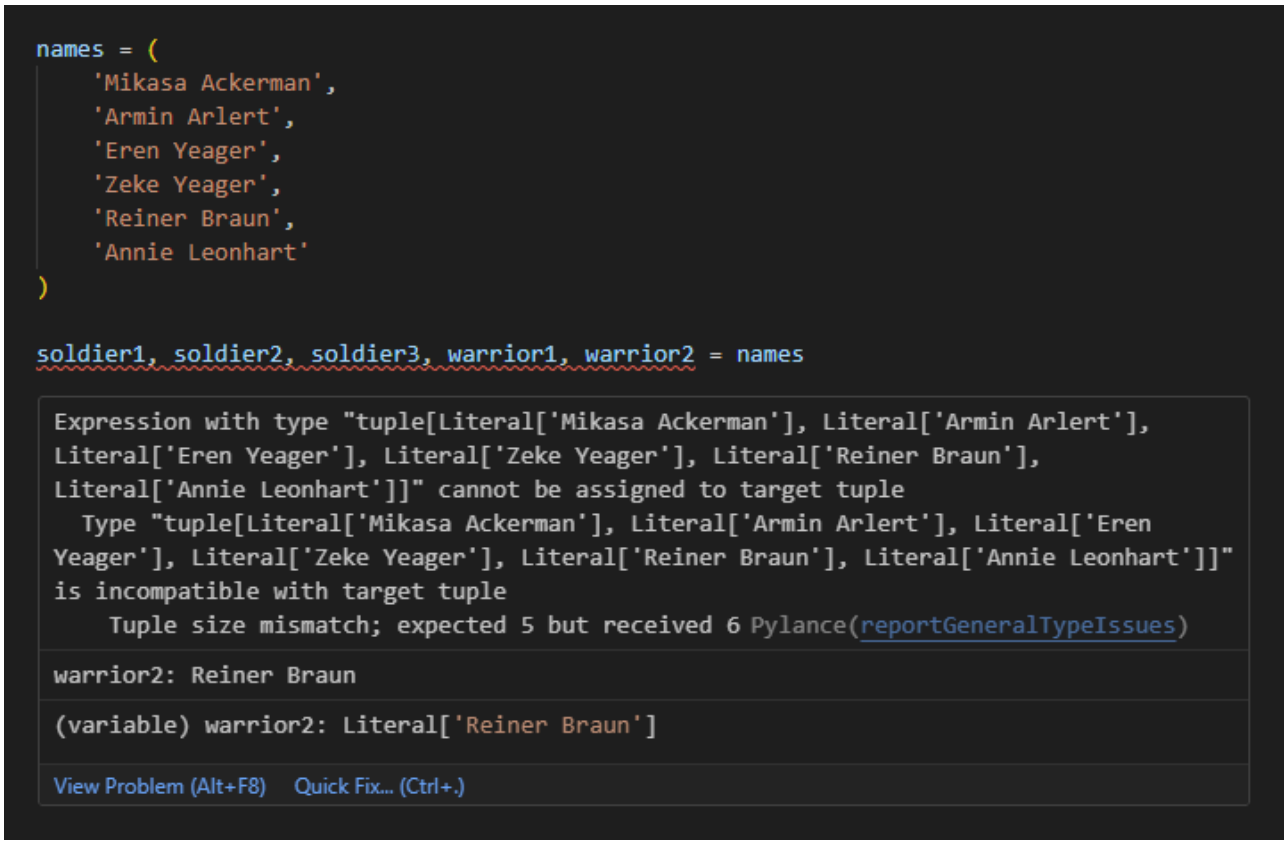
Distribusinya sendiri bisa *1 on 1* yang berarti setiap elemen ditampung 1 variabel, atau hanya beberapa elemen saja yang didistribusikan ke variabel baru, sisanya tetap ditampung dalam bentuk data kolektif.

◉ Unpack 1 element = 1 variable

Pada program berikut ada sebuah tuple bernama `names` yang setiap element-nya perlu untuk didistribusikan ke variabel independen. Karena ada 6 buah elemen dalam tuple tersebut, maka perlu disiapkan 6 buah variabel untuk menampung data masing-masing element.

```
names = (
    'Mikasa Ackerman',
    'Armin Arlert',
    'Eren Yeager',
```

Jika jumlah variabel penampung tidak sama dengan jumlah element data kolektif, misalnya jumlah variabel lebih sedikit, maka error pasti muncul saat eksekusi program.



◉ Unpack hanya `N` elements pertama

Ada alternatif cara lain untuk kasus lainnya dimana element yang perlu ditampung ke variabel baru hanya beberapa element pertama saja. Misalnya, dari 6 element tuple, hanya 3 yang perlu disimpan ke variabel baru. Hal seperti ini bisa dilakukan dengan konsekuensi: sisa element lainnya ditampung dalam bentuk kolektif di satu variabel lain.

Contoh penerapannya bisa dilihat di kode berikut. Tuple yang sama, 3 element pertamanya saja yang ditampung ke variabel baru. Untuk element sisanya, tetap harus ditampung juga tapi cukup di 1 variabel saja, tersimpan dalam

tiple data `list`.

> *Variabel penampung sisa element selalu dalam bentuk `list` meskipun sumber datanya bertipe data lain.*

```
names = (
    'Mikasa Ackerman',
    'Armin Arlert',
    'Eren Yeager',
    'Zeke Yeager',
    'Reiner Braun',
    'Annie Leonhart'
)

soldier1, soldier2, deceiver1, *warriors = names

print(soldier1)  # output → Mikasa Ackerman
print(soldier2)  # output → Armin Arlert
print(deceiver1) # output → Eren Yeager
print(warriors)  # output → ['Zeke Yeager', 'Reiner Braun', 'Annie
Leonhart']
```

Bisa dilihat dari output, bahwa 3 element pertama berhasil ditampung di 3 variabel baru. Sisanya tetap disimpan ke variabel lain yang pada contoh di atas adalah variabel `warriors`.

Penulisan variabel penampung sisa element diawali dengan karakter `*` dan pasti tipe datanya adalah list.

> *Penulisan karakter `*` di awal variabel ini wajib. Jika tidak ditulis, maka Python menganggap ada 4 buah variabel mencoba menampung element tuple.*

> *Jika diterapkan pada kasus di atas, statement tersebut pasti menghasilkan error, karena data `names` berisi 6 element.*

Katakanlah data element yang dibutuhkan hanya 3 pertama, sisanya bisa dibuang, maka bisa gunakan variabel `_` sebagai tempat pembuangan element yang tidak terpakai.

```
soldier1, soldier2, deceiver1, *_ = names

print(soldier1)  # output → Mikasa Ackerman
print(soldier2)  # output → Armin Arlert
print(deceiver1) # output → Eren Yeager
```

◉ Unpack hanya `N` elements terakhir

teknik ini mirip seperti sebelumnya, perbedaannya: variabel penampung element berada di sebelah kanan dan sisa element ditampung di variabel paling kiri.

Jadi variabel bertanda `*` harus ditulis di paling kiri.

Pada contoh berikut, element yang sama disimpan namun dalam bentuk tipe data berbeda, yaitu set.

```
names = {
    'Mikasa Ackerman',
    'Armin Arlert',
    'Eren Yeager',
    'Zeke Yeager',
    'Reiner Braun',
    'Annie Leonhart'
}
```

Variabel penampung sisa element (yaitu `soldiers`) tipe datanya `list`.

◉ Unpack hanya `N` elements pertama dan terakhir

Bagaimana jika elements yang di-unpack adalah yang posisinya ada di samping? Bisa juga, caranya dengan menampung sisanya di tengah.

Pada contoh berikut, ada sebuah list yang 2 element pertamanya ditampung ke variabel independen, 2 terakhir juga ditampung ke variabel independen, dan sisanya (yang posisinya di tengah) ditampung ke variabel lain.

Tanda `*` dituliskan pada variabel yang ditengah dengan penulisan bebas di posisi mana saja, asalkan tidak di awal dan tidak di akhir. Python secara cerdas tahu element mana yang dianggap sisa dengan melihat statementnya.

```
names = [
    'Mikasa Ackerman',
    'Armin Arlert',
    'Eren Yeager',
    'Zeke Yeager',
    'Reiner Braun',
    'Annie Leonhart'
]

soldier1, soldier2, *deceivers, warrior1, warrior2 = names

print(soldier1)  # output → Mikasa Ackerman
print(soldier2)  # output → Armin Arlert
print(deceivers) # output → ['Eren Yeager', 'Zeke Yeager']
print(warrior1)  # output → Reiner Braun
print(warrior2)  # output → Annie Leonhart
```

A.32.2. Packing element tuple, list, set

Packing element adalah operasi pemuatan banyak data ke sebuah data kolektif. Cara penerapannya sangat mudah, cukup tulis saja variabel yang ingin di-pack sebagai element data kolektif. Untuk tipenya bisa berupa tuple, list, maupun set.

```
soldier1 = 'Mikasa Ackerman'
soldier2 = 'Armin Arlert'
soldier3 = 'Eren Yeager'
warrior1 = 'Zeke Yeager'
warrior2 = 'Reiner Braun'
warrior3 = 'Annie Leonhart'

tuple1 = (soldier1, soldier2, soldier3, warrior1, warrior2, warrior3)
print(tuple1)
# output ↓
#
# ('Mikasa Ackerman',
#  'Armin Arlert',
#  'Eren Yeager',
#  'Zeke Yeager',
#  'Reiner Braun',
#  'Annie Leonhart')

list1 = [soldier1, soldier2, soldier3, warrior1, warrior2, warrior3]
print(list1)
# output ↓
#
# ['Mikasa Ackerman',
#  'Armin Arlert',
#  'Eren Yeager',
#  'Zeke Yeager',
#  'Reiner Braun',
#  'Annie Leonhart']
```

> *Perlu diingat, bahwa tipe data set tidak menjamin elemennya tersimpan secara urut. Jadinya sewaktu diakses atau di-print bisa saja urutan elemen berubah.*
>
> *Pembahasan detail mengenai set ada di chapter Set*

◉ Prepend element

Operasi prepend bisa dilakukan dengan mudah menggunakan syntax `(newElement, *oldData)`. Contohnya pada kode berikut, tuple `names` ditambahi element baru dengan posisi di awal.

```
names = [
    'Mikasa Ackerman',
    'Armin Arlert',
    'Eren Yeager',
    'Zeke Yeager',
    'Reiner Braun',
    'Annie Leonhart'
]

names = ('Jean Kirstein', *names)
print(names)
# output ↓
#
# ('Jean Kirstein',
#  'Mikasa Ackerman',
#  'Armin Arlert',
#  'Eren Yeager',
#  'Zeke Yeager',
#  'Reiner Braun',
#  'Annie Leonhart')
```

Operasi `('Jean Kirstein', *names)` menghasilkan data tuple karena disitu

literal tuple digunakan (`( )`). Selain tuple, operasi ini bisa didesain untuk ditampung ke tipe data kolektif lainnya, misalnya `list` dan `set`.

```
names = ('Jean Kirstein', *names)
print(f"type: {type(names).__name__}")
# output → type: tuple

names2 = ['Jean Kirstein', *names]
print(f"type: {type(names2).__name__}")
# output → type: list

names3 = {'Jean Kirstein', *names}
print(f"type: {type(names3).__name__}")
# output → type: set
```

◉ Append element

Append adalah operasi penambahan element baru di posisi akhir data. Penulisan syntax-nya mirip seperti prepend element namun dituliskan terbalik, seperti ini `(*oldData, newElement)`.

```
names = [
    'Mikasa Ackerman',
    'Armin Arlert',
    'Eren Yeager',
    'Zeke Yeager',
    'Reiner Braun',
    'Annie Leonhart'
]

names = ('Jean Kirstein', *names)
print(names)
# output ↓
#
# ('Jean Kirstein',
```

◉ Append dan prepend element bersamaan

Operasi append dan prepend bisa dilakukan secara bersamaan.

```
names = [
    'Mikasa Ackerman',
    'Armin Arlert',
    'Eren Yeager',
    'Zeke Yeager',
    'Reiner Braun',
    'Annie Leonhart'
]

names = ('Jean Kirstein', *names)
print(names)
# output ↓
#
# ('Jean Kirstein',
#  'Mikasa Ackerman',
#  'Armin Arlert',
#  'Eren Yeager',
#  'Zeke Yeager',
#  'Reiner Braun',
#  'Annie Leonhart')

names = [*names, 'Connie Springer']
print(names)
# output ↓
#
# ['Jean Kirstein',
#  'Mikasa Ackerman',
#  'Armin Arlert',
#  'Eren Yeager',
#  'Zeke Yeager',
#  'Reiner Braun',
#  'Annie Leonhart',
#  'Connie Springer']
```

> *Perlu diingat, bahwa tipe data set tidak menjamin elemennya tersimpan secara urut. Jadinya sewaktu diakses atau di-print bisa saja urutan elemen berubah.*
>
> *Pembahasan detail mengenai set ada di chapter Set*

A.32.3. Pemanfaatan teknik unpacking pada argument parameter

Tenik unpacking umum dimanfaatkan pada penyisipan data argument parameter pemanggilan fungsi/method. Contoh aplikasinya bisa dilihat pada kode berikut, dimana ada fungsi bernama `show_biography()` yang akan dipanggil beberapa kali dengan cara pemanggilan berbeda satu sama lain.

```
def show_biography(id, name, occupation, gender):
    print(f"id: {id}")
    print(f"name: {name}")
    print(f"occupation: {occupation}")
    print(f"gender: {gender}")
```

- Pemanggilan ke-1: Argument parameter disisipkan menggunakan cara normal.

  ```
  id = 'U0001'
  name = 'Mikasa Ackerman'
  occupation = 'Paradise Survey Corps'
  gender = 'female'
  show_biography(id, name, occupation, gender)
  ```

- Pemanggilan ke-2: Argument pertama diisi secara normal, sedangkan

parameter kedua dan seterusnya disisipkan menggunakan metode unpacking.

```
user2_id = 'U0002'
user2_data = ('Annie Leonhart', 'Marley Warrior', 'female')
show_biography(user2_id, *user2_data)
```

- Pemanggilan ke-3: metode unpacking juga digunakan tetapi untuk argument ke-1 hingga ke-3 (sesuai dengan jumlah element `user3_data`).

```
user3_data = ('U0003', 'Levi Ackerman', 'Paradise Survey Corps')
show_biography(*user3_data, 'male')
```

- Pemanggilan ke-4: Metode unpacking digunakan pada penyisipan argument parameter ke-2 dan ke-3 saja.

```
user4_data = ('Hange Zoë', 'Paradise Survey Corps')
show_biography('U0004', *user4_data, 'male')
```

A.32.4. Packing-unpacking item dictionary

◉ Operasi unpack pada dictionary

Operasi unpack pada tipe data dictionary mengembalikan hanya nilai key-nya saja dalam bentuk `list`. Nilai value-nya tidak ikut didistribusikan.

```
user_id, *user_data = data
```

◉ Operasi append & prepend pada dictionary

Di atas telah dicontohkan kalau tanda `*` bisa digunakan untuk menampung sisa pendistribusian keys dictionary (ingat, hanya keys-nya saja!), maka berarti tidak bisa digunakan untuk operasi append dan prepend dictionary.

Cara append/prepend dictionary adalah dengan memanfaatkan tanda `**`. Contoh penerapannya bisa dilihat pada kode berikut:

```
data = {
    'name': 'Mikasa Ackerman',
}
print(data)
# output → { 'name': 'Mikasa Ackerman' }

data = {
    **data,
    'occupation': 'Paradise Survey Corps',
}
print(data)
# output → { 'name': 'Mikasa Ackerman', 'occupation': 'Paradise Survey
Corps' }

data = {
    'id': 'U0001',
    **data,
    'gender': 'female',
}
print(data)
# output ↓
#
# {
#   'id': 'U0001',
#   'name': 'Mikasa Ackerman',
#   'occupation': 'Paradise Survey Corps',
#   'gender': 'female'
```

◉ Pemanfaatan teknik unpacking dictionary pada argument parameter

Tanda `**` bisa digunakan untuk operasi unpack dictionary sebagai argument pemanggilan fungsi. Key dictionary menjadi nama parameter, dan value dictionary menjadi nilai argument parameter.

Teknik ini bisa dilakukan dengan ketentuan nama parameter fungsi adalah sama dengan key dictionary.

```
def show_biography(id, name, occupation, gender):
    print(f"id: {id}")
    print(f"name: {name}")
    print(f"occupation: {occupation}")
    print(f"gender: {gender}")

data1 = {
    'id': 'U0001',
    'gender': 'female',
    'name': 'Mikasa Ackerman',
    'occupation': 'Paradise Survey Corps',
}
show_biography(**data1)
# output ↓
#
# id: U0001
# name: Mikasa Ackerman
# occupation: Paradise Survey Corps
# gender: female

data2 = {
    'gender': 'female',
    'name': 'Mikasa Ackerman',
    'occupation': 'Paradise Survey Corps',
}
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../pack-unpack-elements
```

◉ Chapter relevan lainnya

- List
- Tuple
- Set

◉ Referensi

- https://peps.python.org/pep-0448/

A.33. Python OOP ➜ Class & Object

Python mendukung paradigma pemrograman berbasis objek (OOP) melalui implementasi Class dan Object API. Pada bab ini, kita akan mempelajari konsep dasar beserta penerapannya.

> *Pembahasan OOP pada ebook ini lebih fokus pada pengaplikasiannya di Python. Jadi pembahasan teorinya tidak terlalu banyak.*

A.33.1. Pengenalan Class

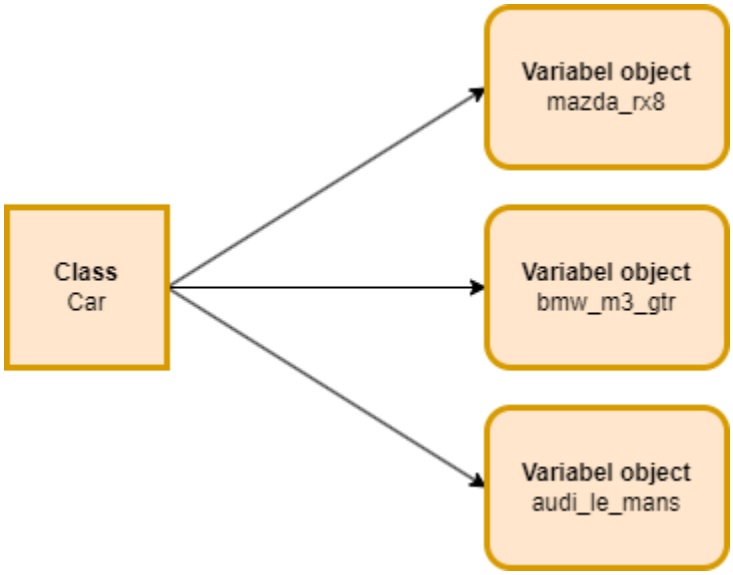
Class adalah *blueprint* untuk membuat variabel, class bisa diartikan juga sebagai tipe data. Di Python, setiap data pasti memiliki tipe data yang tipe tersebut merupakan adalah class. Sebagai contoh:

- Data string `"noval"` tipe datanya adalah class `str`
- Data numerik `24` tipe datanya adalah class `int`
- Data floating point `3.1567` tipe datanya adalah class `float`
- ... dan lainnya

Selain menggunakan class-class yang tersedia di Python Standard Library, kita bisa membuat custom class via keyword `class`. Topik custom class ini merupakan inti pembahasan chapter ini.

Custom class (atau cukup class) digunakan untuk membuat variabel object. Cara termudah memahami hubungan antara class dan objek adalah melalui analogi berikut: dimisalkan ada sebuah class bernama `Car`, class tersebut

kemudian digunakan untuk mendeklarasikan tiga buah variabel bernama `bmw_m3_gtr`, `mazda_rx8`, dan `audi_le_mans`. Ketiga object tipe datanya adalah class `Car`.



Deklarasi class dilakukan dengan menggunakan keyword `class` diikuti oleh nama class yang diinginkan. Lalu di dalam block class tersebut perlu dideklarasikan suatu fungsi dengan skema `__init__(self)` dengan isi body fungsi adalah deklarasi attribute. Contohnya:

```
class Car:
    def __init__(self):
        self.name = ""
        self.manufacturer = ""
        self.year = 0
```

Pada contoh di atas, class `Car` memiliki tiga attribute: `name`, `manufacturer`, dan `year`. Nantinya, variabel objek yang dibuat dari class tersebut akan memiliki tiga atribut sesuai dengan yang dideklarasikan.

> *Fungsi `__init__(self)` disebut dengan method konstruktor. Pembahasan detail mengenai konstruktor ada di chapter Class ➜ Constructor*

◉ Deklarasi class tanpa attribute

Dengan menggunakan keyword `pass`, suatu class bisa dideklarasikan tanpa memiliki attribute. Contoh penerapannya:

```
class Car:
    def __init__(self):
        pass
```

Atau dapat juga ditulis seperti ini:

```
class Car:
    pass
```

> *Pembahasan detail mengenai keyword `pass` ada di chapter Function section Keyword pass*

A.33.2. Naming convention class

Berdasarkan dokumentasi PEP 8 – Style Guide for Python Code, disarankan untuk menulis nama class dalam bentuk TitleCase, contoh: `FavoriteFood`.

A.33.3. Pembuatan Instance object

Object (atau instance object) adalah variabel yang dibuat dari class. Cara pembuatan object adalah dengan memanggil nama class diikuti oleh tanda kurung fungsi `()` (seperti pemanggilan fungsi). Statement tersebut mengembalikan nilai balik berupa object baru yang bertipe data sesuai dengan class yang digunakan.

> *Ada banyak istilah lain yang merujuk pada variabel objek, seperti instance, instance variable, instance object, dan lainnya. Namun tidak usah bingung, karena semua istilah tersebut memiliki makna yang sama.*

Contoh deklarasi class `Person` beserta pembuatan variabel object bernama `person1`:

```
class Person:
    def __init__(self):
        self.first_name = ""
        self.last_name = ""

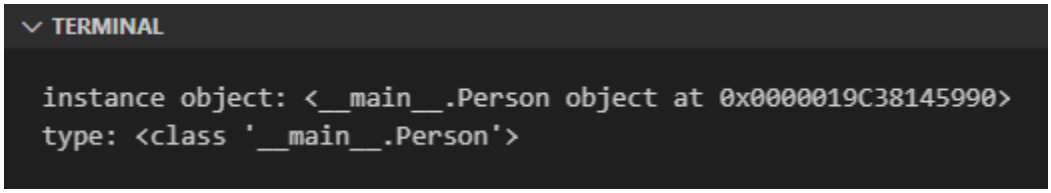
person1 = Person()
print(f"instance object: {person1}")
print(f"type: {type(person1)}")
```

Penjelasan:

- Class `Person` dideklarasikan dengan dua atribut, yaitu `first_name` dan `last_name`.
- Class `Person` dipanggil seperti pemanggilan fungsi (menggunakan sintaks `Person()`) dan menghasilkan variabel objek baru bertipe `Person`, yang

kemudian ditampung dalam variabel `person1`.

Output program:



Dari output program, terlihat bahwa tipe data variabel `person1` adalah class `__main__.Person`. Syntax tersebut artinya adalah tipe data class `Person` yang deklarasinya ada di file `__main__` atau file entrypoint eksekusi program.

Contoh lainnya pembuatan instance object dari class class `Car`:

```
class Car:
    def __init__(self):
        self.name = ""
        self.manufacturer = ""
        self.year = 0

car1 = Car()
car2 = Car()
car3 = Car()
```

A.33.4. Instance Attribute

Salah satu property class adalah attribute. Attribute adalah variabel yang terasosiasi dengan class, jadi dalam pengaksesannya harus dilakukan melalui class dan/atau instance object.

Sebelumnya, kita telah membuat class bernama `Car` yang memiliki 3 attribute:

- `name` untuk menyimpan informasi nama/seri mobil
- `manufacturer` untuk menyimpan informasi manufaktur atau pembuat mobil
- `year` untuk menyimpan informasi tahun rilis mobil

Attribute sebenarnya ada 2 jenis, yaitu instance attribute dan class attribute. **Yang sedang kita pelajari di chapter ini adalah instance attribute.**

> *Perbedaan mendetail antara instance attribute vs class attribute ada di chapter Class ➜ Class Attribute & Method*

Cara deklarasi instance attribute mirip dengan deklarasi variabel, perbedaannya pada penulisannya diawali dengan `self.`. Selain itu deklarasinya harus berada di dalam body fungsi `__init__(self)`.

Untuk mengakses instance attribute, kita dapat melakukannya melalui variabel objek yang dibuat dari class dengan notasi pengaksesan: `<object>.<attribute>`.

Contoh:

```
class Car:
    def __init__(self):
        self.name = ""
        self.manufacturer = ""
        self.year = 0

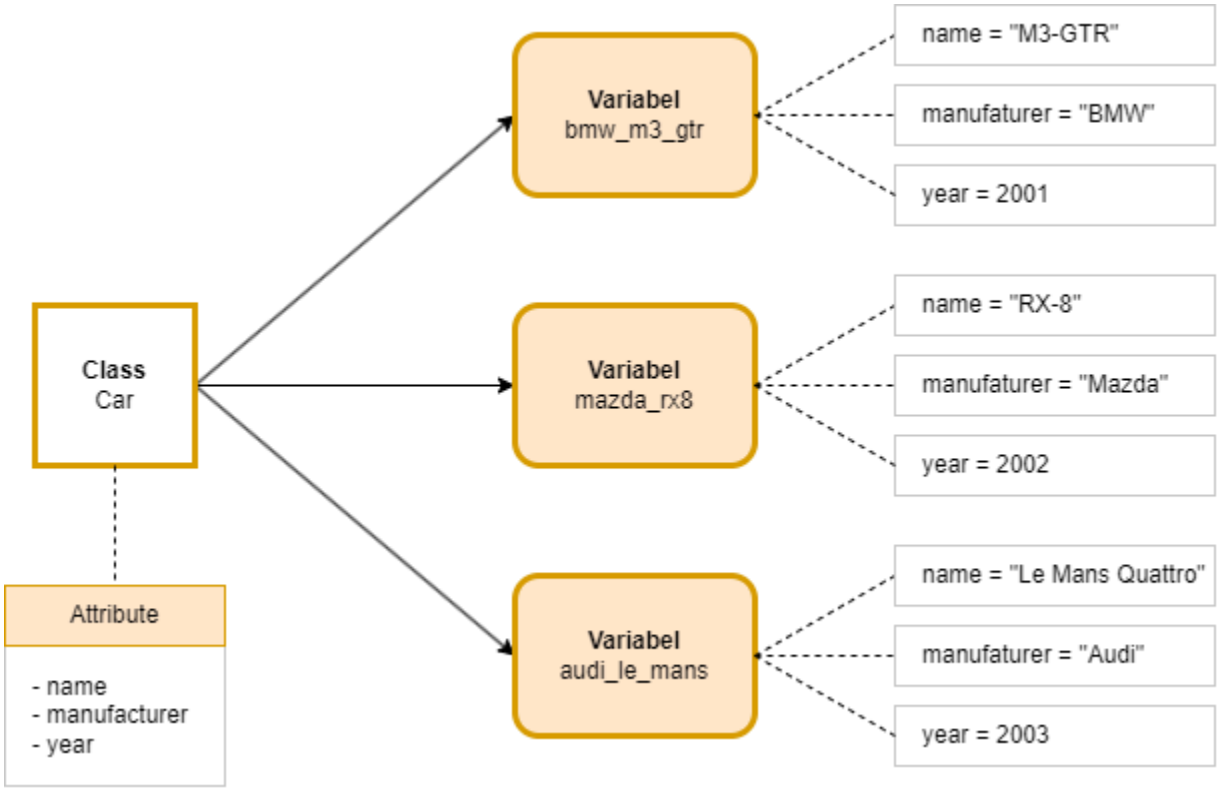
car1 = Car()
print(f"car1 name: {car1.name}")
print(f"car1 manufacturer: {car1.manufacturer}")
print(f"car1 year: {car1.year}")
```

```
TERMINAL

car1 name:
car1 manufacturer:
car1 year: 0
```

Saat di-print, dapat dilihat bahwa semua nilai instance attribute milik `car1` sesuai dengan nilai default yang ditentukan saat deklarasi attribute dalam fungsi `__init__(self)`, yaitu: string kosong `""` untuk attribute `name` & `manufacturer`, dan `0` untuk attribute `year`.

Langkah berikutnya, mari buat tiga buah variabel object dari class `Car` dan isi instance attribute-nya dengan suatu nilai. Tiga variabel yang perlu dibuat adalah `bmw_m3_gtr`, `mazda_rx8`, dan `audi_le_mans`.



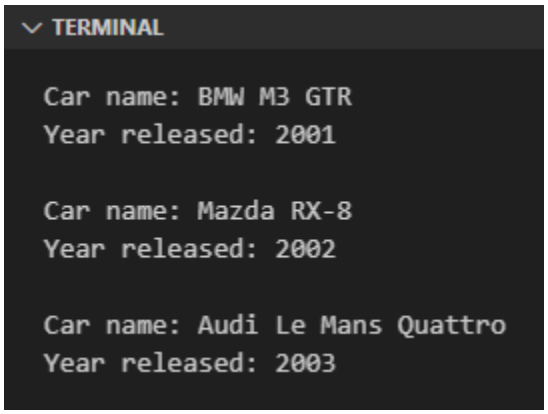
Bentuk penerapannya dalam kode Python kurang lebih seperti ini:

```
car1 = Car()
car1.name = "M3 GTR"
car1.manufacturer = "BMW"
car1.year = 2001
print(f"Car name: {car1.manufacturer} {car1.name}\nYear released:
{car1.year}\n")

car2 = Car()
car2.name = "RX-8"
car2.manufacturer = "Mazda"
car2.year = 2002
print(f"Car name: {car2.manufacturer} {car2.name}\nYear released:
{car2.year}\n")

car3 = Car()
car3.name = "Le Mans Quattro"
car3.manufacturer = "Audi"
car3.year = 2003
print(f"Car name: {car3.manufacturer} {car3.name}\nYear released:
{car3.year}\n")
```

Jalankan program untuk melihat outputnya:



> *Class jika dilihat dari strukturnya memiliki kesamaan dengan dictionary. Class mempunyai attribute name dan value, sementara dictionary*

> *memiliki key dan value.*
>
> *Perbedaan utama dari keduanya adalah pada dictionary key-nya bisa dikelola secara dinamis, sedangkan pada class nama attribute-nya adalah fixed.*

A.33.5. Pengecekan instance object

Fungsi `isinstance()` cukup berguna untuk mengecek apakah suatu instance object tipe datanya adalah class tertentu atau class yang meng-*inherit* class tertentu.

Misalnya, variabel `car1` di atas kita cek apakah tipe data nya adalah class `Car`. Cara penggunaannya cukup panggil fungsi `isinstance()` lalu sertakan variabel object yang ingin dicek sebagai argument pertama dan tipe data class sebagai argument ke-dua.

```
car1 = Car()
car1.name = "M3 GTR"
car1.manufacturer = "BMW"
car1.year = 2001

if isinstance(car1, Car):
    print(f"car1 class is Car")
# output → car1 class is Car
```

A.33.6. Class turunan `object`

Setiap class yang ada di Python baik itu class bawaan Python Standard Library ataupun custom class, secara otomatis adalah turunan dari class bernama

`object`.

Jadi, class `str`, `float`, custom class `Car` yang telah dibuat, dan lainnya, kesemua class-nya adalah turunan dari class `object`.

Silakan cek menggunakan fungsi `isinstance()` untuk membuktikannya:

```
data1 = Car()
if isinstance(data1, Car):
    print(f"data1 class inherit from Car")
if isinstance(data1, object):
    print(f"data1 class inherit from object")

data2 = "Noval Agung"
if isinstance(data2, str):
    print(f"data2 class inherit from str")
if isinstance(data2, object):
    print(f"data2 class inherit from object")

# output ↓
#
# data1 class inherit from Car
# data1 class inherit from object
# data2 class inherit from str
# data2 class inherit from object
```

> *Pembahasan detail mengenai class inheritance ada di chapter OOP → Class Inheritance*

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../class-object
```

◉ Chapter relevan lainnya

- OOP ➜ Instance Method
- OOP ➜ Constructor
- OOP ➜ Property Visibility
- OOP ➜ Instance Attribute & Class Attribute
- OOP ➜ Class Method
- OOP ➜ Static Method
- OOP ➜ Class Inheritance
- OOP ➜ Abstract Method
- OOP ➜ Data Class

◉ Referensi

- https://docs.python.org/3/tutorial/classes.html

A.34. Python OOP ➜ Instance Method

Jika attribute adalah variabel yang berasosiasi dengan class, maka method adalah fungsi yang berasosiasi dengan class.

Python mengenal 3 jenis method yaitu instance method, class method, dan static method. Chapter ini fokus ke pembahasan tentang instance method saja.

> - *Pembahasan detail mengenai class method ada di chapter OOP ➜ Class Method*
> - *Pembahasan detail mengenai static method ada di chapter OOP ➜ Static Method*

A.34.1. Pengenalan Instance Method

Instance method memiliki beberapa karakteristik jika dilihat dari syntax-nya:

1. Deklarasinya di dalam block class
2. Parameter pertamanya adalah `self`
3. Method diakses menggunakan notasi `<object>.<method>()`

Dalam praktik kali ini, kita akan melanjutkan praktek class `Car` yang telah dibuat di chapter sebelumnya.

Ok, pertama-tama siapkan deklarasi class `Car` dengan 4 buah property yaitu `name`, `manufacturer`, `year`, dan `description`. Kemudian dari class tersebut, buat 3 buah instance object baru, lalu print data attribute tiap

variabel.

```
class Car:
    def __init__(self):
        self.name = ""
        self.manufacturer = ""
        self.year = 0
        self.description = ""

all_cars = []

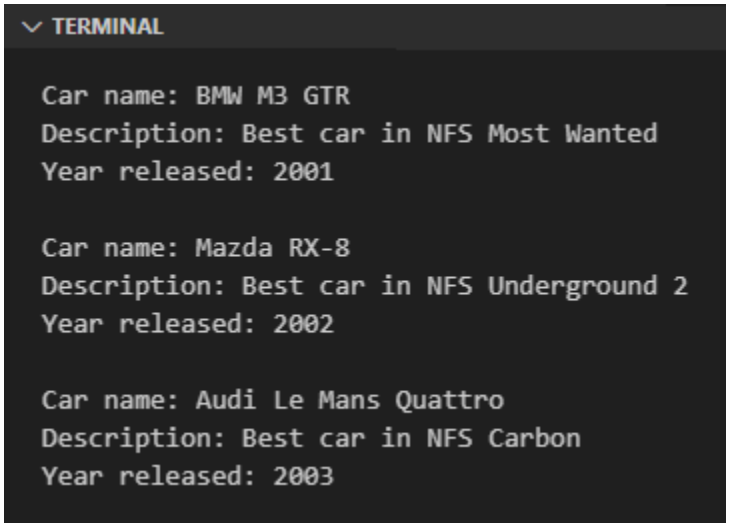
car1 = Car()
car1.name = "M3 GTR"
car1.manufacturer = "BMW"
car1.year = 2001
car1.description = "Best car in NFS Most Wanted"
all_cars.append(car1)

car2 = Car()
car2.name = "RX-8"
car2.manufacturer = "Mazda"
car2.year = 2002
car2.description = "Best car in NFS Underground 2"
all_cars.append(car2)

car3 = Car()
car3.name = "Le Mans Quattro"
car3.manufacturer = "Audi"
car3.year = 2003
car3.description = "Best car in NFS Carbon"
all_cars.append(car3)

for c in all_cars:
    print(f"Car name: {c.manufacturer} {c.name}")
    print(f"Description: {c.description}")
    print(f"Year released: {c.year}")
    print()
```

Output program:



Setelah itu, modifikasi class `Car` dengan menambahkan instance method baru bernama `info()`. Melalui method ini, value attribute di-print.

```
class Car:
    def __init__(self):
        self.name = ""
        self.manufacturer = ""
        self.year = 0
        self.description = ""

    def info(self):
        print(f"Car name: {self.manufacturer} {self.name}")
        print(f"Description: {self.description}")
        print(f"Year released: {self.year}")
```

Pada bagian blok kode perulangan, ganti tiga baris statement print dengan pemanggilan method `info()`.

- Before:

```
for c in all_cars:
    print(f"Car name: {c.manufacturer} {c.name}")
    print(f"Description: {c.description}")
    print(f"Year released: {c.year}")
    print()
```

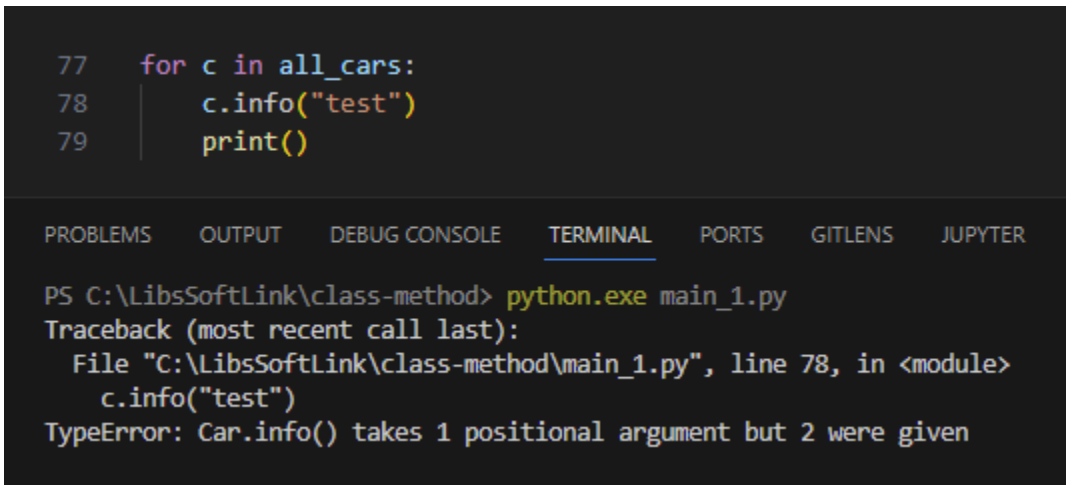
- After:

```
for c in all_cars:
    c.info()
    print()
```

Jalankan program dan lihat outputnya, pasti sama persis dengan program sebelumnya.

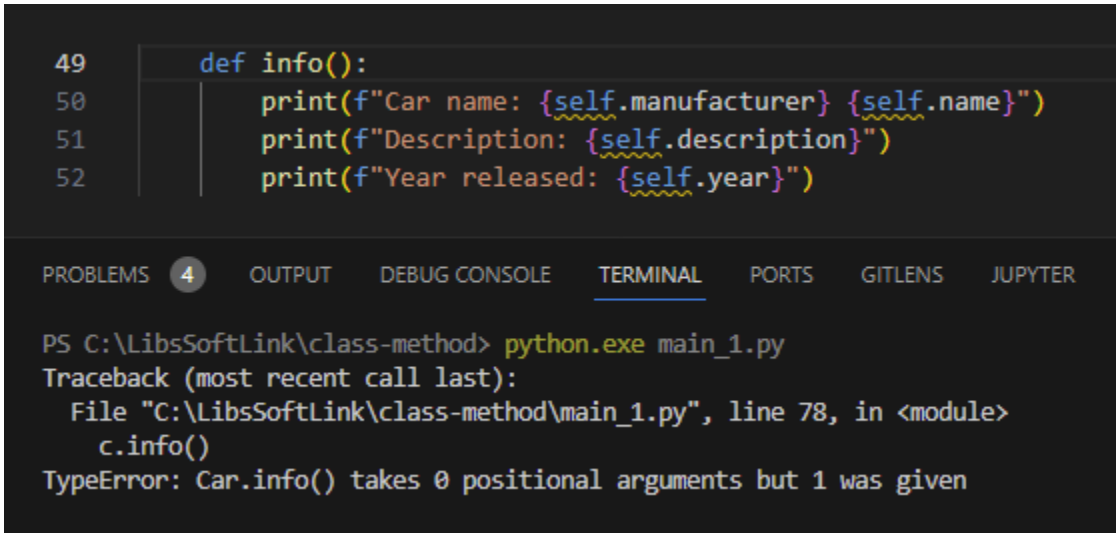
A.34.2. Variabel `self`

Salah satu aturan pada instance method adalah fungsi harus memiliki parameter pertama bernama `self`. Parameter tersebut wajib ada saat deklarasi, dan tidak boleh diisi argument saat pemanggilan. Jika dipaksa diisi dengan argument, maka pasti muncul error.



> *Parameter `self` bisa disebut dengan parameter implicit atau implisit karena kita tidak berinteraksi secara langsung saat pengisian nilai. Nilai `self` otomatis terisi saat pemanggilan instance method via instance object.*

Dimisalkan lagi, parameter `self` tidak ditulis saat deklarasi instance method, hasilnya juga error.



Parameter `self` merupakan variabel yang merepresentasikan suatu object atau instance. Melalui variabel ini, kita dapat mengakses instance attribute maupun instance method (selama property tersebut masih dalam satu class).

Pada contoh sebelumnya, terlihat bagaimana aplikasi dari variabel `self` untuk mengakses attribute:

```
class Car:
    def __init__(self):
        self.name = ""
        self.manufacturer = ""
        self.year = 0
```

Selain untuk mengakses nilainya, dari variabel `self` suatu attribute juga dapat diubah nilainya, sebagaimana pada contoh berikut nilai attribute `year` dan `description` diubah melalui pemanggilan instance method `set_details()`.

Karena instance method wajib dideklarasikan dengan parameter pertama `self`, maka parameter untuk menampung data `year` dan `description` ditulis sebagai parameter setelahnya.

```
class Car:
    def __init__(self):
        self.name = ""
        self.manufacturer = ""
        self.year = 0
        self.description = ""

    def info(self):
        print(f"Car name: {self.manufacturer} {self.name}")
        print(f"Description: {self.description}")
        print(f"Year released: {self.year}")

    def set_details(self, year, description):
        self.year = year
        self.description = description
```

Setelah itu, ubah statement pengisian nilai `year` dan `description` menggunakan method `set_details()` seperti ini:

```
all_cars = []

car1 = Car()
car1.name = "M3 GTR"
car1.manufacturer = "BMW"
car1.set_details(2001, "Best car in NFS Most Wanted")
```

Pada pemanggilan method `set_details()` object `car1`:

- Argument `2001` ditampung oleh parameter `year`
- Argument `Best car in NFS Most Wanted` ditampung oleh parameter `description`.

A.34.3. Naming convention method & param

Mengacu pada dokumentasi PEP 8 – Style Guide for Python Code, nama method dianjurkan untuk ditulis menggunakan snake_case (seperti fungsi). Contohnya bisa dilihat pada method `get_name()` berikut:

```
class FavoriteFood:
    def __init__(self):
        self.name = ""

    def print_name(self):
        print(self.name)

    def get_name(self) -> str:
        return self.name
```

Sedangkan aturan penulisan nama parameter/argument adalah sama seperti nama variabel, yaitu menggunakan snake_case juga. Misalnya:

```
class FavoriteFood:
    def __init__(self):
        self.name = ""

    def print_name(self):
        print(self.name)
```

A.34.4. Pengaksesan instance method dari class

Sebenarnya ada dua cara untuk mengakses instance method:

1. Lewat instance object, contohnya seperti kode `car1.set_details()` yang telah dipraktekan.
2. Lewat class dengan ketentuan dalam pemanggilan methodnya, parameter pertama harus diisi dengan instance object.

Silakan perhatikan kode berikut agar lebih jelas mengenai cara ke-2.

```
class FavoriteFood:
    def __init__(self):
        self.name = ""

    def print_name(self):
        print(self.name)

    def get_name(self) -> str:
        return self.name

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

food1 = FavoriteFood()
food1.set_name("Pizza")
food1.print_name()
print(food1.get_name())

FavoriteFood.set_name(food1, "Burger")
FavoriteFood.print_name(food1)
print(food1.get_name())
```

Pada kode di atas, `food1` merupakan instance object dari class `FavoriteFood`. Lewat object tersebut 3 buah method ini dipanggil: `set_name()`, `print_name()`, dan `get_name()`.

Kemudian dibawahnya lagi, method `set_name()` dan `print_name()` dipanggil ulang namun dengan syntax yang berbeda. Method dipanggil dari class dan argument parameter parameter pertamanya diisi instance object `food1`.

Penulisan pemanggilan method dari class seperti itu adalah diperbolehkan dan ekuivalen dengan pemanggilan instance method via instance object. Perbedaannya:

- Pada pengaksesan instance method via instance object, parameter `self` tidak perlu diisi.
- Pada pengaksesan instance method via class, parameter `self` harus selalu diisi dengan instance object.

Agar makin jelas, silakan lihat tabel berikut. Contoh di kolom pertama adalah ekuivalen dengan contoh di kolom ke 2.

Via instance object	Via class
`food1.set_name("Pizza")`	`FavoriteFood.set_name(food1, "Pizza")`
`food1.print_name()`	`FavoriteFood.print_name(food1)`
`food1.get_name()`	`FavoriteFood.get_name(food1)`

A.34.5. Pengaksesan method dari method lain

Lewat variabel `self` tidak hanya instance attribute yang dapat diakses, melainkan semua jenis property (termasuk instance method). Pada contoh berikut, di dalam method `info()` terdapat statement pemanggilan method yaitu `get_name()`.

Instance method `get_name()` mengembalikan data string berisi kombinasi attribute `manufacturer` dan `name`.

```
class Car:
    def __init__(self):
        self.name = ""
        self.manufacturer = ""
        self.year = 2023
        self.description = ""

    def set_details(self, year, description):
        self.year = year
        self.description = description

    def get_name(self):
        return f"{self.manufacturer} {self.name}"

    def info(self):
        print(f"Car name: {self.get_name()}")
        print(f"Description: {self.description}")
        print(f"Year released: {self.year}")
```

A.34.6. Argument method: positional,

optional, keyword arg

Aturan-aturan dalam deklarasi parameter dan pengisian argument fungsi juga berlaku pada method, diantaranya:

- Parameter method yang memiliki default value:

```
class Car:

    # ...

    def set_details(self, year = 2002, description = ""):
        self.year = year
        self.description = description

    # ...
```

- Positional argument:

```
car1 = Car()
car1.name = "M3 GTR"
car1.manufacturer = "BMW"
car1.set_details(2001, "Best car in NFS Most Wanted")
```

- Optional argument:

```
car2 = Car()
car2.name = "RX-8"
car2.manufacturer = "Mazda"
car2.set_details(description="Best car in NFS Underground 2")
```

- Keyword argument:

```
car3 = Car()
car3.name = "Le Mans Quattro"
car3.manufacturer = "Audi"
car3.set_details(description="Best car in NFS Carbon", year=2003)
```

A.34.7. Argument method: args & kwargs

Sama seperti fungsi, method juga bisa berisi parameter **args** maupun **kwargs**.

Contoh penerapan **kwargs** pada method bisa dilihat di program berikut. Modifikasi program di atas, pada method `set_details()` ubah isinya menjadi seperti ini:

```
class Car:

    # ...

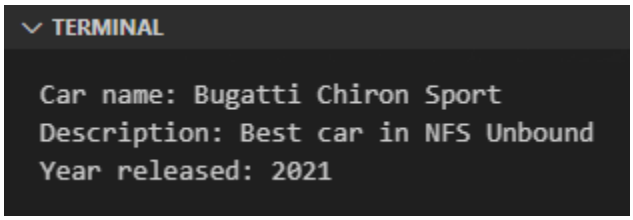
    def set_details(self, **param):
        for key in param:
            if key == "name":
                self.name = param[key]
            if key == "manufacturer":
                self.manufacturer = param[key]
            if key == "year":
                self.year = param[key]
            if key == "description":
                self.description = param[key]

    # ...
```

Melalui parameter ****param**, kita dapat menentukan attribute mana yang akan diisi nilainya secara dinamis. Sekarang panggil methodnya lalu isi sesuai kebutuhan, misalnya:

```
car4 = Car()
car4.set_details(name="Chiron Sport", manufacturer="Bugatti")
car4.set_details(year=2021)
car4.set_details(description="Best car in NFS Unbound")
car4.info()
```

Output program:



Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../instance-method
```

◉ Chapter relevan lainnya

- OOP ➜ Class & Object
- OOP ➜ Constructor
- OOP ➜ Class Method
- OOP ➜ Static Method
- OOP ➜ Abstract Method
- OOP ➜ Data Class

◉ TBA

- method & lambda
- method & closure

◉ Referensi

- https://docs.python.org/3/tutorial/classes.html

A.35. Python OOP ➜ Constructor

Constructor/konstruktor adalah salah satu topik penting dalam pemrograman berbasis object (OOP). Konstruktor sendiri adalah fungsi khusus yang dipanggil saat pembuatan object dilakukan dari suatu class.

Pada chapter ini kita akan belajar tentang konstruktor dan apa saja yang bisa dilakukan didalamnya.

A.35.1. Pengenalan constructor

Di bahasa yang mengadopsi konsep OOP, setiap class memiliki *default constructor*. Sebagai contoh pada statement pembuatan object bertipe class `Car`, syntax `Car()` digunakan dan syntax tersebut merupakan contoh penerapan pengaksesan konstruktor.

> *Sederhananya, konstruktor adalah kelas yang dipanggil seperti fungsi dengan notasi* `<Class>()`.

Agar makin jelas, silakan praktekan kode sederhana berikut:

```
class Mountain:
    pass

mount_everest = Mountain()
print(mount_everest)
# output ➜ <__main__.Mountain object at 0x0000019118A13390>
```

Class `Mountain` dideklarasikan tanpa berisi apapun (oleh karena itu keyword `pass` digunakan). Kelas tersebut bisa digunakan untuk membuat variabel object `mount_everest` dan `mount_kilimanjaro` dengan cara memanggil konstruktornya, yaitu `Mountain()`.

Dalam deklarasi class yang memiliki attribute, constructor wajib di-*replace* (atau istilah OOP-nya adalah di-*override*) dengan custom constructor yang didalamnya berisi deklarasi instance attribute.

Sebagai contoh, class `Mountain` yang telah dibuat dimodifikasi dengan ditambahkan tiga buah instance attribute didalamnya yaitu `name`, `region`, dan `elevation`.

```
class Mountain:
    def __init__(self):
        self.name = ""
        self.region = ""
        self.elevation = 0

    def info(self):
        print(f"name: {self.name}")
        print(f"region: {self.region}")
        print(f"elevation: {self.elevation}m")

mount_everest = Mountain()
mount_everest.name = "Everest"
mount_everest.region = "Asia"
mount_everest.elevation = 8848
mount_everest.info()
# output ↓
#
# name: Everest
# region: Asia
# elevation: 8848m

mount_kilimanjaro = Mountain()
```

Pada contoh di atas, konstruktor `__init__(self)` meng-override default constructor milik class `Mountain` dan digunakan untuk deklarasi instance attribute.

A.35.2. Constructor dengan custom param

Konstruktor dapat didesain untuk memiliki parameter, dan metode ini sangat umum diterapkan.

Sebagai contoh, pada kode berikut class `Mountain` konstruktornya dimodifikasi agar bisa menampung data argument untuk parameter `name`, `region`, dan `elevation`.

```
class Mountain:
    def __init__(self, name, region, elevation):
        self.name = name
        self.region = region
        self.elevation = elevation

    def info(self):
        print(f"name: {self.name}")
        print(f"region: {self.region}")
        print(f"elevation: {self.elevation}m")

mount_everest = Mountain("Everest", "Asia", 8848)
mount_everest.info()
# output ↓
#
# name: Everest
# region: Asia
# elevation: 8848m

mount_kilimanjaro = Mountain("Kilimanjaro", "Africa", 5895)
```

Seperti halnya method, parameter pertama konstruktor harus `self`. Dari sini bisa disimpulkan berarti penambahan parameter harus dituliskan setelah `self`. Bisa dilihat di deklarasi konstruktor class `Mountain`, disitu ada tiga buah parameter dideklarasikan setelah `self`.

A.35.3. Constructor overloading

Overloading merupakan istilah OOP untuk pembuatan banyak konstruktor dengan jumlah dan tipe parameter berbeda.

Python tidak menyediakan API untuk penerapan constructor overloading, namun pada praktiknya bisa dicapai dengan *hack*, misalnya menggunakan opsional/keyword parameter, atau menggunakan *args/**kwargs.

> - *Pembahasan detail mengenai opsional / keyword parameter ada di chapter Function ➜ Positional, Optional, Keyword Arguments*
> - *Pembahasan detail mengenai args dan kwargs ada di chapter Function ➜ Args & Kwargs*

Contoh constructor overloading menggunakan opsional parameter:

```
class Mountain:
    def __init__(self, name = "", region = "", elevation = 0):
        self.name = name
        self.region = region
        self.elevation = elevation

    def info(self):
        print(f"name: {self.name}")
        print(f"region: {self.region}")
        print(f"elevation: {self.elevation}m")
```

Penjelasan:

- Konstruktor class `Mountain` didesain memiliki 3 buah parameter yang kesemuanya memiliki nilai default.
- Variabel object `mount_everest` dibuat dengan dengan mengisi kesemua parameter konstruktornya.
- Variabel object `mount_kilimanjaro` dibuat dengan dengan hanya mengisi dua parameter pertama konstruktor.
- Variabel object `mount_aconcagua` dibuat dengan dengan mengisi parameter `name` dan `elevation` saja pada pemanggilan konstruktor.
- Variabel object `mount_kosciuszko` dibuat dengan tanpa diikuti dengan argument parameter.

Silakan coba explore dengan mempraktekan penggunaan *args / **kwargs pada konstruktor.

A.35.4. Constructor dengan return type `None`

Constructor dipanggil saat inisialisasi object, maka bisa dibilang bahwa constructor tersebut selalu mengembalikan tipe data bertipe class dimana constructor tersebut dideklarasikan.

Dalam pembuatan konstruktor, tidak perlu menuliskan return type maupun return statement.

Meski demikian, sebenarnya ada 1 lagi bentuk penulisan konstruktor, yaitu dengan ditambahkan tipe data `None` dibelakangnya, dan ini diperbolehkan. Contohnya:

```
class Mountain:
    def __init__(self, name = "", region = "", elevation = 0) -> None:
        self.name = name
        self.region = region
        self.elevation = elevation
```

Kode di atas adalah ekuivalen dengan kode berikut:

```
class Mountain:
    def __init__(self, name = "", region = "", elevation = 0):
        self.name = name
        self.region = region
        self.elevation = elevation
```

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../class-constructor
```

◉ Chapter relevan lainnya

- OOP ➜ Class & Object
- OOP ➜ Instance Method
- OOP ➜ Class Method
- OOP ➜ Static Method
- OOP ➜ Abstract Method
- OOP ➜ Data Class

◉ Referensi

- https://docs.python.org/3/tutorial/classes.html

A.36. Python OOP ➜ Property Visibility

Visibility atau privacy dalam konteks OOP merujuk pada penentuan apakah property (baik itu attribute atau method) dapat diakses secara public atau hanya bisa diakses dari dalam class (private).

Di bab ini, kita akan membahas implementasinya di Python.

A.36.1. Pengenalan visibility/privacy

Python, dari segi API kode yang tersedia, sebenarnya tidak secara eksplisit mendukung implementasi visibility property instance class. Semua attribute dan method secara default bersifat public di Python.

> *Property public berarti property tersebut dapat diakses melalui instance object atau dari luar block* `class`.

Meskipun demikian, ada beberapa praktik umum untuk menandai bahwa suatu method atau attribute adalah private. Salah satunya adalah menggunakan teknik *name mangling*, di mana nama attribute atau method ditulis dengan diawali 2 karakter underscore, misalnya `__name`, `__list_items`, dan sejenisnya.

Penamaan tersebut tidak benar-benar membuat visibility property menjadi private, melainkan hanya sebagai penanda saja. Property sendiri tetap bisa diakses secara publik.

Mari kita coba praktekan. Pertama siapkan project baru dengan struktur seperti

ini:

Project structure

```
property-visibility/
|─── models/
|    |─── __init__.py
|    |─── company.py
|    └─── product.py
└─── main.py
```

Package `models` berisi module `company` dan `product`, masing-masing berisi deklarasi class domain model `Company` dan `Product`. Sedangkan file `main.py` merupakan file entrypoint program.

Isi kedua tersebut dengan kode berikut:

models/product.py

```
class Product:

    def __init__(self, name, category):
        self.name = name
        self.category = category
        self.__version = 1.0
```

models/product.py

```
class Company:

    def __init__(self, name = "", products =[]):
        self.name = name
        self.products = products
```

Class `Company` memiliki 4 buah fungsi:

- Constructor `__init__()` yang memiliki parameter `name` dan `products`. Argument parameter tersebut datanya disimpan ke attribute.
- Fungsi `__print_name()`, tugasnya memunculkan isi attribute `name`. Dari namanya (yang diawali karakter `__`) bisa disimpulkan bahwa fungsi ini didesain hanya untuk digunakan secara private (digunakan dalam blok class saja) dan tidak diperuntukan untuk public.
- Fungsi `__print_products()`, tugasnya memunculkan isi data produk. Fungsi ini sama seperti `__print_name()` yaitu di-desain memiliki visibility private.
- Method `info()` yang didalamnya memanggil method `__print_name()` dan `__print_products()`.

Selain yang telah disebutkan di atas, class `Company` dan `Product` memiliki 1 property private bernama `__version` dimana property ini kita isi dengan informasi versi class.

Selanjutnya, buka `main.py`, import kedua class tersebut, kemudian buat beberapa data, lalu print isinya.

main.py

```
from models import company
from models import product

if __name__ == "__main__":
    data = []

    c1 = company.Company(name="Microsoft", products=[
```

Output program:

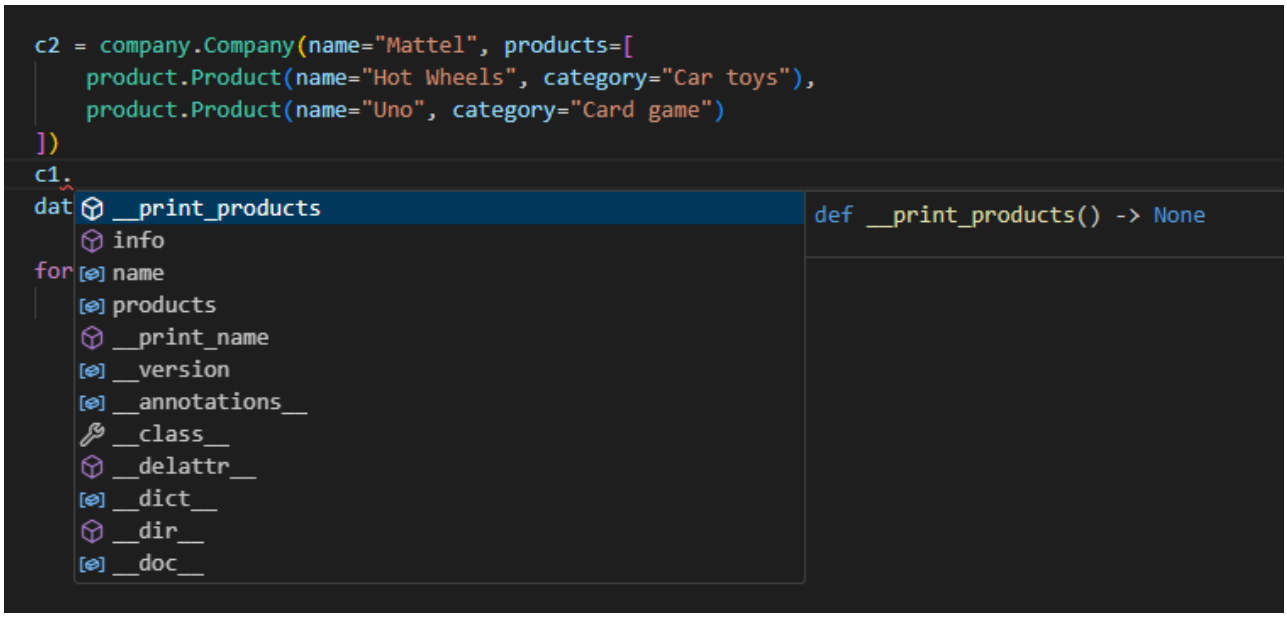
```
TERMINAL

PS C:\LibsSoftLink\dasarpemrogramanpython\examples\property-visibility> python.exe main.py
company name: Microsoft
products:
  -> Windows (Operating system)
  -> Office 365 (Productivity software)

company name: Mattel
products:
  -> Hot Wheels (Car toys)
  -> Uno (Card game)
```

A.36.2. Property dengan prefix `___`

Sebelumnya telah disebutkan bahwa prefix `__` dalam penerapannya tidak benar-benar membuat property menjadi private. Silakan test dengan autocompletion editor, property private masih muncul, menandakan bahwa property tersebut tidak benar-benar private.



Meskipun demikian, solusi ini cukup efektif untuk memberi petunjuk kepada programmer bahwa property tersebut tidak didesain untuk konsumsi publik.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../property-visibility
```

◉ Chapter relevan lainnya

- OOP ➜ Class & Object
- OOP ➜ Instance Attribute & Class Attribute

◉ Referensi

- https://docs.python.org/3/tutorial/classes.html#private-variables

A.37. Python OOP ➜ Instance Attribute & Class Attribute

Pada chapter ini, kita akan belajar lebih dalam tentang attribute suatu class, jenis-jenisnya, dan penerapannya.

A.37.1. Attributes

Attribute merupakan salah satu property class selain method. Python mengenal dua jenis attribute yaitu instance attribute (yang sudah kita bahas di beberapa chapter sebelumnya) dan class attribute.

◉ Instance attribute

Instance attribute adalah variabel yang terasosiasi dengan instance object, jadi pengaksesannya harus lewat object. Contoh penerapan instance attribute:

```
class Pencil:

    def __init__(self):
        self.note = "A class type to represent a book"

pencil1 = Pencil()
print(f"Object pencil1 note: {pencil1.note}")
# output ➜ Object pencil1 note: A class type to represent a book
```

Pada contoh di atas, variabel `note` merupakan instance attribute milik class

`Pencil`. Ciri khas attribute bisa dilihat dari tempat deklarasinya, yaitu di dalam konstruktor dan menempel pada variabel `self`.

> *Instance attribute terkadang disebut dengan **data attribute** atau **instance variable**. Ketiga istilah tersebut merepresentasikan maksud yang sama.*

◉ Class attribute

Class attribute adalah variabel yang terasosiasi dengan class yang pengaksesanya bisa langsung dari class atau bisa juga via object (seperti pengaksesan instance attribute). Contoh penerapan class attribute:

```
class Book:
    note = "A class type to represent a book"

print(f"Class Book note: {Book.note}")
# output → Class Book note: A class type to represent a book
```

Berbeda dibanding contoh sebelumnya, kali ini variabel `note` dideklarasikan langsung di bawah blok `class` (tidak di dalam konstruktor) dan tidak ditempelkan ke variabel `self`. Cara deklarasi class attribute bisa dibilang sama seperti deklarasi variabel bisa.

Class attribute bisa diakses langsung dari class contohnya seperti pada kode di atas, selain itu bisa juga diakses lewat instance object. Contoh:

```
class Book:
    note = "A class type to represent a book"

print(f"Class Book note: {Book.note}")
```

> *Class attribute terkadang disebut dengan **class variable**. Kedua istilah tersebut merepresentasikan maksud yang sama.*

◉ Kombinasi instance attribute & class attribute

Instance attribute dan class attribute keduanya bisa digunakan secara bersamaan dalam satu class yang sama. Penerapannya cukup umum, misalnya pada kasus dimana ada class yang memiliki 2 kategori attribute. Misalnya pada class `Song` berikut, ada attribute yang berasosiasi langsung dengan class yaitu `note` & `version`; dan ada juga attribute lainnya yang berasosiasi dengan instance object.

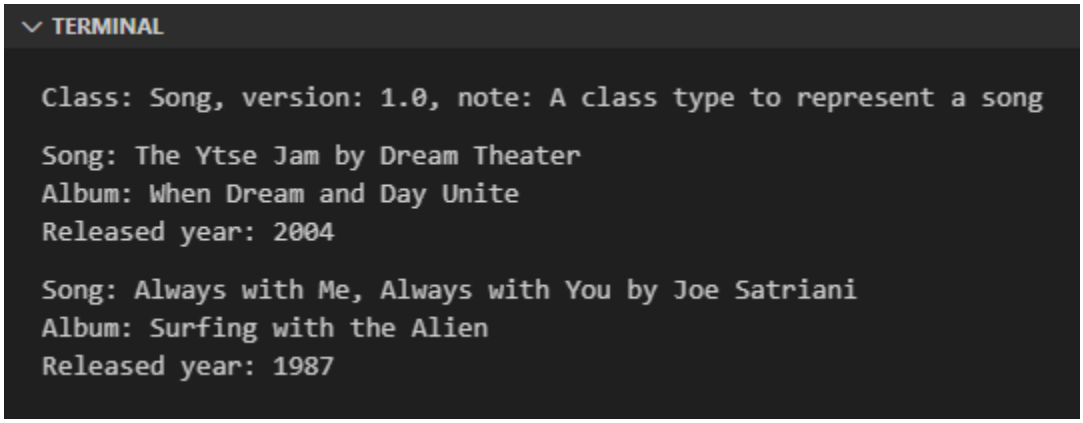
```
class Song:
    note = "A class type to represent a song"
    version = 1.0

    def __init__(self, name = "", artist = "", album = "", released_year
= 2000):
        self.name = name
        self.artist = artist
        self.album = album
        self.released_year = released_year

    def info(self):
        print(f"Song: {self.name} by {self.artist}")
        print(f"Album: {self.album}")
        print(f"Released year: {self.released_year}")

songs = [
    Song(
        name="The Ytse Jam",
        artist="Dream Theater",
        album="When Dream and Day Unite",
        released_year=2004
```

Output program:



Class attribute biasanya diterapkan untuk menyimpan data yang sifatnya global dan tidak terpaut ke object, contohnya seperti attribute `note` dan `version` pada class `Song` di atas.

Berbeda dengan instance attribute yang digunakan untuk menyimpan data spesifik per object. Bisa dilihat di contoh ada list berisi 2 element yang dibuat dari class `Song`. Masing-masing instance object tersebut data attribute-nya berbeda satu sama lain.

A.37.2. Attribute lookup

Saat suatu instance attribute diakses dari instance object, yang terjadi di balik layar adalah Python melakukan *lookup* (atau pengecekan) terhadap attribute dengan urutan sebagai berikut:

1. Jika instance attribute ditemukan, maka Python mengembalikan value instance attribute.
2. Jika instance attribute yang dicari tidak ditemukan, maka Python mengembalikan value class attribute.
3. Jika class attribute yang dicari tidak ditemukan, maka error.

Dari kondisi lookup di atas bisa disimpulkan bahwa sewaktu pengaksesan instance attribute, Python selalu memprioritaskan data yang ada di instance attribute dibanding property lainnya.

Lalu bagaiamana jika misalnya ada class yang miliki class attribute dan juga instance attribute yang namanya sama persis? Jawabannya: Python tetap memprioritaskan nilai instance attribute.

Contoh bisa dilihat pada kode berikut:

```
class Person:
    name = "A person"

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

person1 = Person("Noval Agung Prayogo")
print(f"Object person1 name: {person1.name}")
# output → Object person1 name: Noval Agung Prayogo

print(f"Class Person name: {Person.name}")
# output → Class Person name: A person
```

Variabel `person1` ketika diakses attribute `name`-nya, yang dikembalikan adalah nilai instance attribute, meskipun sebenarnya class `Person` juga memiliki class attribute dengan nama yang sama.

A.37.3. Attribute mutability

◉ Perubahan nilai instance attribute

Instance attribute datanya adalah menempel ke instance object. Jadinya, setiap object bisa saja memiliki attribute dengan value berbeda satu sama lain.

Dimisalkan ada satu variabel object yang nilai attribute-nya diubah, maka efek perubahan hanya terjadi di variabel tersebut saja, tidak berefek ke variabel lain. Agar lebih jelas silakan lihat contoh berikut:

```
class Pencil:

    def __init__(self):
        self.note = "A class type to represent a pencil"

pencil1 = Pencil()
pencil1.note = "A pencil"
pencil2 = Pencil()

print(f"Object pencil1 note: {pencil1.note}")
# output → Object pencil1 note: A pencil

print(f"Object pencil2 note: {pencil2.note}")
# output → Object pencil2 note: A class type to represent a pencil
```

◉ Perubahan nilai class attribute dari instance object

Bagaimana jika attribute yang diubah adalah class attribute, dan perubahan dilakukan lewat instance object? Jawabannya: nilai baru hasil operasi assignment tersebut akan ditampung sebagai nilai instance attribute dan efeknya hanya ada pada object saja (tidak berefek ke class). Contoh:

```
class Book:
    note = "A class type to represent a book"

book1 = Book()
book2 = Book()
book2.note = "A book"

print(f"Class Book note: {Book.note}")
```

Bisa dilihat pada bagian statement `book2.note = "A book"` efek peruabahannya hanya pada instance object-nya (`book2`). Class attribute `Book.note` nilainya tetap.

◉ Perubahan nilai class attribute secara langsung

Beda lagi untuk kasus dimana attribute yang diubah nilainya adalah class attribute dengan perubahan dilakukan secara langsung dari class-nya. Perubahan tersebut akan berefek ke semua object dan class itu sendiri.

Sebagai contoh, pada kode berikut, object `book1` dan `book2` dibuat dari class `Book`. Kemudian class attribute `Book.note` diubah nilainya, efeknya: class attribute dalam `book1` dan `book2` juga ikut berubah.

```
class Book:
    note = "A class type to represent a book"

book1 = Book()
book2 = Book()

Book.note = "A book"

print(f"Class Book note: {Book.note}")
# output → Class Book note: A book

print(f"Object book1 note: {book1.note}")
# output → Object book1 note: A book

print(f"Object book2 note: {book2.note}")
# output → Object book2 note: A book
```

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../instance-attribute-class-attribute
```

◉ Chapter relevan lainnya

- OOP ➜ Class & Object
- OOP ➜ Property Visibility

◉ TBA

- list-type attribute behaviour on class attribute vs instance attribute

◉ Referensi

- https://docs.python.org/3/tutorial/classes.html

A.38. Python OOP ➜ Class Method

Pada chapter ini kita akan belajar tentang jenis method lain yang tersedia di Python, yaitu class method, serta perbedaannya dibanding instance method dan constructor.

A.38.1. Pengenalan Class method

Class method adalah method yang pemiliknya adalah class dengan pengaksesan adalah via class, berbeda dibanding instance method yang diperuntukan untuk instance object. Jika instance method memiliki parameter `self` yang isinya adalah instance object, maka class method memiliki parameter `cls` yang isinya adalah tipe data class.

Jika dilihat dari segi deklarasinya, class method dan instance method memiliki perbedaan berikut:

- Instance method memiliki parameter pertama bernama `self`, dengan isi adalah instance object.
- Class method memiliki parameter pertama bernama `cls` yang merupakan alias dari class dimana method tersebut dideklarasikan (misalnya class method dideklarasikan dalam class `Car`, maka paramter `cls` berisi tipe data class `Car`).
- Selain perbedaan di atas, class method dideklariskan dengan menuliskan decorator `@classmethod`

> *Pembahasan detail mengenai decorator dibahas pada chapter Decorator*

Agar lebih jelas silakan pelajari kode berikut lalu praktekan. Disini sebuah class bernama `ClanHouse` dibuat dengan isi 3 buah fungsi:

- Constructor `__init__()` dengan overload parameter `name` dan `house`
- Class method `create()` digunakan untuk membuat instance object `ClassHouse`
- Instance method `info()` digunakan untuk menampilkan data `name` dan `house`

```
class ClanHouse:

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

    @classmethod
    def create(cls):
        obj = cls()
        return obj

    def info(self):
        print(f"{self.name} of {self.house}")
```

Bisa dilihat perbedaan deklarasi instance method dan class method di contoh tersebut. Method `create()` memiliki parameter pertama `cls` yang merupakan alias untuk tipe data class `ClanHouse`. Lewat `cls`, dibuat sebuah instance object bernama `obj` kemudian dijadikan nilai balik method `create()`.

> *Statement `obj = cls()` dalam method `create()` adalah ekuivalen dengan `obj = ClanHouse()`,*
> *karena dalam method tersebut nilai `cls` adalah tipe data class dimana*

> *method dideklarasikan, yaitu* `ClanHouse`

Selanjutnya, dari class `ClanHouse` akan dibuat 3 buah instance object berikut:

- Object `p1` dibuat menggunakan constructor
- Object `p2` dibuat menggunakan constructor juga, dengan parameter di-overload
- Object `p3` dibuat menggunakan class method `create()`. Class method tersebut diakses dari class, bisa dilihat dari syntax-nya yaitu `ClanHouse.create()`.

```
p1 = ClanHouse()
p1.name = "Paul Atriedes"
p1.house = "House of Atriedes"
p1.info()
# output → Paul Atriedes of House of Atriedes

p2 = ClanHouse("Lady Jessica", "Bene Gesserit")
p2.info()
# output → Lady Jessica of Bene Gesserit

p3 = ClanHouse.create()
p3.name = "Baron Vladimir Harkonnen"
p3.house = "House of Harkonnen"
p3.info()
# output → Baron Vladimir Harkonnen of House of Harkonnen
```

> *Parameter* `cls` *bisa disebut dengan parameter implicit atau implisit karena kita tidak berinteraksi secara langsung saat pengisian nilai. Nilai* `cls` *otomatis terisi saat class method diakses.*

A.38.2. Class method parameter

Class method juga bisa memiliki parameter seperti umumnya fungsi. Jika pada instance method dan constructor parameter adalah ditulis setelah `self`, pada class method parameter ditulis setelah `cls`. Contoh:

```
class ClanHouse:

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

    @classmethod
    def create(cls, name = "", house = ""):
        obj = cls()
        obj.name = name
        obj.house = house
        return obj

    def info(self):
        print(f"{self.name} of {self.house}")

p2 = ClanHouse("Lady Jessica", "Bene Gesserit")
p2.info()
# output → Lady Jessica of Bene Gesserit

p4 = ClanHouse.create("Glossu Rabban", "House of Harkonnen")
p4.info()
# output → Glossu Rabban of House of Harkonnen
```

Dari kode di atas bisa dilihat bahwa parameter `cls` milik class method diperlakukan mirip seperti parameter `self` milik constructor dan instance method.

- Pada saat pengaksesan instance method atau constructor, parameter `self` adalah diabaikan karena otomatis berisi instance object.
- Sifat yang sama juga berlaku pada parameter `cls` pada class method. Saat diakses via class (contoh: `ClanHouse.create()`), parameter `cls` diabaikan.

Parameter `cls` pada method `create()` berisi tipe data class `ClanHouse`, dan pembuatan instance object selalu via pemanggilan nama class, maka dari sini pemanggilan `cls()` dalam method `create()` juga bisa diikuti dengan pengisian argument parameter.

Sebagai perbandingan, kedua bentuk pemanggilan constructor via `cls()` berikut adalah ekuivalen:

- Cara 1: variabel `cls` digunakan dipanggil sebagai constructor tanpa parameter

```
class ClanHouse:

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

    @classmethod
    def create(cls, name = "", house = ""):
        obj = cls()
        obj.name = name
        obj.house = house
        return obj
```

- Cara 2: variabel `cls` digunakan dipanggil sebagai constructor dengan diisi argument parameter

```
class ClanHouse:

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

    @classmethod
    def create(cls, name = "", house = ""):
        obj = cls(name, house)
        return obj
```

A.38.3. Pengaksesan class method via instance object

Sampai sini penulis rasa bisa dipahami perbedaan cara pengaksesan antara instance method dan class method. Instance method diakses via instance object, dan class method diakses via class.

Selain cara tersebut, sebenarnya class method bisa juga diakses via instance object *lho*, dan hal seperti ini diperbolehkan penerapannya. Caranya bisa dilihat pada kode berikut:

```
class ClanHouse:

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

    def info(self):
        print(f"{self.name} of {self.house}")

    @classmethod
    def create(cls, name = "", house = ""):
```

Dari kode di atas bisa dilihat perbedaan dari sisi pembuatan object dan pengaksesan method antara `p2` dan `p4`.

- Instance object `p2` dibuat via constructor `ClanHouse()`
- Instance object `p4` dibuat via class method `create()`
- Dari kedua object, diakses method `info()`

Yang menarik untuk dibahas adalah `p5`. Object `p5` dibuat dari pemanggilan class method `create()` namun pengaksesannya adalah via instance object `p2`. Penulisan seperti itu diperbolehkan. Parameter `cls` pada class method `create()` akan terisi dengan nilai tipe data class object `p2` (yaitu `ClanHouse`).

A.38.4. Pengaksesan instance method via class

Jika class method bisa diakses via instance object, instance method juga bisa diakses via Class. Caranya cukup panggil instance method via class lalu isi parameter `self` dengan instance object. Contoh:

```
p2 = ClanHouse("Lady Jessica", "Bene Gesserit")
ClanHouse.info(p2)
# output → Lady Jessica of Bene Gesserit

p4 = ClanHouse.create("Glossu Rabban", "House of Harkonnen")
ClanHouse.info(p4)
# output → Glossu Rabban of House of Harkonnen

p5 = p2.create("Irulan Corrino", "Corrino Empire")
ClanHouse.info(p5)
# output → Irulan Corrino of Corrino Empire
```

Pengaksesan instance method via class mengharuskan parameter `self` milik method untuk diisi dengan object. Hal ini berbeda dibanding pengaksesan instance method via instance object dimana parameter `self` otomatis terisi nilai instance object.

A.38.5. Pengaksesan class attribute via `cls`

Pada chapter sebelumnya, OOP ➜ Instance Attribute & Class Attribute, kita telah mempelajari tentang perbedaan instance attribute dibanding class attribute.

Class attribute bisa diakses via instance object maupun class. Dalam konteks class method dimana `cls` adalah berisi tipe data class, pengaksesan class attribute memungkinkan untuk dilakukan via variabel `cls`.

Contoh penerapannya bisa dilihat pada kode berikut:

```
class ClanHouse:

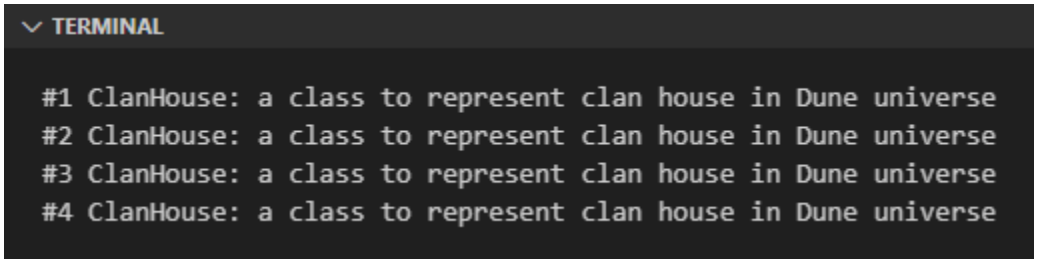
    note = "ClanHouse: a class to represent clan house in Dune universe"

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

    @classmethod
    def create(cls, name = "", house = ""):
        print("#1", cls.note)

        obj = cls(name, house)
        print("#2", obj.note)
```

Output program:



A.38.6. Summary

Dari banyak hal yang telah dipelajari di chapter ini, secara garis besar perbedaan antara constructor, instance method, dan class method bisa dilihat di bawah ini:

◉ Constructor

- Fungsi dideklarasikan di dalam block `class`
- Deklarasinya menggunakan nama fungsi `__init__()`
- Parameter pertama harus `self`, berisi instance object
- Pemanggilan constructor mengembalikan instance object
- Pengaksesannya via pemanggilan nama class, contoh: `ClanHouse()`

◉ Instance method

- Fungsi dideklarasikan di dalam block `class`
- Parameter pertama harus `self`, berisi instance object
- Pengaksesan instance method:
  - Via instance object, contoh: `p2.info()`
  - Via class dengan menyisipkan instance object sebagai argument pemanggilan. contoh: `ClanHouse.info(p2)`

◉ Class method

- Fungsi dideklarasikan di dalam block `class`
- Fungsi memiliki decorator `@classmethod`
- Parameter pertama harus `cls`, berisi tipe data class
- Pengaksesan class method:
  - Via class, contoh: `ClanHouse.create()`
  - Via instance object, contoh: `p2.create()`

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../class-method
```

◉ Chapter relevan lainnya

- OOP ➜ Class & Object
- OOP ➜ Instance Method
- OOP ➜ Constructor
- OOP ➜ Static Method
- Function ➜ Decorator
- OOP ➜ Abstract Method

◉ Referensi

- https://docs.python.org/3/tutorial/classes.html

A.39. Python OOP ➜ Static Method

Chapter ini membahas tentang static method beserta penggunaan dan perbedaannya dibanding jenis method lainnya.

A.39.1. Pengenalan static method

Telah kita pelajari bahwa suatu fungsi agar dikenali sebagai method harus dideklarasikan di dalam block `class` dan memiliki parameter implicit `self` (untuk instance method) dan `cls` (untuk class method).

Selain dua method tersebut, ada lagi jenis method lain bernama static method atau method statis, yang ciri khasnya adalah memiliki decorator `@staticmethod` dan tidak memiliki parameter *implicit* `self` maupun `cls`.

Static method bisa diakses via instance object maupun via class. Contoh penerapannya bisa dilihat pada kode berikut:

```
class Person:

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

    def info(self):
        print(f"{self.name}")

    @classmethod
    def create(cls, name = ""):
        obj = cls()
```

Pada contoh di atas, ada dua buah static method dideklarasikan:

- Method `say_hello()` dideklarasikan tanpa parameter
- Method `say_something()` dideklarasikan dengan 2 buah parameter

Kedua method tersebut diakses untuk memunculkan 5 buah output berbeda via instance object maupun via class `Person`:

- Method `say_hello()` dipanggil 2x via instance object `edward` dan via class `Person`
- Method `say_something()` juga sama, diakses via instance object 2x dan diakses via class 1x

A.39.2. Fungsi `staticmethod()`

Python menyediakan fungsi bernama `staticmethod()` yang kegunaannya adalah untuk mengkonversi fungsi biasa (yang dideklarasikan di luar class) menjadi static method milik suatu class.

Sebagai contoh, kode praktek yang telah ditulis kita *refator* menjadi seperti ini. Fungsi `say_hello()` dan `say_something()` dideklarasikan sebagai fungsi biasa. Kemudian dijadikan sebagai class method milik class `Person` via peneparan `staticmethod()`.

```
def say_hello():
    print("hello")

def say_something(message, name = None):
    if name != None:
        print(f"{name} said: {message}")
    else:
        print(message)
```

Cara penerapan fungsi `staticmethod()` adalah dengan cukup memanggilnya untuk membungkus fungsi biasa, lalu nilai baliknya ditampung sebagai attribute class.

```
class Person:

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

    say_hello = staticmethod(say_hello)
    say_something = staticmethod(say_something)
```

Attribute `say_hello` dan `say_something` keduanya menjadi static method.

Nama class attribute penampung pemanggilan fungsi `staticmethod()` bisa nama apapun, tidak harus sama dengan nama fungsi aslinya. Contohnya bisa dilihat pada kode berikut, fungsi `say_something()` dijadikan sebagai class method bernama `greet()` milik class `Person`.

```
def say_something(message, name = None):
    if name != None:
        print(f"{name} said: {message}")
    else:
        print(message)

class Person:

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

    greet = staticmethod(say_something)

p5 = Person("Ezio Auditore da Firenze")
p5.greet("hello", p5.name)
# output → Ezio Auditore da Firenze said: hello
```

Fungsi `say_something()` sendiri tetap bisa digunakan secara normal meskipun telah dijadikan sebagai static method milik class `Person`.

A.39.3. Summary

Perbedaan antara constructor, instance method, class method, dan static method bisa dilihat di bawah ini:

◉ Constructor

- Fungsi dideklarasikan di dalam block `class`
- Deklarasinya menggunakan nama fungsi `__init__()`
- Parameter pertama harus `self`, berisi instance object
- Pemanggilan constructor mengembalikan instance object
- Pengaksesannya via pemanggilan nama class, contoh: `Person()`

◉ Instance method

- Fungsi dideklarasikan di dalam block `class`
- Parameter pertama harus `self`, berisi instance object
- Pengaksesan instance method:
  - Via instance object, contoh: `p2.info()`
  - Via class dengan menyisipkan instance object sebagai argument pemanggilan. contoh: `Person.info(p2)`

◉ Class method

- Fungsi dideklarasikan di dalam block `class`
- Fungsi memiliki decorator `@classmethod`

- Parameter pertama harus `cls`, berisi tipe data class
- Pengaksesan class method:
  - Via class, contoh: `Person.create()`
  - Via instance object, contoh: `p2.create()`

◉ Static method

- Fungsi dideklarasikan di dalam block `class`
- Fungsi memiliki decorator `@staticmethod`
- **Tidak** memiliki implicit parameter `self` maupun `cls`
- Pengaksesan static method:
  - Via class, contoh: `Person.say_hello()`
  - Via instance object, contoh: `p1.say_hello()`

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../static-method
```

◉ Chapter relevan lainnya

- OOP ➜ Class & Object
- OOP ➜ Instance Method
- OOP ➜ Constructor
- OOP ➜ Class Method

- Function ➜ Decorator
- OOP ➜ Abstract Method

◉ Referensi

- https://docs.python.org/3/tutorial/classes.html

A.40. Python Decorator

Chapter ini membahas tentang decorator, yaitu syntax yang penulisannya diawali dengan huruf `@` dituliskan tepat sebelum deklarasi fungsi atau method.

A.40.1. Pengenalan decorator

Pada dua chapter sebelumnya (OOP ➜ Class Method dan OOP ➜ Static Method) kita telah mempelajari 2 buah decorator, yaitu `@classmethod` dan `@staticmethod`.

Decorator tersebut merupakan bawaan dari Python Standard Library, kita bisa langsung menggunakannya sesuai kebutuhan.

Setiap decorator memiliki tugas spesifik yang berbeda satu sama lain, misalnya:

- Decorator `@classmethod` digunakan untuk menandai suatu method adalah class method
- Decorator `@staticmethod` digunakan untuk menandai suatu method adalah static method

Selain dua decorator tersebut, ada juga beberapa lainnya yang nantinya akan dibahas ketika sudah masuk topik yang relevan dengan kegunaan masing-masing decorator.

Selain menggunakan decorator bawaan Python, kita juga bisa membuat custom decorator sendiri yang bisa kita desain sesuai kebutuhan. Pada chapter ini kita akan mempelajari caranya.

A.40.2. Custom decorator

Decorator sebenarnya adalah sebuah fungsi, yang menerima parameter berupa fungsi, yang isinya juga mengembalikan fungsi/closure.

Agar lebih jelas, silakan lihat kode berikut. Terdapat sebuah fungsi bernama `inspeksi()` yang parameter dan nilai baliknya adalah fungsi. Fungsi `inspeksi()` ini dipergunakan sebagai decorator fungsi lain yaitu `say_hello()`.

```
def inspeksi(func):
    def inner_func():
        return func()
    return inner_func

@inspeksi
def say_hello():
    print("hello world")

say_hello()
# output → hello world
```

Bisa dilihat di kode, bahwa fungsi `inspeksi` bisa langsung digunakan sebagai decorator dengan cukup menambahkan karakter `@` sebagai prefix lalu dituliskan tepat sebelum deklarasi fungsi.

> *Untuk nama closure tidak ada aturan yang mengikat harus dinamai apa. Pada contoh di atas closure fungsi `inspeksi()` penulis namai `inner_func()`.*

Fungsi `say_hello()` ditempeli decorator `@inspeksi`, artinya pemanggilan

fungsi tersebut adalah ekuivalen dengan statement ke-2 kode berikut:

```
say_hello()
# output → hello world

inspeksi(say_hello)()
# output → hello world
```

Penerapan decorator `@inspeksi` membuat eksekusi fungsi yang ditempelinya menjadi terbungkus oleh fungsi `inspeksi()`.

Ok, sekarang kita breakdown lebih detail lagi tentang fungsi `inspeksi()` yang sudah dibuat. Kegunaan dari fungsi ini sebenarnya apa? Jawabannya adalah **tidak ada**, tidak ada gunanya sama sekali, karena ini hanyalah contoh versi sederhana tentang penerapan custom decorator.

Agar fungsi `inspeksi()` lebih berguna, mari kita tambahkan sesuatu. Tepat sebelum dan sesudah pemanggilan parameter closure `func` milik fungsi `inspeksi()`, tambahkan statement print.

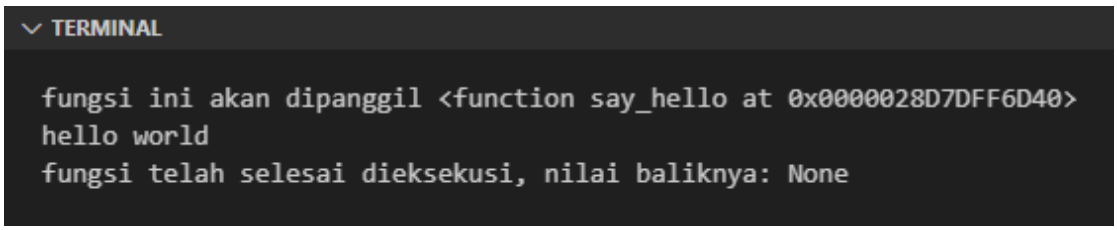
```
def inspeksi(func):
    def inner_func():
        print("fungsi ini akan dipanggil", func)
        res = func()
        print("fungsi telah selesai dieksekusi, nilai baliknya:", res)
        return res

    return inner_func

@inspeksi
def say_hello():
    print("hello world")

say_hello()
```

Jalankan program, outputnya kurang lebih seperti ini:



Bisa dilihat sekarang ada pesan muncul sebelum string `hello world` di-print. Hal ini karena kita telah menambahkan 2 statement print yang ditempatkan sebelum dan setelah eksekusi fungsi.

> *Penulis akan cukup sering menggunakan istilah **fungsi decorator** pada chapter ini, dan istilah tersebut mengacu ke fungsi yang digunakan sebagai decorator (bukan mengacu ke fungsi yang ditempeli decorator).*
>
> *Pada contoh di atas, fungsi decorator yang dimaksud adalah fungsi `inspeksi()`.*

Fungsi `say_hello()` tidak memiliki nilai balik, jadinya fungsi tersebut mengembalikan tipe data `None`. Bisa dilihat pada pesan ke-2 nilai balik pemanggilan `func()` disitu adalah `None`.

Jika diilustrasikan statement print-nya saja, program di atas mengeksekusi 3 statement ini:

```
print("fungsi ini akan dipanggil", func)
print("hello world")
print("fungsi telah selesai dieksekusi, nilai baliknya:", res)
```

Sampai sini semoga cukup jelas. Selanjutnya kita akan praktek penerapan decorator untuk case yang tidak sesederhana contoh di atas, dengan harapan pemahaman pembaca mengenai topik decorator ini makin mantab.

A.40.3. Contoh penerapan custom decorator

Pada praktek selanjutnya ini, kita akan membuat program yang memunculkan list berisi angka random. Kemudian dari list tersebut dibentuk sebuah list baru berisi elemen unik, lalu darinya dibuat list baru lagi yang isi elemen uniknya diurutkan secara *descending*.

◉ Tahap 1: Program awal

Pertama, tulis kode berikut kemudian jalankan:

```
import random

def generate_random_list(length):
    r = []

    for _ in range(0, length):
        n = random.randint(0, 10)
        r.append(n)

    return r

def unique_list(data):
    s = set(data)
    r = list(s)
    return r

def reverse_list(data):
    data.sort(reverse=True)
    return data

data = generate_random_list(15)
print("data:", data)
```

O iya, karena disini module `random` digunakan untuk generate elemen list, bisa jadi hasil generate di lokal masing-masing adalah berbeda dengan angka yang muncul di tutorial ini.

◉ Tahap 2: Decorator unique & reverse

Selanjutnya, kita refactor fungsi `unique_list()` dan `reverse_list()` yang sudah di tulis menjadi decorator.

- Fungsi `unique_list()` diubah menjadi fungsi decorator bernama `decorator_unique_list()`

```
def decorator_unique_list(func):

    def execute(length):
        data = func(length)
        s = set(data)
        r = list(s)
        return r

    return execute
```

- Fungsi `reverse_list()` diubah menjadi fungsi decorator bernama `decorator_reverse_list()`

```
def decorator_reverse_list(func):

    def execute(length):
        data = func(length)
        data.sort(reverse=True)
        return data

    return execute
```

Bisa dilihat pada kedua decorator yang telah ditulis, kode utama masing-masing decorator dituliskan setelah statement `data = func(length)`. Variabel `data` disitu isinya adalah hasil pemanggilan method dimana decorator ditempelkan nantinya. Variabel tersebut kemudian diolah sesuai dengan kebutuhan.

- Decorator `decorator_unique_list()` menghasilkan data list berisi elemen unik
- Decorator `decorator_reverse_list()` menghasilkan data list berisi elemen dengan urutan terbalik

> *Disini penulis menggunakan prefix `decorator_` pada nama fungsi untuk membedakan mana fungsi biasa dan mana fungsi decorator.*

◉ Tahap 3: Decorator dipergunakan

Selanjutnya, dua buah fungsi baru dibuat yang masing-masing menggunakan decorator yang telah dipersiapkan:

- Fungsi `generate_random_unique_list()` ditempeli decorator `@decorator_unique_list`, membuat data nilai balik fungsi ini diteruskan ke proses pencarian list berisi elemen unik.

  ```
  @decorator_unique_list
  def generate_random_unique_list(length):
      return generate_random_list(length)

  print(generate_random_unique_list(15))
  # output → [0, 3, 4, 5, 6, 7, 8, 9, 10]
  ```

  Fungsi `generate_random_unique_list()` menghasilkan proses yang ekuivalen dengan kode berikut:

```
data = generate_random_list(length)
res = unique_list(data)
print(res)
```

- Fungsi `generate_random_reverse_sorted_list()` ditempeli decorator `@decorator_reverse_list`, membuat data nilai balik fungsi ini diteruskan ke proses perubahan pengurutan elemen menjadi terbalik.

```
@decorator_reverse_list
def generate_random_reverse_sorted_list(length):
    return generate_random_list(length)

print(generate_random_reverse_sorted_list(15))
# output → [10, 10, 10, 9, 8, 8, 8, 8, 8, 7, 4, 4, 2, 0, 0]
```

Fungsi `generate_random_reverse_sorted_list()` menghasilkan proses yang ekuivalen dengan kode berikut:

```
data = generate_random_list(length)
res = reverse_list(data)
print(res)
```

A.40.4. Chaining decorator

Chaining decorator adalah istilah untuk penerapan lebih dari satu decorator pada sebuah fungsi. Pada contoh di atas, fungsi ditempeli hanya satu decorator saja. Pada praktiknya, fungsi bisa saja menggunakan lebih dari 1 decorator.

Misalnya pada program berikut, kita buat fungsi baru yang menggunakan decorator `@decorator_unique_list` dan juga `@decorator_reverse_list`.

```python
def decorator_unique_list(func):

    def execute(length):
        print("decorator_unique_list | before", func)
        data = func(length)
        print("decorator_unique_list | after")
        s = set(data)
        r = list(s)
        return r

    return execute

def decorator_reverse_list(func):

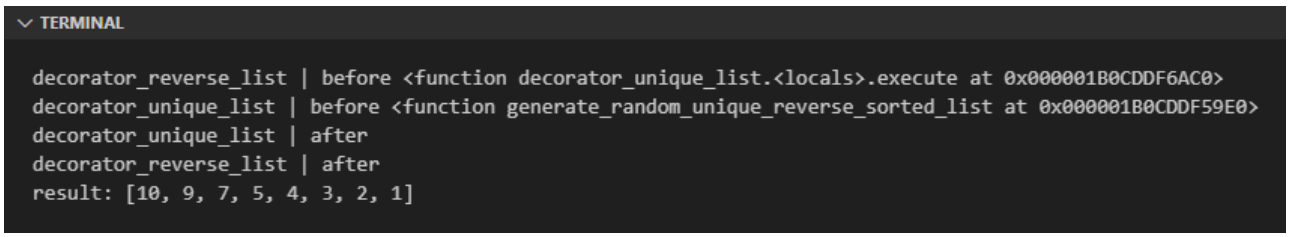
    def execute(length):
        print("decorator_reverse_list | before", func)
        data = func(length)
        print("decorator_reverse_list | after")
        data.sort(reverse=True)
        return data

    return execute

@decorator_reverse_list
@decorator_unique_list
def generate_random_unique_reverse_sorted_list(length):
    return generate_random_list(length)

print("result:", generate_random_unique_reverse_sorted_list(15))
```

Output eksekusi program:



Bisa dilihat, data list yang dihasilkan adalah unik dan urutannya terbalik, menandakan dua decorator yang kita pasang ke fungsi `generate_random_unique_reverse_sorted_list()` bekerja dengan baik.

Pada chaining decorator, urutan eksekusi fungsi decorator adalah dari yang paling bawah kemudian ke atas. Ilustrasi eksekusi fungsi dan decorator pada contoh yang telah dipraktikan kurang lebih ekuivalen dengan kode di bawah ini:

```
data =
decorator_reverse_list(decorator_unique_list(generate_random_list(length)))
print(data)

# ... atau ...

data1 = generate_random_list(length)
data2 = decorator_unique_list(data1)
data3 = decorator_reverse_list(data2)
print(data3)
```

A.40.5. *args & **kwargs pada decorator

Idealnya, sebuah decorator dibuat dengan desain parameter se-fleksibel mungkin, karena bisa saja decorator diterapkan pada fungsi dengan berbagai macam skema parameter.

Pada contoh yang telah dipraktekan, closure nilai balik fungsi decorator memiliki parameter yang sangat spesifik, yaitu `length`. Dari sini berarti decorator tersebut hanya bisa digunakan pada fungsi yang parameternya sesuai.

Ada tips atau *best practice* dalam mendesain fungsi decorator. Gunakan parameter ***args** & ****kwargs** pada deklarasi *inner function*, kemudian saat memanggil `func` lakukan operasi *unpacking*. Dengan ini parameter apapun

yang disisipkan di fungsi yang ditempeli decorator, akan di-pass ke decorator sesuai dengan aslinya.

> *Pembahasan detail mengenai teknik **unpacking** ada di chapter Packing & Unpacking*

Source code final:

```
import random

def generate_random_list(length):
    r = []

    for _ in range(0, length):
        n = random.randint(0, 10)
        r.append(n)

    return r

def decorator_unique_list(func):

    def execute(*args, **kwargs):
        data = func(*args, **kwargs)
        s = set(data)
        r = list(s)
        return r

    return execute

def decorator_reverse_list(func):

    def execute(*args, **kwargs):
        data = func(*args, **kwargs)
        data.sort(reverse=True)
        return data

    return execute
```

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../decorator
```

◉ Chapter relevan lainnya

- Function ➜ Positional, Optional, Keyword Arguments
- OOP ➜ Class Method
- OOP ➜ Static Method
- OOP ➜ Abstract Method
- OOP ➜ Data Class

◉ Referensi

- https://peps.python.org/pep-0318/
- https://peps.python.org/pep-0448/
- https://python101.pythonlibrary.org/chapter25_decorators.html
- https://stackoverflow.com/questions/6392739/what-does-the-at-symbol-do-in-python

A.41. Python OOP ➜ Class Inheritance

Chapter ini membahas tentang salah satu aspek penting dalam pemrograman OOP, yaitu inheritance atau pewarisan sifat, dimana sifat yang dimaksud adalah property seperti attribute, method, dan lainnya.

A.41.1. Pengenalan Inheritance

Untuk mewujudkan inheritance setidaknya dua buah class dibutuhkan:

- Super class / parent class, yaitu class yang property-nya ingin diwariskan atau diturunkan ke class dibawahnya.
- Sub class / derrived class, yaitu class yang mewarisi property dari parent class.

Misalkan, ada sebuah class bernama `Vehicle` dan class ini memiliki property berikut:

1. Constructor
2. Class attribute `note`
3. Instance attribute `name`
4. Instance attribute `number_of_wheels`
5. Instance method `drive_sound()`

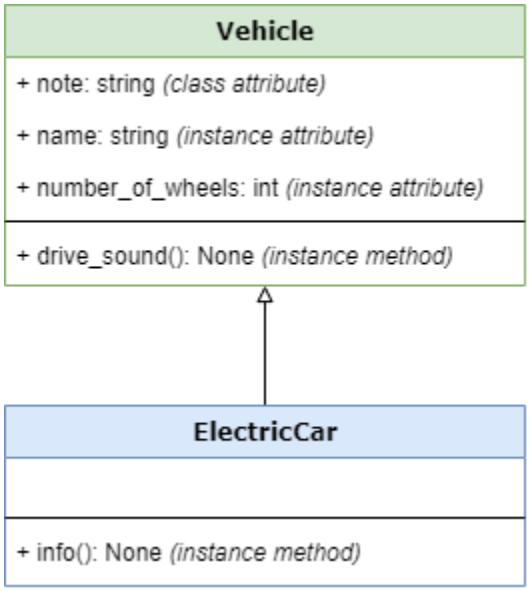
Class `Vehicle` kemudian dijadikan sebagai parent class pada class `ElectricCar`.

Class `ElectricCar` kita desain tidak memiliki attribute. Namun karena dia

merupakan *sub class* dari `Vehicle` maka secara *implicit* mewarisi semua property yang ada di class `Vehicle`. Maka via object bertipe class `ElectricCar` nantinya kita bisa mengakses property class `Vehicle`.

Class `ElectricCar` memiliki satu buah method bernama `info()` yang isinya adalah menampilkan data property yang diwarisi oleh class `Vehicle`

Ilustrasi diagram UML-nya seperti ini:



Dari gambar di atas, secara teori, object yang dibuat dari class `ElectricCar` bisa mengakses property class itu sendiri serta property lain yang diwarisi super class.

Sedangkan object dari class `Vehicle` hanya bisa mengakses property class itu sendiri saja.

Ok, sekarang mari kita terapkan skenario di atas di Python. Definisikan class `Vehicle` dan class `ElectricCar` beserta isi masing-masing property.

```
class Vehicle:
    note = "class to represent a car"

    def __init__(self):
        self.name = "common vehicle"
        self.number_of_wheels = 4

    def drive_sound(self):
        return "vroom vroooommmm"

class ElectricCar(Vehicle):
    def info(self):
        print(self.name, "has", self.number_of_wheels, "wheels. engine
sound:", self.drive_sound())
```

Pada deklarasi class `ElectricCar` penulisannya ada yang unik. Nama class ditulis dengan notasi `ElectricCar(Vehicle)` yang artinya adalah class `ElectricCar` dideklarasikan sebagai sub class class `Vehicle`.

> **INFO**
>
> Notasi umum penulisan inheritance kurang lebih seperti ini:
>
> ```
> class SuperClass:
>     pass
>
> class SubClass(SuperClass):
>     pass
> ```

Dari dua class yang telah dideklarasikan, selanjutnya buat beberapa instance object lalu akses property-nya. Setelah itu coba run program dan lihat outputnya.

```
v1 = Vehicle()
print(v1.name, "has", v1.number_of_wheels, "wheels. engine sound:",
v1.drive_sound())
# output → common vehicle has 4 wheels. engine sound: vroom vroooommmm

v2 = ElectricCar()
v2.name = "electric car"
print(v2.name, "has", v2.number_of_wheels, "wheels. engine sound:",
v2.drive_sound())
# output → electric car has 4 wheels. engine sound: vroom vroooommmm

v3 = ElectricCar()
v3.name = "electric car"
v3.info()
# output → electric car has 4 wheels. engine sound: vroom vroooommmm
```

Bisa dilihat dari contoh, bahwa property milik class `Vehicle` bisa diakses via instance object yang dibuat dari class itu sendiri maupun dari object yang dibuat dari subclass `ElectricCar`.

A.41.2. Class `object` inheritance

Python memiliki class bawaan bernama `object` yang pada praktiknya otomatis menjadi super class dari semua class bawaan Python maupun custom class yang kita buat sendiri.

Contohnya class `Vehicle` dan `ElectricCar` yang telah dibuat, kedua class tersebut otomatis juga menjadi sub class dari class `object` ini.

Untuk membuktikan, silakan test saja menggunakan kombinasi seleksi kondisi dan fungsi `isinstance()`.

```
class Vehicle:
```

A.41.3. Constructor overriding

Overriding adalah istilah pemrograman OOP untuk menimpa/mengganti suatu method dengan method baru yang nama dan strukturnya sama tapi isinya berbeda.

Pada section ini, teknik overriding kita akan terapkan pada constructor. Constructor `Vehicle` yang secara *implicit* diwariskan ke class `ElectricCar`, di sub class-nya kita replace dengan constructor baru. Silakan pelajari kode berikut agar lebih jelas:

```
class Vehicle:
    note = "class to represent a car"

    def __init__(self):
        self.name = "common vehicle"
        self.number_of_wheels = 4

    def drive_sound(self):
        return "vroom vroooommmm"

class ElectricCar(Vehicle):
    def __init__(self):
        self.name = "electric car"

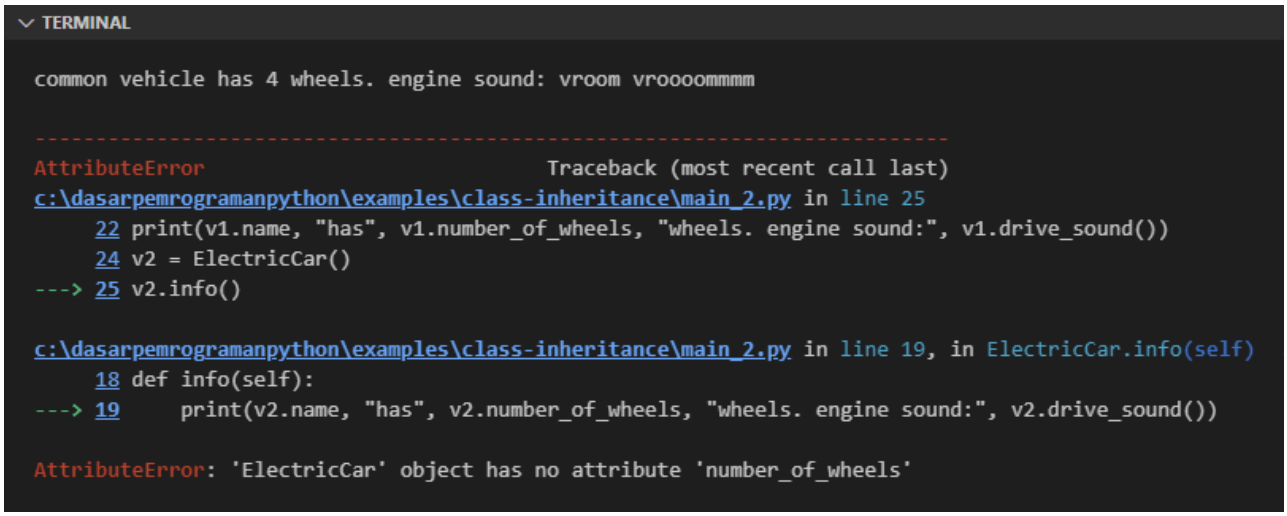
    def info(self):
        print(self.name, "has", self.number_of_wheels, "wheels. engine
sound:", self.drive_sound())

v1 = Vehicle()
print(v1.name, "has", v1.number_of_wheels, "wheels. engine sound:",
v1.drive_sound())

v2 = ElectricCar()
```

Perbedaan kode ini dibanding sebelumnya ada di bagian constructor class `ElectricCar`. Disitu untuk setiap object baru yang dibuat, nilai attribute `name`-nya diisi dengan string `electric car`.

Secara teori, idealnya program di atas bisa jalan normal. Maka mari coba run saja dan lihat hasilnya:



Oops! Statement pengaksesan property object `v1` berjalan normal, namun error muncul pada statement print ke-2 dimana property object `v2` diakses.

Pesan errornya kurang lebih menginformasikan bahwa class `ElectricCar` tidak memiliki attribute `number_of_wheels`. Aneh, padahal secara teori property tersebut diwariskan oleh super class yaitu `Vehicle`, namun setelah ditambahkan kode constructor baru yang meng-override constructor parent class, programnya malah error.

Perlu diketahui bahwa penerapan operasi override mengakibatkan kode pada super class benar-benar dihapus dan diganti dengan kode baru. Pada contoh yang sudah ditulis, di constructor `Vehicle` ada dua buah property dideklarasikan, yaitu `name` dan `number_of_wheels`. Sedangkan pada class `ElectricCar`, hanya property `name` dideklarasikan.

```
class Vehicle:
    def __init__(self):
        self.name = "common vehicle"
        self.number_of_wheels = 4

# ... vs ...

class ElectricCar(Vehicle):
    def __init__(self):
        self.name = "electric car"
```

Constructor baru milik class `ElectricCar` menimpa constructor milik super class-nya. Dan pada constructor baru ini property `number_of_wheels` tidak dideklarasikan. Efeknya, property tersebut menjadi tidak ada, menyebabkan pesan error seperti berikut:

```
AttributeError: 'ElectricCar' object has no attribute
'number_of_wheels'
```

Solusi permasalahan di atas ada pada penjelasan section berikut ini.

A.41.4. Fungsi `super()`

Fungsi `super()` adalah salah satu fungsi istimewa bawaan python, yang ketika diakses di dalam suatu instance method maka pemanggilannya mengarah ke variabel `self` milik super class (bukan variabel `self` milik class itu sendiri).

Misalnya statement `super()` ditulis pada constructor class `ElectricCar`, maka dari fungsi tersebut kita mendapatkan akses ke object `self` milik super class yaitu class `Vehicle`. Kemudian dari object `self`, property super class

bisa diakses dengan mudah. Termasuk constructor super class juga bisa diakses.

Ok, sekarang mari coba tambahkan statement `super()` pada constructor `ElectricCar`, lalu dari nilai balik fungsi, chain lagi dengan mengakses constructor `__init__()` milik super class.

Terapkan perubahan tersebut lalu jalankan ulang program. Sekarang error tidak akan muncul.

```
class Vehicle:
    note = "class to represent a car"

    def __init__(self):
        self.name = "common vehicle"
        self.number_of_wheels = 4

    def drive_sound(self):
        return "vroom vroooommmm"

class ElectricCar(Vehicle):
    def __init__(self):
        super().__init__()
        self.name = "electric car"

    def info(self):
        print(self.name, "has", self.number_of_wheels, "wheels. engine
sound:", self.drive_sound())

v1 = Vehicle()
print(v1.name, "has", v1.number_of_wheels, "wheels. engine sound:",
v1.drive_sound())
# output → common vehicle has 4 wheels. engine sound: vroom vroooommmm

v2 = ElectricCar()
print(v2.name, "has", v2.number_of_wheels, "wheels. engine sound:",
v2.drive_sound())
```

Jika dianalogikan, bisa dibilang kode di atas adalah ekuivalen dengan kode ke-2 berikut:

- Kode setelah perubahan:

```
class Vehicle:
    def __init__(self):
        self.name = "common vehicle"
        self.number_of_wheels = 4

class ElectricCar(Vehicle):
    def __init__(self):
        super().__init__()
        self.name = "electric car"
```

- Ekuivalen dengan kode berikut:

```
class Vehicle:
    def __init__(self):
        self.name = "common vehicle"
        self.number_of_wheels = 4

class ElectricCar(Vehicle):
    def __init__(self):
        # statement berikut terpanggil dari __init__() milik super
class
        self.name = "common vehicle"
        self.number_of_wheels = 4

        # kemudian statement berikut dieksekusi setelahnya
        self.name = "electric car"
```

Sampai sini semoga cukup jelas.

A.41.5. Alternatif cara mengakses super class constructor

Selain menggunakan `super().__init__()` ada cara lain untuk memanggil konstruktor super class, yaitu dengan mengakses method `__init__()` via class secara langsung. Contoh:

```
class Vehicle:
    note = "class to represent a car"

    def __init__(self):
        self.name = "common vehicle"
        self.number_of_wheels = 4

    def drive_sound(self):
        return "vroom vroooommmm"

class ElectricCar(Vehicle):
    def __init__(self):
        Vehicle.__init__(self)
        self.name = "electric car"

    def info(self):
        print(self.name, "has", self.number_of_wheels, "wheels. engine
sound:", self.drive_sound())
```

Statement `Vehicle.__init__(self)` pada kode di atas adalah ekuivalen dengan kode `super().__init__()` pada program sebelumnya.

Teknik pemanggilan constructor via class ini lebih sering digunakan pada class yang memiliki parent class lebih dari satu. Lebih jelasnya akan kita bahas di bawah.

A.41.6. Method overriding

Tidak hanya constructor, method super class juga bisa di-override dengan method baru. Pada kode berikut, method `drive_sound()` di-override dengan isi mengembalikan nilai string berbeda, yang sebelumnya `vroom vroooommmm` kini menjadi `zzzzzzz`.

Coba aplikasikan perubahan berikut lalu run ulang programnya.

```
class Vehicle:
    note = "class to represent a car"

    def __init__(self):
        self.name = "common vehicle"
        self.number_of_wheels = 4

    def drive_sound(self):
        return "vroom vroooommmm"

class ElectricCar(Vehicle):
    def __init__(self):
        super().__init__()
        self.name = "electric car"

    def info(self):
        print(self.name, "has", self.number_of_wheels, "wheels. engine
sound:", self.drive_sound())

    def drive_sound(self):
        return "zzzzzzz"

v1 = Vehicle()
print(v1.name, "has", v1.number_of_wheels, "wheels. engine sound:",
v1.drive_sound())
# output → common vehicle has 4 wheels. engine sound: vroom vroooommmm
```

Bisa dilihat pada statement ke-2, sekarang bunyi mesin berubah menjadi `zzzzzzz`.

Dalam konteks inheritance, ketika di sub class terdapat method dengan nama yang sama persis dengan super class, maka pemanggilan method tersebut dari object yang dibuat via sub class adalah mengarah ke method yang ada di sub class. Konsep ini disebut dengan method overriding.

- Object `v1` dibuat via class `Vehicle`, pengaksesan method `drive_sound()` mengarah ke method milik class tersebut
- Object `v2` dibuat via class `ElectricCar`, pengaksesan method `drive_sound()` (yang dilakukan secara implisit via method `info()`) mengarah ke method milik class tersebut, dan bukan milik super class.

A.41.7. Aturan overriding

Setiap bahasa pemrograman yang mengadopsi OOP, aturan penerapan method overriding berbeda satu sama lain. Di Python sendiri, method dianggap meng-override suatu method atau constructor super class jika namanya adalah dideklarasikan sama persis. Perihal apakah skema parameter-nya diubah, atau return type-nya diubah, itu tidak menjadi syarat wajib overriding.

Agar lebih jelas silakan lihat dan pelajari kode berikut:

```
class Vehicle:
    note = "class to represent a car"

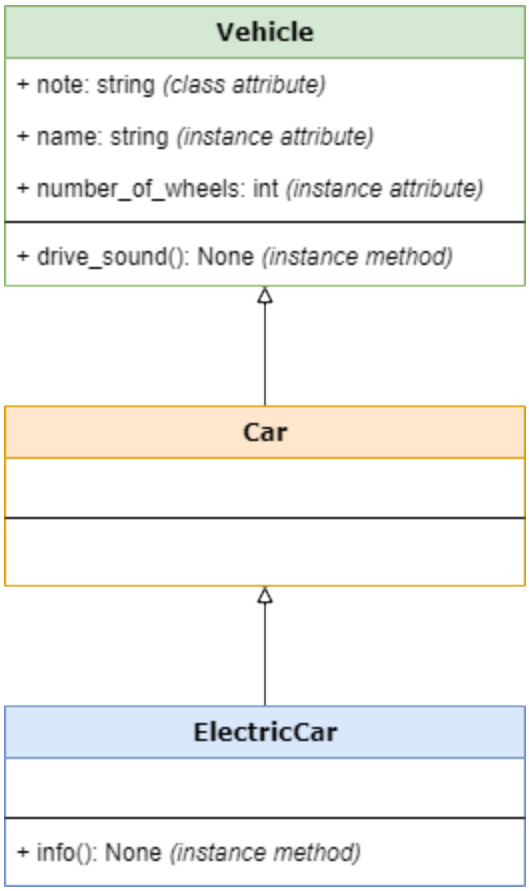
    def __init__(self):
        self.name = "common vehicle"
        self.number_of_wheels = 4
```

Method `drive_sound()` di-override dengan diubah skema parameternya, dari yang tidak memiliki parameter sekarang menjadi memiliki parameter `sound`. Selain itu tipe datanya juga diubah, dari yang sebelumnya string menjadi tuple.

A.41.8. Nested inheritance

Penerapan inheritance tidak hanya terbatas pada dua buah class saja, melainkan bisa lebih. Class bisa diturunkan, kemudian turunannya diturunkan lagi, dan seterusnya.

Contoh pengaplikasiannya bisa dilihat pada kode berikut dimana ada class `Vehicle`, `Car`, dan `ElectricCar`; yang ketiganya menjalin hubungan inheritance dengan hirarki seperti ini:



Source code implementasi:

```
class Vehicle:
    note = "class to represent a car"

    def __init__(self, name = "common vehicle", number_of_wheels = 4):
        self.name = name
        self.number_of_wheels = number_of_wheels

    def drive_sound(self):
        return "vroom vroooommmm"

class Car(Vehicle):
    pass

class ElectricCar(Car):
```

A.41.9. Special name → class attribute `__mro__`

Setiap class memiliki class attribute `__mro__` yang berisi informasi hirarki class itu sendiri. Attribute tersebut bertipe data tuple. Dari nilai balik attribute tersebut gunakan perulangan untuk mengiterasi seluruh elemennya.

```
print("hirarki class ElectricCar:")
for cls in ElectricCar.__mro__:
    print(f"→ {cls.__name__}")

print("hirarki class Car:")
for cls in Car.__mro__:
    print(f"→ {cls.__name__}")

print("hirarki class Vehicle:")
for cls in Vehicle.__mro__:
    print(f"→ {cls.__name__}")

# output ↓
#
# hirarki class ElectricCar:
# → ElectricCar
# → Car
# → Vehicle
# → object
#
# hirarki class Car:
# → Car
# → Vehicle
# → object
#
# hirarki class Vehicle:
# → Vehicle
```

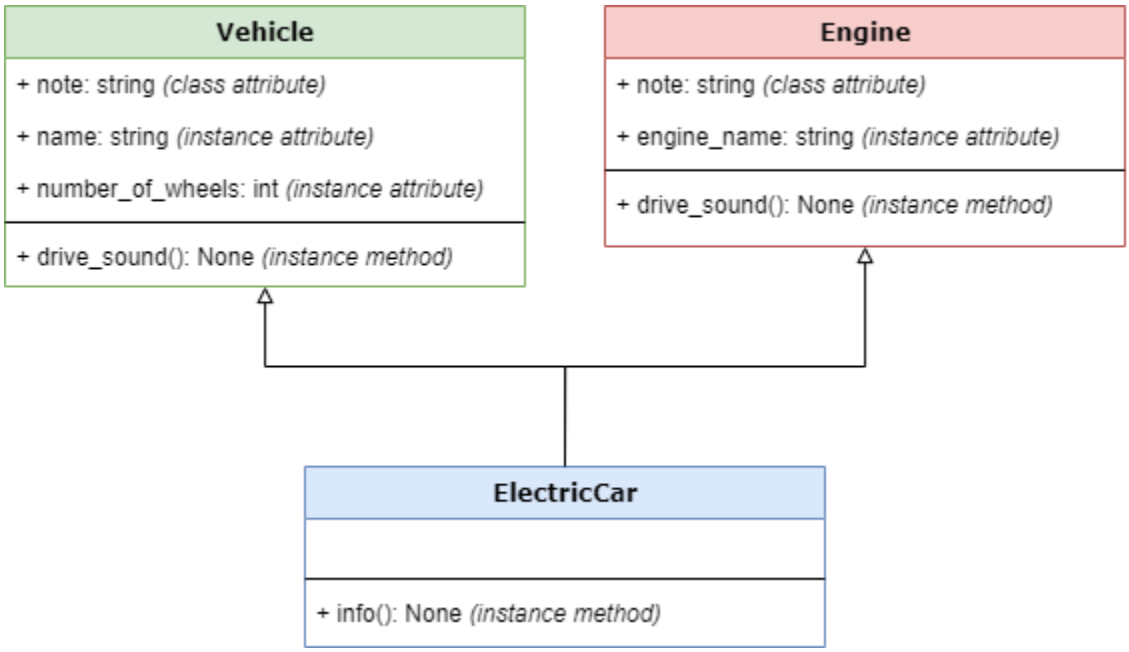
Hirarki paling atas semua class selalu class `object`.

> *MRO sendiri merupakan kependekan dari istilah **Method Resolution Order***

A.41.10. Multiple inheritance

Suatu class tidak dibatasi hanya bisa menjadi sub class dari 1 buah class saja. Bisa jadi ada lebih dari 1 class yang diturunkan dengan level hirarki yang sama.

Sebagai contoh kita buat penerapan inheritance dengan hirarki seperti diagram berikut:



Source code:

```
class Vehicle:
    note = "class to represent a car"

    def __init__(self, name = "common vehicle", number_of_wheels = 4):
        self.name = name
        self.number_of_wheels = number_of_wheels


from typing import Final

ENGINE_ELECTRIC: Final = "electric engine"
ENGINE_PETROL: Final = "petrol engine"
ENGINE_DIESEL: Final = "diesel engine"

class Engine:
    note = "class to represent engine"

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

    def drive_sound(self):
        if self.engine_name == ENGINE_ELECTRIC:
            return "zzzzzzz"
        elif self.engine_name == ENGINE_PETROL:
            return "vroom vroooommmm"
        elif self.engine_name == ENGINE_DIESEL:
            return "VROOM VROOM VROOOOMMM"


class ElectricCar(Vehicle, Engine):

    def __init__(self):
        Vehicle.__init__(self, "electric car", 4)
        Engine.__init__(self, ENGINE_ELECTRIC)

    def info(self):
        print(self.name, "has", self.number_of_wheels, "wheels. engine
sound:", self.drive_sound())
```

Khusus untuk penerapan inheritance dengan lebih dari 1 super class, dianjurkan untuk tidak menggunakan fungsi `super()` untuk mengakses `self` milik parent class, karena `self` disitu mengarah ke object `self` milik super class urutan pertama (yang pada contoh adalah class `Vehicle`).

Dianjurkan untuk memanggil constructor super class secara langsung via `ClassName.__init__()` sesuai kebutuhan. Contohnya bisa dilihat di kode di atas, `Vehicle.__init__()` dan `Engine.__init__()` keduanya diakses pada constructor class `ElectricCar`.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../class-inheritance
```

◉ Chapter relevan lainnya

- OOP ➜ Class & Object
- OOP ➜ Abstract Method
- OOP ➜ Data Class

◉ Referensi

- https://docs.python.org/3/tutorial/classes.html#inheritance
- https://docs.python.org/3/library/stdtypes.html#class.__mro__
- https://en.wikipedia.org/wiki/Unified_Modeling_Language

A.42. Python OOP ➜ Abstract Method

Pada chapter ini kita akan mempelejari tentang apa itu abstract method beserta bagaimana penerapannya pada pemrograman OOP menggunakan Python.

A.42.1. Pengenalan abstract method

Abstract method merupakan jenis method yang dideklarasikan dengan isi tidak melakukan apa-apa, hanya statement `pass`. Nantinya ketika class (dimana method tersebut berada) di-inherit ke sub class lain, maka sub-class harus meng-override method milik super class tersebut.

Abstract method umum digunakan pada situasi dimana ada beberapa class yang memiliki method yang sama, namun isinya berbeda satu sama lain. Agar seragam, maka beberapa class tersebut harus menjadi sub class dari sebuah super class yang sama. Super class sendiri berisi abstract method, dibuat sebagai acuan spesifikasi untuk sub class.

> *Pada pemrograman secara umum, fungsi tanpa isi biasa disebut dengan header function*

Agar lebih mudah untuk memahami konsep dan penerapan abstract method, kita akan mulai pembelajaran dengan praktek tanpa penerapan abstract method terlebih dahulu.

Ok, siapkan sebuah class bernama `Object2D` dengan isi satu buah method

bernama `calculate_area()`. Kemudian class tersebut diturunkan ke dua class baru bernama `Triangle` dan `Circle`.

Class `Triangle` dan `Circle` keduanya memiliki bentuk implementasi `calculate_area()` berbeda satu sama lain karena memang secara aturan rumus perhitungan luas segitiga dan lingkaran adalah berbeda.

Alasan kenapa ada deklarasi method `calculate_area()` di parent class adalah agar sub class `Triange` dan `Circle` memiliki method `calculate_area()` dengan skema seragam.

Berikut adalah source code-nya:

```
class Object2D:
    def calculate_area(self):
        pass

class Triangle(Object2D):
    def __init__(self, b, h):
        self.b = b
        self.h = h

    def calculate_area(self):
        return 1/2 * self.b * self.h

class Circle(Object2D):
    def __init__(self, r):
        self.r = r

    def calculate_area(self):
        return 3.14 * self.r * self.r

obj1 = Triangle(4, 10)
area = obj1.calculate_area()
print(f"area of {type(obj1).__name__}: {area}")
# output → area of Triangle: 20.0
```

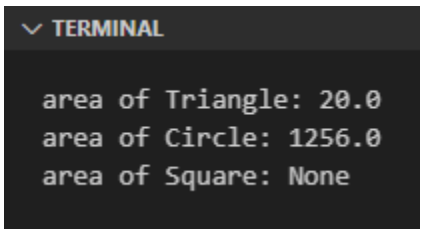
Kode di atas berjalan normal sesuai harapan, namun memiliki kekurangan, yaitu ketika class `Object2D` diturunkan ke suatu class, bisa saja sub class tidak meng-override method `calculate_area()`.

Contoh, pada kode berikut dibuat class baru bernama `Square` yang tidak meng-override method `calculate_area()`:

```
class Square(Object2D):
    def __init__(self, s):
        self.s = s

obj3 = Square(6)
area = obj3.calculate_area()
print(f"area of {type(obj3).__name__}: {area}")
# output → area of Square: None
```

Output:



Kode di atas ketika di-run tidak menghasilkan error, berjalan normal, hanya saja outputnya tidak sesuai harapan karena class `Square` tidak mempunyai method `calculate_area()`. Tanpa adanya method tersebut, maka pemanggilan `calculate_area()` mengarah ke method super class yang isinya mengembalikan nilai `None`.

Di *real life*, ukuran source code yang kita maintain bisa saja berisi ratusan atau bahkan puluhan ribu baris dengan jumlah file sangat banyak. Di case yang seperti itu cukup susah mengecek mana class yang implementasinya sudah sesuai spesifikasi dan mana yang belum, karena saat program dijalankan tidak

ada error atau warning. Untuk mengatasi masalah tersebut, solusinya adalah dengan mengimplementasikan abstract method.

A.42.2. Praktek abstract method

Di Python versi 3.4+, suatu method menjadi abstract method ketika memenuhi kriteria berikut:

- Super class meng-inherit class bawaan Python bernama `ABC` milik module `abc`.
- Method yang dijadikan acuan (yang nantinya wajib di-override) perlu ditambahi decorator `@abstractmethod` milik module `abc`.

> *ABC merupakan kependekan dari Abstract Base Class, sebuah module bawaan Python Standard Library yang berisi banyak property untuk keperluan abstraction.*

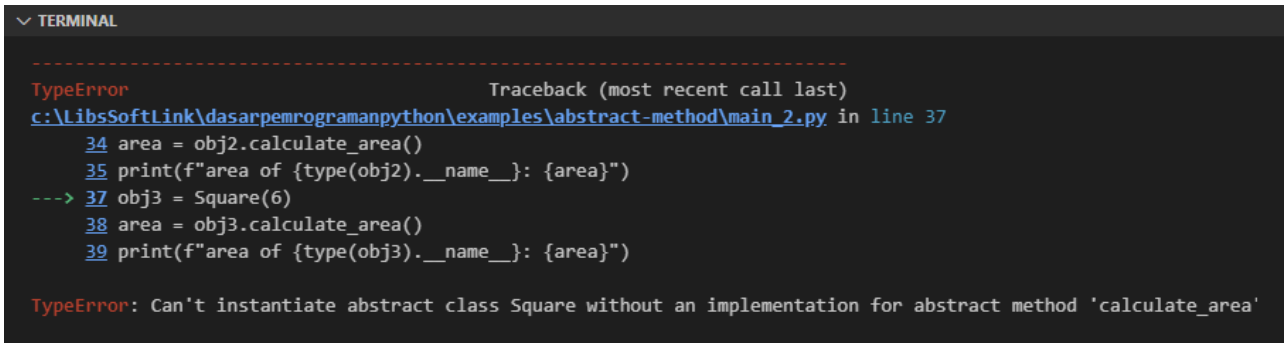
Sekarang, aplikasikan 2 hal di atas ke kode yang telah ditulis. Kurang lebih hasil akhirnya seperti ini. Perbedaannya ada pada deklarasi class `Object2D` dan deklarasi method `calculate_area()`.

```
from abc import ABC, abstractmethod

class Object2D(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

class Triangle(Object2D):
    def __init__(self, b, h):
        self.b = b
```

Selanjutnya, coba jalankan, pasti muncul error karena class `Square` tidak berisi implementasi method `calculate_area()`.

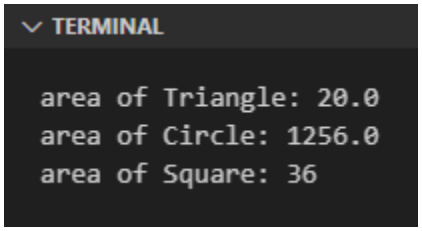


Untuk memperbaiki error, override method `calculate_area()` milik class `Square` agar sesuai spesifikasi.

```
class Square(Object2D):
    def __init__(self, s):
        self.s = s

    def calculate_area(self):
        return self.s * self.s
```

Output program setelah diperbaiki:



Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../abstract-class
```

◉ Chapter relevan lainnya

- OOP ➜ Class & Object
- OOP ➜ Instance Method
- OOP ➜ Constructor
- OOP ➜ Class Method
- OOP ➜ Static Method
- Function ➜ Decorator
- OOP ➜ Class Inheritance

◉ Referensi

- https://docs.python.org/3/library/abc.html
- https://stackoverflow.com/questions/13646245/is-it-possible-to-make-abstract-classes-in-python/13646263#13646263

A.43. Python OOP ➜ DataClass

Data class adalah class yang didesain khusus untuk tujuan penyimpanan data.

Python menyediakan module bernama `dataclasses` dengan isi beberapa API untuk mempermudah pembuatan data class. Pada chapter ini kita akan mempelajarinya.

A.43.1. Pengenalan Data Class

Di bawah ini dicontohkan class bernama `Planet` disiapkan untuk pembuatan data berisi informasi planet. Class ini memiliki 3 buah instance attribute, yaitu `name`, `diameter`, dan `natural_sattelites`. Deklarasi attribute ditulis dalam fungsi `__init__()`.

```
class Planet:
    def __init__(self, name, diameter, natural_sattelites):
        self.name = name
        self.diameter = diameter
        self.natural_sattelites = natural_sattelites

planets = [
    Planet("Mercury", 4879, []),
    Planet(name="Venus", diameter=12104, natural_sattelites=[]),
    Planet(diameter=12742, name="Earth", natural_sattelites=["Moon"]),
]

for p in planets:
    print(f"{p.name} | {p.diameter} km | {len(p.natural_sattelites)}
moons")
```

Contoh di atas menurut penulis cukup jelas dan mudah dipahami karena jika pembaca mengikuti pembahasan secara urut, maka sudah familiar dengan class, attributes, dan object.

Tidak semua class dibuat untuk keperluan operasi terhadap data, ada yang didesain sebagai utilitas, wrapper, atau lainnya. Khusus untuk class yang memang disiapkan untuk pengelolahan data, penulisannya akan lebih praktis menggunakan decorator `@dataclass` milik module `dataclasses`.

Class `Planet` yang telah dibuat di atas, deklarasinya diubah menggunakan dataclass, hasilnya kurang lebih seperti ini:

```
from dataclasses import dataclass

@dataclass
class Planet:
    name: str
    diameter: float
    natural_sattelites: list[str]

planets = [
    Planet("Mercury", 4879, []),
    Planet("Venus", 12104, []),
    Planet("Earth", 12742, ["Moon"]),
]

for p in planets:
    print(f"{p.name} | {p.diameter} km | {len(p.natural_sattelites)}
moons")

# output ↓
#
# Mercury | 4879 km | 0 moons
# Venus | 12104 km | 0 moons
# Earth | 12742 km | 1 moons
```

Cukup tambahkan decorator `@dataclass` saat deklarasi class, lalu tulis attribute seperti penulisan class attribute, tak lupa tentukan tipe data tiap-tiap attribute. Dengan itu maka class otomatis menjadi data class, attribute-nya menjadi instance attribute dan bisa langsung diisi nilainya (via argument constructor) saat pembuatan object.

- Penulisan data class

```
from dataclasses import dataclass

@dataclass
class Planet:
    name: str
    diameter: float
    natural_sattelites: list[str]
```

- Penulisan class biasa

```
class Planet:
    def __init__(self, name, diameter, natural_sattelites):
        self.name = name
        self.diameter = diameter
        self.natural_sattelites = natural_sattelites
```

A.43.2. Attribute mutability

Selayaknya seperti class biasa, instance attribute dataclass adalah *mutable* atau bisa diubah nilainya. Contoh penerapan:

```
mars = Planet("Mars", 4, ["Phobos", "Deimos"])
mars.name = "Red Planet"
mars.diameter = 6779
```

A.43.3. Instance method

Data class bisa memiliki instance method dengan penulisan deklarasi sama persis seperti deklarasi method pada umumnya. Contohnya bisa dilihat di bawah ini, dimana ada data class bernama `Country` berisi 3 buah instance attribute dan satu buah instance method bernama `info()`.

```
@dataclass
class Country:
    name: str
    seasons: list
    number_of_populations: float

    def info(self) -> str:
        return f"{self.name} | {len(self.seasons)} seasons |
{self.number_of_populations} million population"

countries = [
    Country("Indonesia", ["Rainy", "Dry"], 275.5),
    Country("Palestine", ["Winter", "Summer", "Autumn", "Spring"], 5.044),
    Country("Mongolia", ["Winter", "Summer", "Autumn", "Spring"], 3.398),
]

for c in countries:
    print(c.info())

# output ↓
#
# Indonesia | 2 seasons | 275.5 million population
# Palestine | 4 seasons | 5.044 million population
# Mongolia | 4 seasons | 3.398 million population
```

A.43.4. Attribute default value

Attribute data class bisa ditentukan nilai defaultnya menggunakan operator assignment `=` (penulisannya seperti deklarasi variabel). Dengan memberikan nilai default pada attribute, menjadikan parameter konstruktor menjadi opsional. Contoh:

```
@dataclass
class Country:
    name = "Indonesia"
    seasons = ["Rainy", "Dry"]
    number_of_populations = 275.5

    def info(self) -> str:
        return f"{self.name} | {len(self.seasons)} seasons |
{self.number_of_populations} million population"

c = Country()
print(c.info())
# output → Indonesia | 2 seasons | 275.5 million population
```

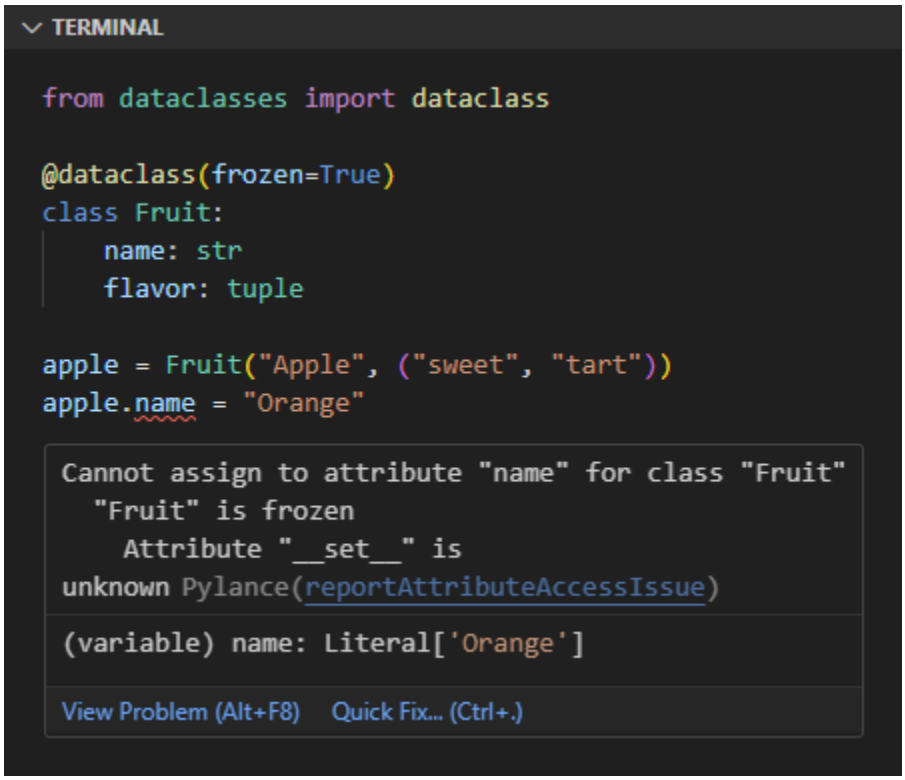
A.43.5. Frozen attribute

Frozen data class adalah data class yang *immutable*, artinya setelah dideklarasikan maka tidak bisa diubah nilai attribute-nya. Cara meng-enable frozen attribute adalah dengan menambahkan `frozen=True` pada decorator `@dataclass`.

```
from dataclasses import dataclass

@dataclass(frozen=True)
```

Kode di atas menghasilkan error karena semua attribute class `Fruit` immutable.



A.43.6. Inheritance

Data class bisa diturunkan seperti umumnya class dengan cara penulisan masih sama. Contoh:

```
from dataclasses import dataclass

@dataclass
class Animal:
    name: str

@dataclass
class Bird(Animal):
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../dataclass
```

◉ Chapter relevan lainnya

- OOP ➜ Class & Object
- OOP ➜ Instance Method
- OOP ➜ Constructor
- OOP ➜ Instance Attribute & Class Attribute
- OOP ➜ Class Inheritance
- Function ➜ Decorator

◉ Referensi

- https://docs.python.org/3/library/dataclasses.html

A.44. Python Duck Typing vs Structural Typing

Pada chapter ini kita akan belajar salah satu konsep yang ada di bahasa pemrograman dinamis, yaitu **duck typing**, beserta perbandingannya dengan **structural typing**

A.44.1. Duck typing

Istilah *duck typing* berasal dari kalimat *If it looks like a duck and quacks like a duck, it's a duck*.

Duck typing adalah konsep yang menjelaskan bahwa compiler/interpreter tidak perlu tau apakah suatu fungsi itu merupakan method, lambda, atau berasal dari class tertentu, atau apapun lainnya; selama fungsi tersebut saat diakses memenuhi kriteria (nama dan skema parameter-nya sama) maka fungsi dianggap valid secara logika.

Mari kita praktekan agar lebih jelas maksudnya apa. Pertama, siapkan sebuah fungsi bernama `do_the_math()`. Tugas fungsi ini sangat sederhana, yaitu menerima parameter `obj`, kemudian lewat variabel tersebut method `calculate_area()` diakses.

```
def do_the_math(obj):
    area = obj.calculate_area()
    print(f"area of {type(obj).__name__}: {area}")
```

Selanjutnya adalah bagian terpenting dari pembelajaran di chapter ini, fungsi yang sudah dibuat akan di test menggunakan beberapa skenario.

◉ Skenario 1: Instance method

Buat class baru untuk operasi perhitungan luas segitiga. Operasi perhitungannya disiapkan di instance method bernama `calculate_area()`. Dari sini, object buatan class ini harusnya bisa dipergunakan sebagai argument fungsi `do_the_math()`.

```
class Triangle:
    def __init__(self, b, h):
        self.b = b
        self.h = h

    def calculate_area(self):
        return 1/2 * self.b * self.h

obj1 = Triangle(4, 10)
do_the_math(obj1)
# output → area of Triangle: 20.0
```

Hasilnya: OK ✅

◉ Skenario 2: Attribute berisi closure

Berikutnya, siapkan class baru lagi dengan attribute bernama `calculate_area`, lalu isi nilai attribute tersebut dengan closure. Disini dicontohkan closure-nya adalah fungsi `number_10()` yang tugasnya mengembalikan nilai numerik `10`.

```
def number_10():
    return 10

class AreaOf2x10:
```

Hasilnya: OK ✅

Fungsi `do_the_math()` berjalan sesuai harapan tanpa melihat tipe data dan struktur dari argument-nya seperti apa. Selama class memiliki property bernama `calculate_area` dan bisa diakses dalam bentuk notasi fungsi, maka bukan masalah.

◉ Skenario 3: Attribute berisi lambda

Pada skenario ini, sebuah class bernama `AreaOfRandomInt` dibuat disertai dengan attribute bernama `calculate_area` yang berisi operasi perkalian angka random yang ditulis dalam syntax lambda.

```
import random

class AreaOfRandomInt:
    def __init__(self) -> None:
        self.calculate_area = lambda : random.randint(0, 10) * 2

obj3 = AreaOfRandomInt()
do_the_math(obj3)
# output → 16
```

Hasilnya: OK ✅

◉ Skenario 4: Class method

Bisa dibilang skenario ini yang paling unik. Buat sebuah class baru berisi class method `calculate_area()`. Lalu jadikan class tersebut sebagai argument pemanggilan fungsi `do_the_math()`. Jadi disini kita tidak menggunakan instance object sama sekali.

```
class NotReallyA2dObject:
    @classmethod
    def calculate_area(cls):
        return "where is the number?"

do_the_math(NotReallyA2dObject)
# output → where is the number?
```

Hasilnya: OK ✅

Fungsi `do_the_math()` tetap bisa menjalankan tugasnya dengan baik, bahkan untuk argument yang bukan instance object sekalipun. Selama argument memiliki fungsi `calculate_area()` maka semuanya aman terkendali.

A.44.2. Structural typing

Structural typing bisa diibaratkan sebagai duck typing tapi versi yang lebih ketat. Structural typing mengharuskan suatu fungsi atau method untuk memilki spesifikasi yang sama persis sesuai yang dideklarasikan. Misalnya ada suatu object berisi method dengan hanya nama fungsi dan skema parameternya saja yang sama dibanding yang dibutuhkan, maka itu tidak cukup dan error pasti muncul.

Cara penerapan structural typing adalah dengan menentukan tipe data parameter secara *explicit*. Mari coba praktekan via kode berikut agar lebih jelas.

Pertama, siapkan sebuah class bernama `Object2D` yang memiliki abstract method `calculate_area()`. Lalu buat juga fungsi `do_the_math()` tapi kali ini argument nya bertipe data `Object2D`.

```
from abc import ABC, abstractmethod

class Object2D(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

def do_the_math(obj: Object2D):
    area = obj.calculate_area()
    print(f"area of {type(obj).__name__}: {area}")
```

Dari sini terlihat bahwa untuk bisa menggunakan fungsi `do_the_math()` data argument harus bertipe `Object2D` atau class turunannya. Inilah bagaimana structural typing diaplikasikan di Python.

Selanjutnya, buat class implementasinya, tak lupa panggil fungsi `do_the_math()`, dan isi argument-nya menggunakan instance object. Jalankan program, hasilnya tidak akan error, karena saat pemanggilan fungsi `do_the_math()` argument yang disisipkan tipe datanya sesuai spesifikasi, yaitu bertipe `Object2D` atau class turunannya.

```
class Triangle(Object2D):
    def __init__(self, b, h):
        self.b = b
        self.h = h

    def calculate_area(self):
        return 1/2 * self.b * self.h

class Circle(Object2D):
    def __init__(self, r):
        self.r = r

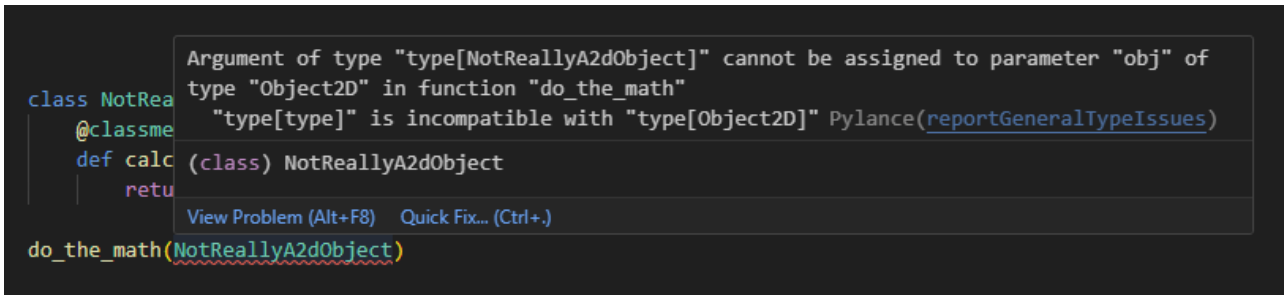
    def calculate_area(self):
        return 3.14 * self.r * self.r
```

Selanjutnya, coba test fungsi `do_the_math()` menggunakan argument berisi data yang bukan bertipe `Object2D` dan juga bukan turunannya.

```
class NotReallyA2dObject:
    @classmethod
    def calculate_area(cls):
        return "where is the number?"

do_the_math(NotReallyA2dObject)
```

Silakan cek di editor masing-masing, pada statement `do_the_math()` terlihat ada warning.



> *Python merupakan bahasa pemrograman dinamis yang dukungan terhadap structural typing tidak terlalu bagus. Keterangan tidak valid pada gambar di atas hanyalah warning, tidak benar-benar error. Kode program sendiri tetap bisa dijalankan.*

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../duck-typing-vs-structural-typing
```

◉ Chapter relevan lainnya

- Tipe Data
- OOP ➔ Abstract Method

◉ Referensi

- https://stackoverflow.com/questions/4205130/what-is-duck-typing
- https://docs.python.org/3/glossary.html#term-duck-typing

A.45. Python Pattern Matching ➜ match

Pada chapter ini, kita akan membahas tentang teknik seleksi kondisi pencocokan pola di Python yang cukup advance untuk mengenali banyak variasi pola, yaitu structural pattern matching.

A.45.1. Pengenalan pattern matching

Pattern matching merupakan teknik pencocokan pola menggunakan kombinasi keyword `match` dan `case`. Penggunaan dasarnya mirip seperti seleksi kondisi menggunakan keyword `if`.

Silakan lihat 2 kode berikut untuk mengetahui perbedaan syntax-nya:

- Kode seleksi kondisi:

```
command = input("Enter command: ")

if command == "greet":
    name = input("Your name: ")
    print("hello", name)

elif command == "find sum of numbers":
    numbers = input("The numbers separated by space: ")
    total = 0
    for str in numbers.split(' '):
        total = total + int(str)
    print("total:", total)

elif command == "lucky number":
```

- Kode yang sama dengan penerapan pattern matching:

```
command = input("Enter command: ")

match command:

    case "greet":
        name = input("Your name: ")
        print("hello", name)

    case "sum_numbers":
        numbers = input("The numbers separated by space: ")
        total = 0
        for str in numbers.split(' '):
            total = total + int(str)
        print("total:", total)

    case "lucky_number":
        import random
        n = random.randint(0, 100)
        print("your lucky number:", n)

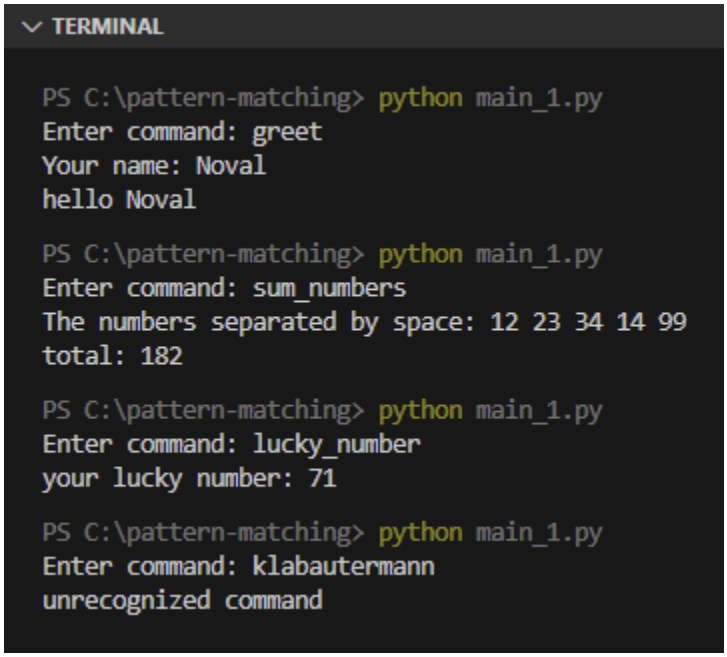
    case _:
        print("unrecognized command")
```

Kode diatas ketika di-run akan meminta sebuah inputan ke user untuk kemudian disimpan di variabel `command`. Dari inputan tersebut kemudian dilakukan sebuah proses sesuai command yang dipilih:

- Jika command adalah `greet`, maka program meminta inputan lagi untuk disimpan di variabel `nama`. Lalu output `hello {nama}` ditampilkan ke layar.
- Jika command adalah `sum_numbers`, inputan angka dengan pembatas karakter spasi harus diisi, kemudian totalnya dimunculkan ke layar.

- Jika command adalah `lucky_number`, maka sebuah angka random ditampilkan.
- Jika command adalah selain 3 pilihan diatas, pesan `unrecognized command` muncul.

Test run program:



Pada penerapannya, keyword `match` ditulis diikuti oleh suatu data yang ingin dicek nilainya, kemudian di dalam block ditentukan kondisi atau pola pencocokan via keyword `case`.

Gunakan kondisi `case _:` untuk menangkap kondisi yang tidak terpenuhi (seperti `else` pada seleksi kondisi `if`). Kondisi yang selalu terpenuhi ini biasa disebut dengan *wildcard*, yang idealnya ditulis di paling akhir.

A.45.2. Pencocokan pola data sequence

Kode yang telah dipraktekan kita akan refactor, dimodifikasi agar inputan bisa

menerima command sekaligus argument (ditandai dengan karakter spasi).

Inputan `command` yang sebelumnya string, kini di-split menggunakan karakter spasi, menghasilkan data list. Data list tersebut dicek menggunakan keyword `match`.

Pada kode berikut, program didesain untuk menerima 3 jenis command:

- Command `greet`, tugasnya masih sama seperti pada program sebelumnya.
- Command `greet thrall`, memunculkan pesan ke `Thrall`
- Command `sum_numbers`, tugasnya masih sama seperti pada program sebelumnya.

Source code:

```
command = input("Enter command: ")
inputs = command.split(' ')

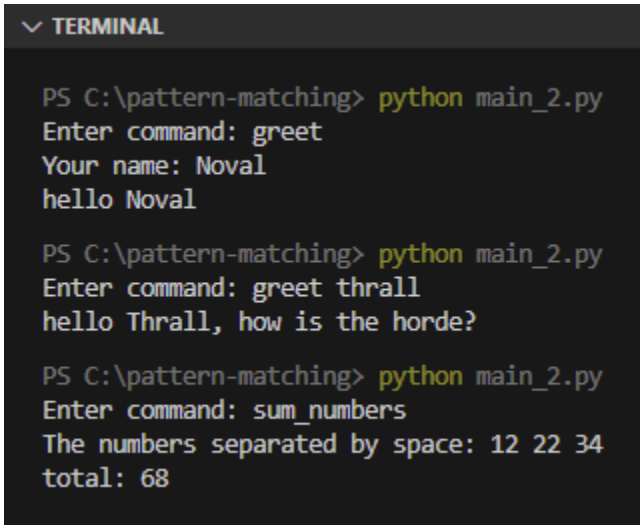
match inputs:

    case ["greet"]:
        name = input("Your name: ")
        print("hello", name)

    case ["greet", "thrall"]:
        print("hello Thrall, how is the horde?")

    case ["sum_numbers"]:
        numbers = input("The numbers separated by space: ")
        total = 0
        for str in numbers.split(' '):
            total = total + int(str)
        print("total:", total)
```

Output program:



Dengan teknik pattern matching, Python bisa mengenali pola list secara cerdas. Kondisi seperti: *ketika lebar list adalah x dengan element adalah y* bisa diterapkan via teknik ini.

◉ Pencocokan sebagian pola

Pencocokan juga bisa dilakukan dengan kondisi sebagian saja, misalnya, *jika data adalah list dengan index pertama berisi* `greet`. Dengan pola seperti ini, nilai element lainnya bisa ditampung dalam suatu variabel.

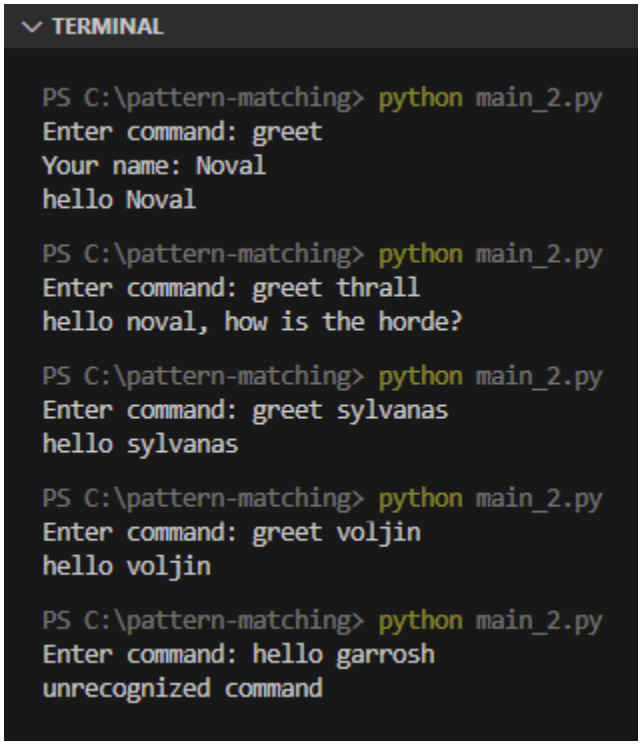
Contoh penerapannya bisa dilihat di kode berikut:

```
command = input("Enter command: ")

match command.split(' '):

    case ["greet"]:
        name = input("Your name: ")
        print("hello", name)
```

Output program:



Terlihat di output inputan `greet sylvanas` dan `greet voljin` cocok dengan pola `case ["greet", name]`.

◉ Pencocokan pola `*args`

Teknik pattern matching mendukung pola dimana pencocokan dilakukan untuk sebagian pola dan sisanya disimpan di satu variabel saja, misalnya `*args`. Ini merupakan kombinasi penerapan patern matching dan teknik packing data sequence.

Pada kode berikut, ada 2 buah kondisi pola `sum_numbers`, yang pertama ketika inputan hanya `sum_numbers` saja, dan yang kedua ketika inputan `sum_numbers` diikuti oleh data lain (yang nilainya diharapkan berbentuk numerik karena akan dihitung totalnya).

```
command = input("Enter command: ")

match command.split(' '):

    case ["greet"]:
        name = input("Your name: ")
        print("hello", name)

    case ["greet", "thrall"]:
        print("hello noval, how is the horde?")

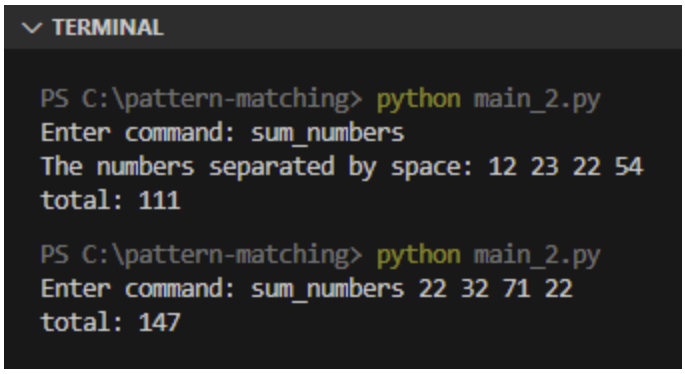
    case ["greet", name]:
        print("hello", name)

    case ["sum_numbers"]:
        numbers = input("The numbers separated by space: ")
        total = 0
        for str in numbers.split(' '):
            total = total + int(str)
        print("total:", total)

    case ["sum_numbers", *args]:
        total = 0
        for str in args:
            total = total + int(str)
        print("total:", total)

    case _:
        print("unrecognized command")
```

Output program:



Penjelasan eksekusi program:

- Eksekusi ke-1: inputan adalah `sum_numbers`, cocok dengan kondisi `case ["sum_numbers"]`.
- Eksekusi ke-2: inputan adalah `sum_numbers 22 32 71 22`, cocok dengan kondisi `case ["sum_numbers", *args]`. Nilai numerik disitu disimpan pada variabel `*args`.

> - *Lebih detailnya mengenai `*args` dibahas pada chapter Function ➜ Args & Kwargs*
> - *Lebih detailnya mengenai teknik packing dibahas pada chapter Pack Unpack ➜ Tuple, List, Set, Dict*

◉ Pencocokan pola wildcard

Kondisi `case _` bisa difungsikan sebagai `else` pada pattern matching. Semua kondisi yang tidak terpenuhi masuk ke block tersebut. Alternatif lainnya adalah menggunakan `case namaVariabel`.

Pada kode berikut dua buah kondisi baru ditambahkan:

- Kondisi `case [other]`, akan cocok jika inputan adalah list dengan lebar 1 item.

Contoh inputan yang memenuhi kriteria ini: `hello`, `makan`, `kesedihan`, dan kata-kata tanpa spasi lainnya. Hal ini karena inputan di-split menggunakan spasi (` `) dengan hasil pasti berupa list dengan lebar 1 elemen. Isi dari variabel `other` adalah inputan string.

- Kondisi `case other`, cocok dengan pola inputan apapun. Semua inputan ditampung di variabel `other` dalam bentuk list.

```
command = input("Enter command: ")

match command.split(' '):

    case ["greet"]:
        name = input("Your name: ")
        print("hello", name)

    case ["greet", "thrall"]:
        print("hello noval, how is the horde?")

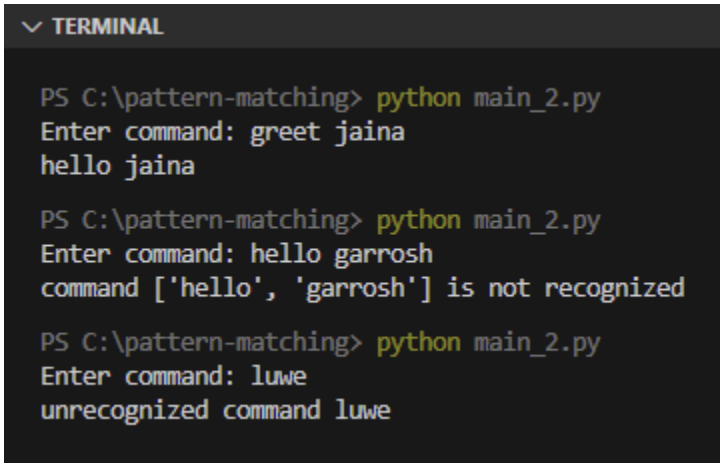
    case ["greet", name]:
        print("hello", name)

    case ["sum_numbers"]:
        numbers = input("The numbers separated by space: ")
        total = 0
        for str in numbers.split(' '):
            total = total + int(str)
        print("total:", total)

    case ["sum_numbers", *args]:
        total = 0
        for str in args:
            total = total + int(str)
        print("total:", total)

    case [other]:
```

Output program:



A.45.3. Pencocokan pola + seleksi kondisi

◉ Pencocokan pola + logika OR

Kondisi `or` bisa digunakan pada 1 block `case`, caranya dengan menuliskan inputan diapit tanda `()` dan didalam kondisi ditulis menggunakan pembatas karakter pipe `|`.

Pada kode berikut, ada 3 buah kondisi baru ditambahkan:

- Kondisi `case ["greet", name, ("morning" | "afternoon" | "evening") as t]`

  Hanya akan terpenuhi jika inputan pertama adalah `greet` dan inputan ke-3 adalah `morning` atau `afternoon` atau `evening`. Inputan ke-3 ditampung ke variabel bernama `t` (syntax `as` digunakan disitu).

- Kondisi `case ["greet", name] | ["hello", name]`

Hanya akan terpenuhi jika inputan adalah 2 kata dan kata pertama adalah `greet` atau `hello`.

- Kondisi `case ([other] | other) as o`

  Hanya akan terpenuhi jika inputan adalah list dengan lebar 1 element atau inputan selain yang dikenali oleh kondisi-kondisi sebelumnya (wildcard).

  Catatan: kondisi ini redundan, harusnya `case other:` saja sudah cukup untuk difungsikan sebagai `else`. Namun pada contoh ini tetap ditulis hanya untuk menunjukan format penulisan kondisi pola dengan logika OR.

Source code setelah dimodifikasi:

```
command = input("Enter command: ")

match command.split(' '):

    case ["greet"]:
        name = input("Your name: ")
        print("hello", name)

    case ["greet", "thrall"]:
        print("hello noval, how is the horde?")

    case ["greet", name, ("morning" | "afternoon" | "evening") as t]:
        print("hello", name, "good", t)

    case ["greet", name] | ["hello", name]:
        print("hello", name)

    case ([other] | other) as o:
        print(f"command {o} is not recognized")
```

◉ Pencocokan pola + keyword `if`

Keyword `if` boleh dituliskan setelah kondisi `case`. Penambahannya menjadikan pencocokan pola diikuti oleh seleksi kondisi.

Pada kode berikut, kondisi `case ["greet", name] if name == "thrall"` ditambahkan. Kondisi tersebut hanya akan terpenuhi jika:

- Inputan berisi 2 buah kata dengan pembatas spasi
- Kata pertama adalah `greet`
- Kata kedua adalah `thrall` (pengecekannya via seleksi kondisi `if`)

Source code:

```
command = input("Enter command: ")

match command.split(' '):

    case ["greet"]:
        name = input("Your name: ")
        print("hello", name)

    case ["greet", name] if name == "thrall":
        print("hello noval, how is the horde?")

    case ["greet", name, ("morning" | "afternoon" | "evening") as t]:
        print("hello", name, "good", t)

    case other:
        print(f"command {other} is not recognized")
```

A.45.4. Pencocokan pola tipe data

lainnya

Tidak hanya tipe data string dan list saja yang bisa digunakan pada pattern matching, melainkan banyak sekali tipe data lainnya juga bisa digunakan. Termasuk diantaranya adalah tipe data sequence seperti `tuple` dan `dictionary`, dan juga tipe data dari custom class.

Pada contoh berikut, ada 3 buah program dibuat yang kesemuanya berisi flow dan pattern matching yang sama persis. Perbedaannya:

- Pada program pertama, `tuple` digunakan pada pattern matching
- Program ke-2 menggunakan tipe data `dict`
- Data custom class digunakan pada program ke-3

Di dalam program, data list dibuat berisi element bertipe `tuple` atau `dict` atau object class yang menyimpan data koordinat `x` dan `y`.

Pattern matching digunakan pada setiap elemen list untuk dicek posisi `x` dan `y`-nya, apakah di tengah, di kanan, di atas, atau di posisi lainnya.

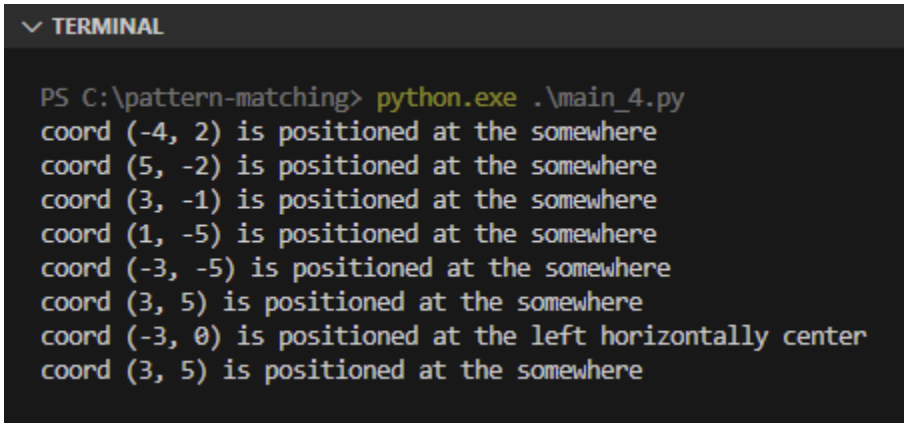
◉ Pencocokan pola data tuple

Source code:

```
import random

def coords_generator(n = 5):
    coords = []
    for _ in range(0, n):
        coords.append((random.randint(-5, 5), random.randint(-5, 5)))
    return coords
```

Output program:



◉ Pencocokan pola data dictionary

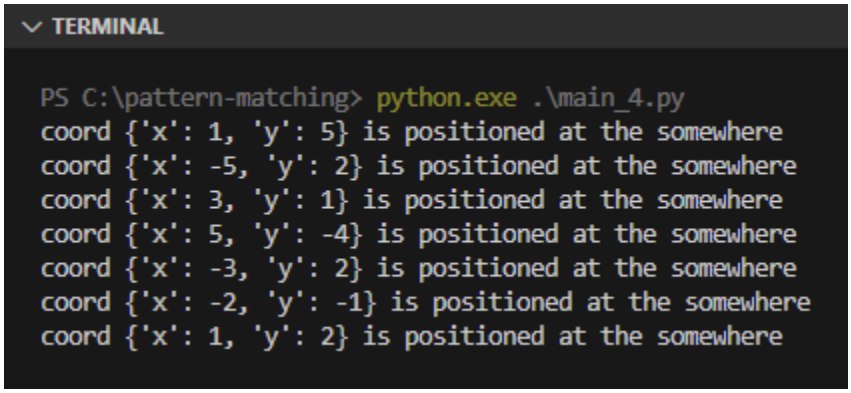
Source code:

```
import random

def coords_generator(n = 5):
    coords = []
    for _ in range(0, n):
        coords.append({
            "x": random.randint(-5, 5),
            "y": random.randint(-5, 5)
        })
    return coords

data = coords_generator(8)
for c in data:
    match c:
        case {"x": 0, "y": 0}:
            print(f"coord {c} is positioned at the center")
        case {"x": x, "y": 0} if x < 0:
            print(f"coord {c} is positioned at the left horizontally
center")
        case {"x": x, "y": 0} if x > 0:
```

Output program:



◉ Pencocokan pola data object class

Source code:

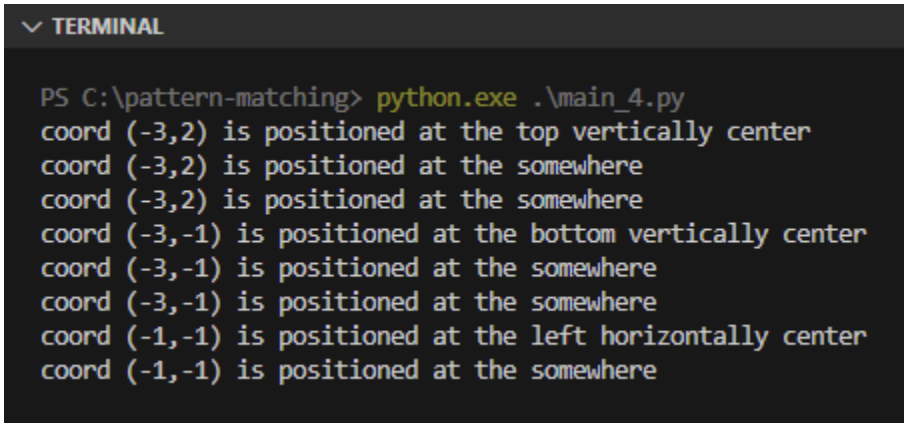
```
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def info(self):
        return f"({x},{y})"

import random

def coords_generator(n = 5):
    coords = []
    for _ in range(0, n):
        coords.append(Point(random.randint(-5, 5), random.randint(-5, 5)))
    return coords

data = coords_generator(8)
for c in data:
    match c:
        case Point(x=0, y=0):
            print(f"coord {c.info()} is positioned at the center")
        case Point(x=x, y=0) if x < 0:
```

Output program:



Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../pattern-matching
```

◉ Chapter relevan lainnya

- Seleksi kondisi ➜ if, elif, else

◉ Referensi

- https://peps.python.org/pep-0622/
- https://peps.python.org/pep-0636/

A.46. Python Error & Exception

Secara teknis Python interpreter mengenal dua jenis error, yaitu *syntax error* dan *exception*. Sebenarnya ada lagi satu jenis error lainnya, yaitu yang munculnya hanya di level linter (di editor) namun tidak membuat eksekusi program menjadi gagal.

Pada chapter ini kita akan membahas tentang topik tersebut.

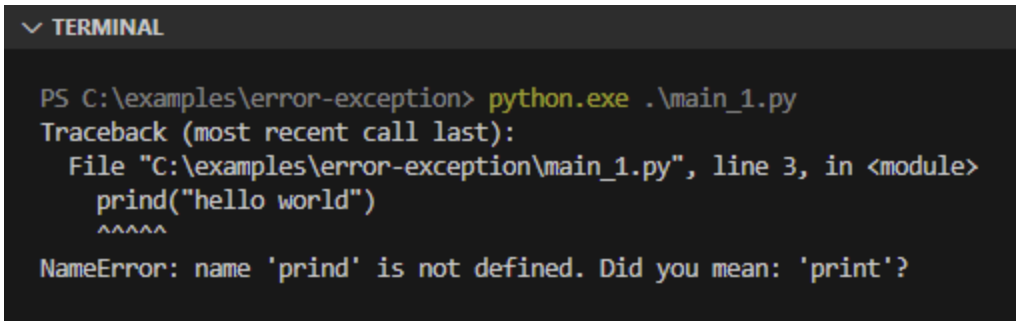
A.46.1. Syntax error

Syntax error adalah salah satu jenis error yang ada di Python, yang jika muncul maka bisa dipastikan eksekusi program adalah gagal atau terhenti. Syntax error disebabkan oleh kesalahan penulisan.

Misalnya ada typo pada pemanggilan fungsi print yang tidak sengaja tertulis sebagai `prind()`, sedangkan di source code sendiri tidak ada fungsi lain yang dideklarasikan dengan nama itu, maka pada situasi seperti ini terjadi syntax error.

```
prind("hello world")
```

Output program:



Satu-satunya solusi untuk mengatasi syntax error adalah dengan memperbaiki kode, memastikan semua penulisannya benar sesuai aturan di Python.

A.46.2. Exception

Exception adalah jenis error yang muncul saat *runtime* (saat program dijalankan). Berbeda dengan syntax error yang munculnya saat proses eksekusi program (sebelum program benar-benar running).

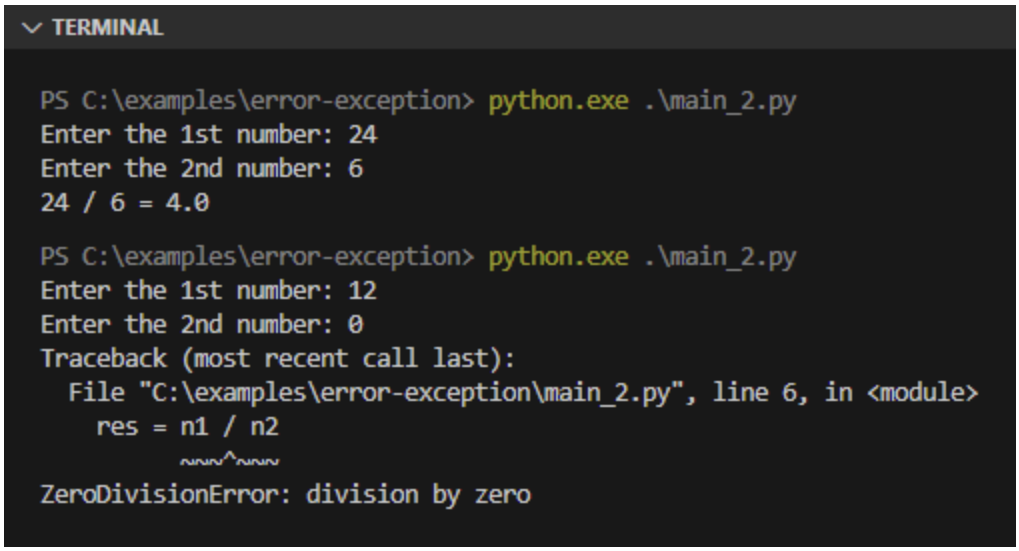
Salah satu exception yang umumnya ada di bahasa pemrograman adalah **zero division error**. Error ini muncul ketika ada operasi aritmatika pembagian suatu bilangan numerik terhadap bilangan `0`.

Contoh kasus exception:

```
n1 = int(input("Enter the 1st number: "))
n2 = int(input("Enter the 2nd number: "))

res = n1 / n2
print(f"{n1} / {n2} = {res}")
```

Output program:

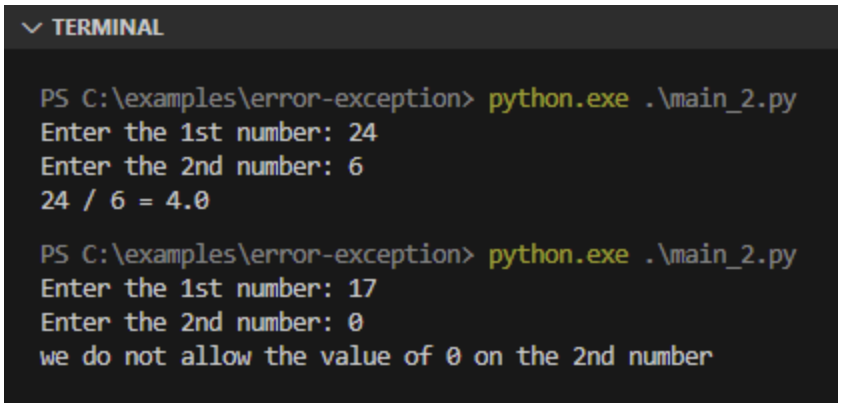


Exception bisa diantisipasi dengan menambahkan validasi, misalnya untuk kasus di atas bisa dengan ditambahkan seleksi kondisi pengecekan nilai `n2`. Jika nilainya adalah `0`, maka program dihentikan dan pesan peringatan dimunculkan.

```
n1 = int(input("Enter the 1st number: "))
n2 = int(input("Enter the 2nd number: "))

if n2 == 0:
    print("we do not allow the value of 0 on the 2nd number")
else:
    res = n1 / n2
    print(f"{n1} / {n2} = {res}")
```

Output program:



Alternatif solusi lainnya untuk mengatasi exception adalah dengan pengaplikasian kombinasi keyword `try` dan `catch`. Lebih detailnya akan dibahas di chapter berikutnya, di chapter Exception Handling (try, catch, finally).

A.46.3. Throw exception

Di atas kita belajar salah satu cara antisipasi exception, yaitu dengan penambahan validasi sesuai kebutuhan, jika kondisi berpotensi menghasilkan exception maka pesan custom error dimunculkan.

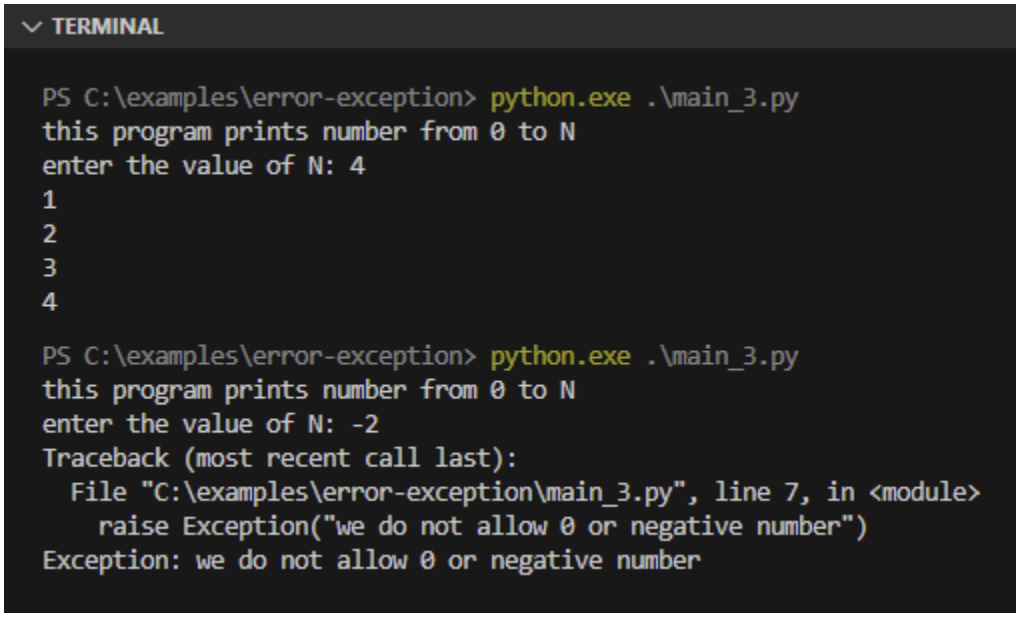
Selanjutnya, kita akan belajar cara untuk dengan sengaja membuat atau melempar exception (istilah umumnya *throwing exception*). Dengan melempar exception, program akan terhenti secara paksa.

Pada program berikut ini, sejumlah baris angka dimunculkan sesuai inputan. Jika inputannya `0` atau bilangan negatif maka exception dilempar, membuat program terhenti.

```
print("this program prints number from 0 to N")
n = int(input("enter the value of N: "))

if n <= 0:
```

Outputnya:



Cara membuat exception adalah dengan menggunakan keyword `raise` diikuti dengan pemanggilan class `Exception()` yang argument-nya diisi dengan custom error.

Pada contoh di atas, exception dimunculkan dengan pesan error `we do not allow 0 or negative number`.

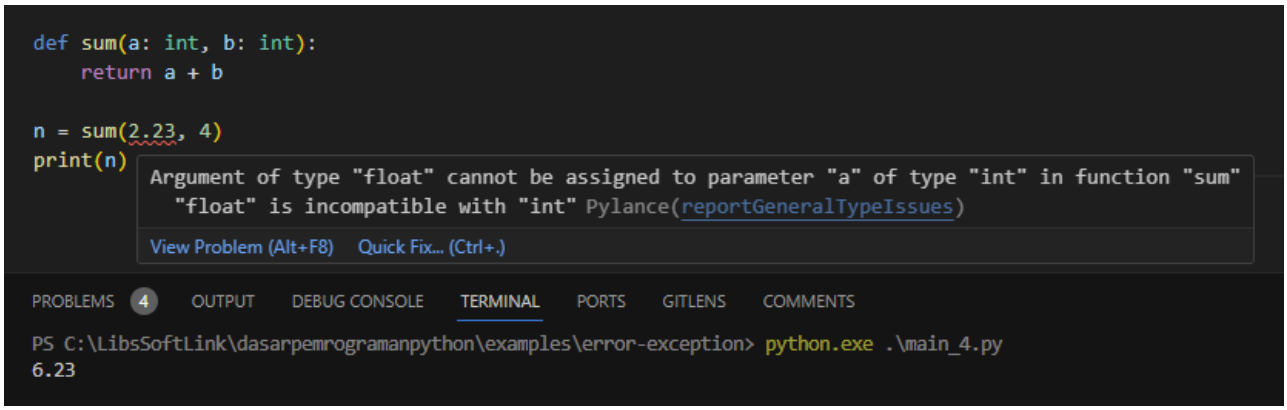
A.46.4. Linter error / warning

Linter adalah suatu program utilitas yang berguna untuk melakukan pengecekan kualitas kode saat pengembangan (penulisan kode). Linter akan memunculkan error atau warning jika ditemukan pada beberapa bagian kode yang ditulis adalah kurang baik.

Di Python, jika pembaca menggunakan VSCode editor dan sudah meng-install extension Python, linter akan otomatis bekerja saat menulis kode.

Linter error adalah warning yang muncul di editor saat kode tidak sesuai baik secara *syntactic* maupun secara *semantic*. Error yang muncul karena alasan semantik tidak akan membuat program terhenti atau gagal running. Program tetap bisa jalan normal saat di-run.

Salah satu contoh linter error adalah ketika ada suatu fungsi yang saat pemanggilannya diisi oleh tipe data dengan tipe yang tidak sesuai dibanding dengan yang sudah dideklarasikan. Pada situasi seperti ini error muncul di editor, ada highlight merah di situ.



Meskipun tidak membuat program terhenti saat running, ada baiknya untuk selalu menulis kode dengan baik dan benar sesuai aturan.

Catatan chapter 🗒️

◉ Source code praktik

`github.com/novalagung/dasarpemrogramanpython-example/../error-exception`

◉ Chapter relevan lainnya

- Exception Handling ➜ try, except, else, finally

◉ Referensi

- https://docs.python.org/3/tutorial/errors.html

A.47. Python Exception Handling (try, except, else, finally)

Chapter ini membahas tentang penanganan exception via keyword `try`, `except`, `else`, dan `finally`. Metode exception handler ini sangat efektif karena tidak membutuhkan validasi error secara manual satu per satu menggunakan seleksi kondisi.

> *Pembahasan tentang apa itu exception sendiri ada di chapter Error & Exception*

A.47.1. Keyword `try` & `except`

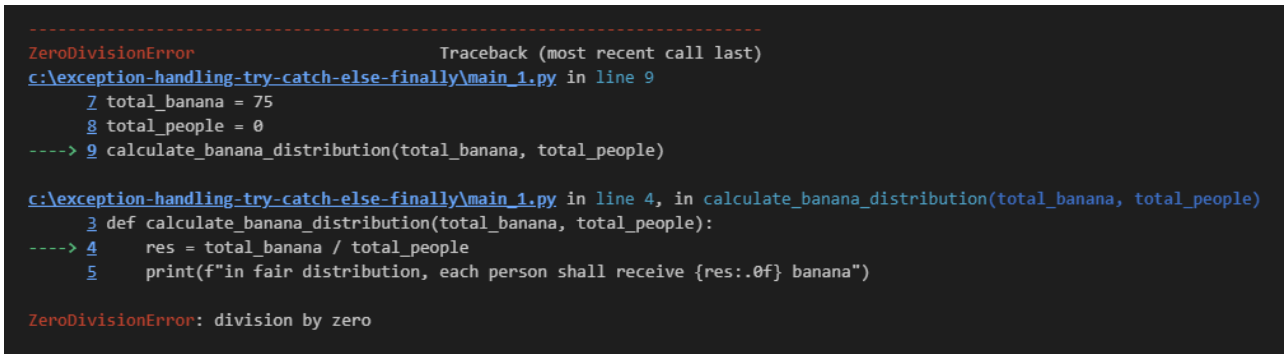
Kita mulai pembelajaran dengan sebuah kode sederhana untuk menghitung pembagian pisang ke warga. Hasil operasi pembagian tersebut kemudian di-print.

```
def calculate_banana_distribution(total_banana, total_people):
    res = total_banana / total_people
    print(f"in fair distribution, each person shall receive {res:.0f}
banana")

total_banana = 75
total_people = 3
calculate_banana_distribution(total_banana, total_people)
# output → in fair distribution, each person shall receive 25 banana
```

Sekilas tidak ada yang aneh dari kode di atas, kode akan berjalan normal ketika di-run.

Sekarang coba set nilai `total_people` menjadi `0` kemudian re-run. Pasti muncul exception karena ada operasi pembagian numerik terhadap nilai `0`.



Salah satu solusi penyelesaian error di atas bisa dengan penambahan seleksi kondisi. Alternatif solusi lainnya adalah dengan mengaplikasikan kombinasi keyword `try` dan `except`. Caranya:

- Tempatkan statement (yang berpotensi memunculkan exception) ke dalam block `try`
- Tambahkan block `except` dimana isinya adalah hanlder ketika exception muncul

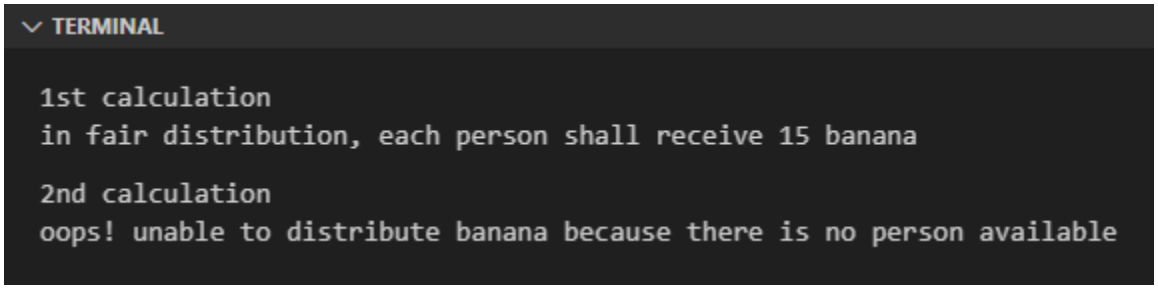
Contoh penerapan:

```
try:
    print("1st calculation")
    calculate_banana_distribution(75, 5)

    print("2nd calculation")
    calculate_banana_distribution(25, 0)

    print("3rd calculation")
```

Output program:



Cara kerja `try` dan `except` adalah Python akan mencoba untuk mengeksekusi statement dalam block `try` terlebih dahulu. Kemudian jika ada exception, maka program dihentikan dan block `except` dijalankan.

Kurang lebih alur eksekusi program di atas adalah seperti ini:

1. Statement pemanggilan `calculate_banana_distribution()` pertama tidak menghasilkan exception. Hasilnya normal.
2. Statement pemanggilan yang ke-2 menghasilkan exception.
    i. Kemudian eksekusi statement dalam block `try` dihentikan secara paksa.
    ii. Kemudian block `except` dieksekusi.
3. Statement `calculate_banana_distribution()` ke-3 tidak akan dijalankan.

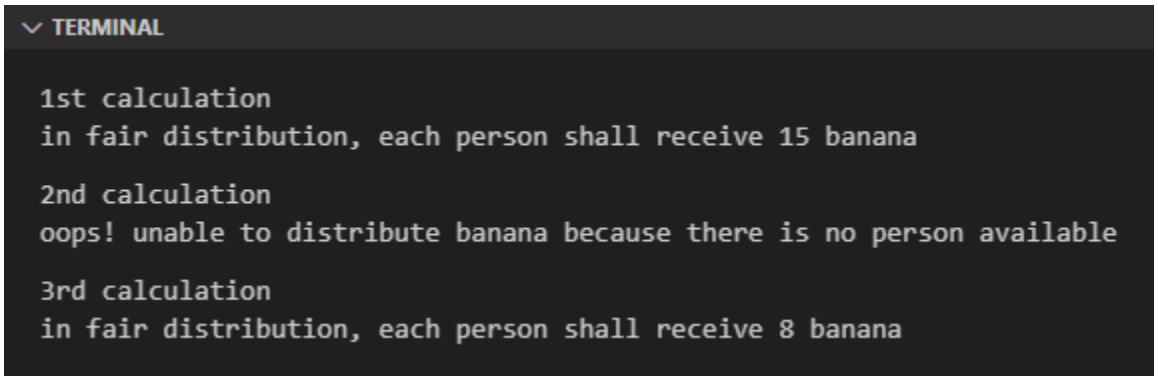
Bisa dilihat di output, pesan `oops! unable to distribute banana because there is no person available` muncul, menandai akhir eksekusi block `try` & `except`.

Terkait penempatan block `try` & `except` sendiri bisa di bagian dimana fungsi dipanggil, atau di dalam fungsi itu sendiri. Contoh:

```
def calculate_banana_distribution(total_banana, total_people):
    try:
```

Kode di atas menghasilkan output yang berbeda dibanding sebelumnya. Karena exception handler-nya ada di dalam fungsi `calculate_banana_distribution()`, maka eksekusi `try` & `catch` hanya terhenti di dalam fungsi tersebut saja. Di bagian pemanggilan fungsi sendiri, eksekusinya tetap berlanjut. Efeknya statement pemanggilan fungsi `calculate_banana_distribution()` ke-3 tetap berjalan.

Output program:



Silakan gunakan block `try` & `except` sesuai kebutuhan, tempatkan di bagian kode yang memang dirasa paling pas.

A.47.2. Explicit exception handler

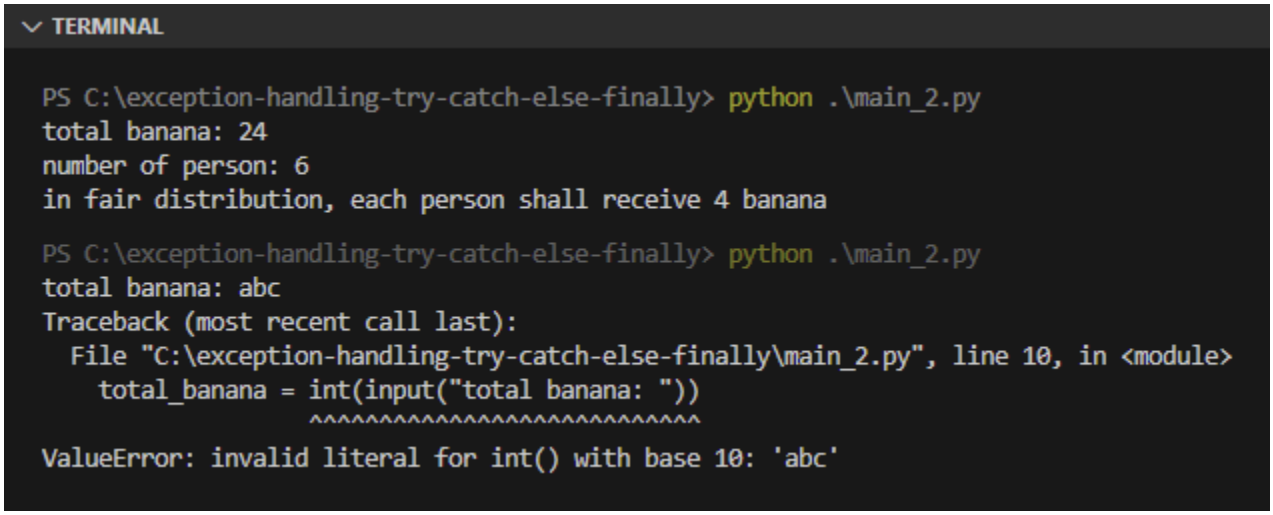
Suatu exception bisa ditangkap secara spesifik dengan menuliskan varian exception-nya setelah keyword `except`. Contoh penerapannya bisa di lihat pada kode berikut, dimana exception `ZeroDivisionError` perlu ditangkap ketika muncul.

```
try:
    total_banana = int(input("total banana: "))
    total_people = int(input("number of person: "))
    res = total_banana / total_people
```

Kode akan di-test dengan dijalankan dua kali dengan skenario berikut:

- Eksekusi ke-1: nilai pembagi di-set `6`
- Eksekusi ke-2: salah satu inputan di-set huruf, efeknya muncul exception `ValueError`

Output program:

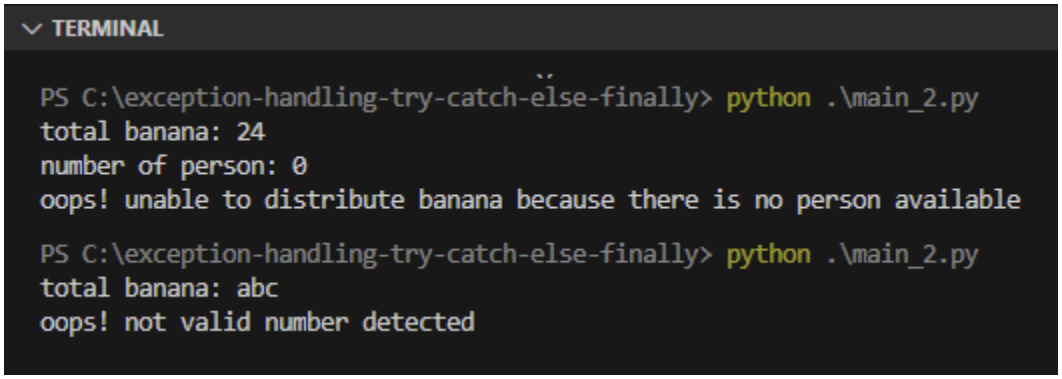


Bisa dilihat di eksekusi pertama, block exception handler berjalan sesuai ekspektasi. Namun pada eksekusi ke-2 ketika inputan diisi dengan huruf, ada exception baru muncul dan tidak tertangkap. Hal ini karena di kode ditentukan secara eksplisit hanya exception `ZeroDivisionError` yang ditangkap.

Untuk menangkap exception lain caranya bisa dengan menambahkan block `except` baru. Pada kode berikut ada 2 exception yang akan ditangkap, yang keduanya memunculkan pesan berbeda.

```
try:
    total_banana = int(input("total banana: "))
    total_people = int(input("number of person: "))
    res = total_banana / total_people
    print(f"in fair distribution, each person shall receive {res:.0f}
```

Output program:



Sampai sini semoga cukup jelas.

◉ Menangkap banyak exception sekaligus

Bagaimana jika 2 exception yang ditangkap didesain untuk memunculkan pesan sama? Maka gunakan notasi penulisan berikut. Tulis saja exceptions yang dingin di tangkap sebagai element tuple.

```
try:
    total_banana = int(input("total banana: "))
    total_people = int(input("number of person: "))
    res = total_banana / total_people
    print(f"in fair distribution, each person shall receive {res:.0f}
banana")
except (ValueError, ZeroDivisionError):
    print("oops! something wrong")
```

Pada kode di atas, ketika exception `ValueError` atau `ZeroDivisionError` muncul, maka pesan `oops! something wrong` ditampilkan.

◉ Menangkap semua exception

Jika exception yang ingin ditangkap adalah semua varian exception, maka

cukup tulis `except:` saja, atau gunakan class `except Exception:` disitu. Contoh:

- Menggunakan `except:`

```
try:
    total_banana = int(input("total banana: "))
    total_people = int(input("number of person: "))
    res = total_banana / total_people
    print(f"in fair distribution, each person shall receive {res:.0f}
banana")
except:
    print("oops! something wrong")
```

- Menggunakan `except Exception:`

```
try:
    total_banana = int(input("total banana: "))
    total_people = int(input("number of person: "))
    res = total_banana / total_people
    print(f"in fair distribution, each person shall receive {res:.0f}
banana")
except Exception:
    print("oops! something wrong")
```

Tipe data `Exception` sendiri merupakan class bawaan Python Standard Library yang dimana di-inherit oleh semua varian exception. Contohnya seperti `ValueError` dan `ZeroDivisionError` keduanya merupakan sub class dari class `Exception`.

◉ Memunculkan pesan exception

Biasanya dalam penangkapan exception, pesan exception aslinya juga perlu dimunculkan mungkin untuk keperluan debugging. Hal seperti ini bisa

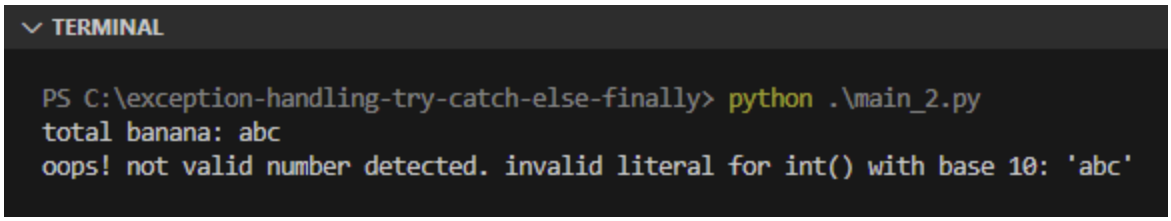
dilakukan dengan menambahkan keyword `as` setelah statement `except` kemudian diikuti variabel penampung data exception.

Penulisannya bisa dilihat di kode berikut:

```
try:
    total_banana = int(input("total banana: "))
    total_people = int(input("number of person: "))
    res = total_banana / total_people
    print(f"in fair distribution, each person shall receive {res:.0f} banana")
except ValueError as err:
    print(f"oops! not valid number detected. {err}")
except ZeroDivisionError as err:
    print(f"oops! unable to distribute banana because there is no person available. {err}")
except Exception as err:
    print(f"oops! something wrong. {err}")
```

Kode di atas akan menangkap 3 macam exception:

- Ketika ada `ValueError`, maka dimunculkan pesan `oops! not valid number detected` diikut dengan pesan error aslinya bawaan exception `ValueError`.
- Ketika ada `ZeroDivisionError`, maka dimunculkan pesan `oops! unable to distribute banana because there is no person available` diikut dengan pesan error aslinya bawaan exception `ZeroDivisionError`.
- Ketika ada exception apapun itu (selain dua di atas), maka dimunculkan pesan `oops! something wrong` diikut dengan pesan error aslinya bawaan exception.



Bisa dilihat pada gambar di atas, error bawaan exception dimunculkan juga setelah custom message yang kita buat.

◉ Alternatif penulisan exception

Dalam operasi penangkapan lebih dari 1 varian exception, penulisannya bisa cukup dalam satu block `except` saja tetapi didalamnya perlu ada seleksi kondisi untuk mengecek spesifik exception yang muncul yang mana. Contoh:

```
try:
    total_banana = int(input("total banana: "))
    total_people = int(input("number of person: "))
    res = total_banana / total_people
    print(f"in fair distribution, each person shall receive {res:.0f}
banana")

except Exception as err:
    if err == ValueError:
        print(f"oops! not valid number detected. {err}")
    elif err == ZeroDivisionError:
        print(f"oops! unable to distribute banana because there is no
person available. {err}")
    else:
        print(f"oops! something wrong. {err}")
```

A.47.3. Keyword `try`, `except` & `else`

Keyword `else` bisa dikombinasikan dengan `try` dan `except`. Block `else`

tersebut hanya akan dieksekusi ketika tidak terjadi exception.

Pada praktek yang sudah ditulis, statement `print(f"in fair distribution...")` yang merupakan output kalkulasi, ideal untuk ditulis pada block `else` karena statement tersebut hanya muncul ketika tidak ada exception.

Kode sebelumnya jika di-refactor jadi seperti ini:

```
try:
    total_banana = int(input("total banana: "))
    total_people = int(input("number of person: "))
    res = total_banana / total_people
except ValueError as err:
    print(f"oops! not valid number detected. {err}")
except ZeroDivisionError as err:
    print(f"oops! unable to distribute banana because there is no person
available. {err}")
except Exception as err:
    print(f"oops! something wrong. {err}")
else:
    print(f"in fair distribution, each person shall receive {res:.0f}
banana")
```

Penjelasan alur program di atas:

1. Program diawali dengan eksekusi statement dalam block `try`
2. Jika terjadi exception `ValueError`, maka dimunculkan pesan error
3. Jika terjadi exception `ZeroDivisionError`, maka dimunculkan pesan error
4. Jika terjadi exception lainnya, maka dimunculkan pesan error
5. Jika tidak terjadi exception sama sekali, maka block `else` dijalankan

Block `else` mengenali semua variabel yang dideklarasikan di block `try`. Oleh karena itu variabel `res` bisa langsung di-print di block tersebut.

A.47.4. Keyword `try`, `except` & `finally`

Keyword `finally` adalah keyword yang berguna untuk menandai bahwa eksekusi suatu block `try` & `except` telah selesai. Block `finally` hanya dieksekusi ketika deretan block selesai, tanpa mengecek apakah ada exception atau tidak.

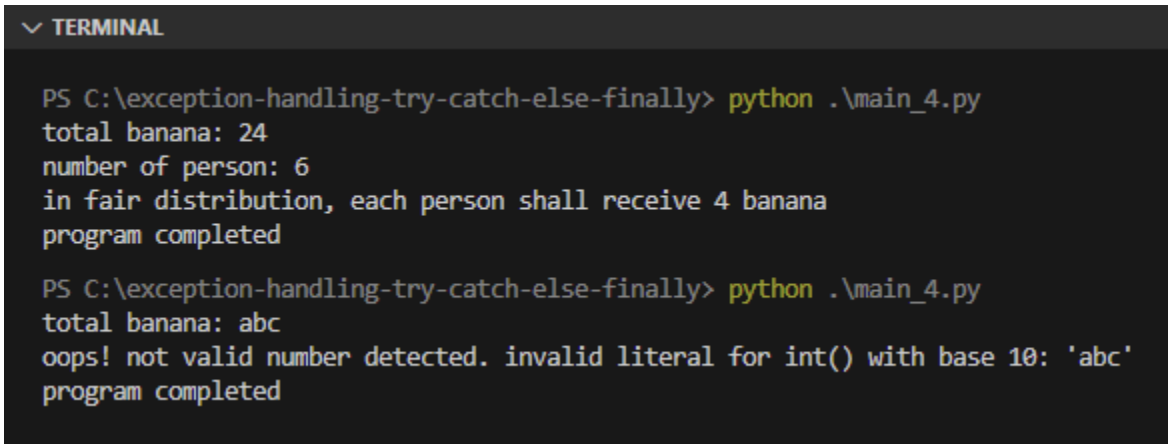
Sebagai contoh, kode sebelumnya dimodifikasi lagi menjadi seperti ini:

```
try:
    total_banana = int(input("total banana: "))
    total_people = int(input("number of person: "))
    res = total_banana / total_people
    print(f"in fair distribution, each person shall receive {res:.0f}
banana")
except ValueError as err:
    print(f"oops! not valid number detected. {err}")
except ZeroDivisionError as err:
    print(f"oops! unable to distribute banana because there is no person
available. {err}")
except Exception as err:
    print(f"oops! something wrong. {err}")
finally:
    print(f"program completed")
```

Penjelasan alur program di atas:

1. Program diawali dengan eksekusi statement dalam block `try`
2. Jika terjadi exception `ValueError`, maka dimunculkan pesan error
3. Jika terjadi exception `ZeroDivisionError`, maka dimunculkan pesan error
4. Jika terjadi exception lainnya, maka dimunculkan pesan error
5. Setelah program selesai, entah itu ada exception atau tidak, pesan

`program completed` di-print



A.47.5. Keyword `try`, `except`, `else` & `finally`

Bentuk sempurna dari exception handler adalah kombinasi dari 4 keyword yang telah dipelajari (`try`, `except`, `else` & `finally`).

- Block `try` untuk eksekusi statement
- Block `except` untuk menangkap exception
- Block `else` untuk kebutuhan ketika tidak ada exception
- Block `finally` untuk menandai bahwa eksekusi block exception handler telah selesai

Contoh program dengan penerapan 4 keyword ini:

```
try:
    total_banana = int(input("total banana: "))
    total_people = int(input("number of person: "))
    res = total_banana / total_people
except ValueError as err:
```

Penjelasan alur program di atas:

1. Program diawali dengan eksekusi statement dalam block `try`
2. Jika terjadi exception `ValueError`, maka dimunculkan pesan error
3. Jika terjadi exception `ZeroDivisionError`, maka dimunculkan pesan error
4. Jika terjadi exception lainnya, maka dimunculkan pesan error
5. Jika tidak terjadi exception sama sekali, maka block `else` dijalankan
6. Setelah program selesai, entah itu ada exception atau tidak, pesan `program completed` di-print

---

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../exception-handling-try-except-else-finally
```

◉ Chapter relevan lainnya

- Error & Exception

◉ TBA

- catch custom exception

◉ Referensi

- https://docs.python.org/3/library/exceptions.html
- https://docs.python.org/3/tutorial/errors.html

A.48. Python DocString

Pada chapter ini kita akan membahas tentang docstring beserta cara penerapan dan manfaatnya.

A.48.1. Pengenalan docstring

Di pembelajaran awal yaitu pada chapter Komentar, telah disinggung bahwa salah satu cara menulis komentar adalah menggunakan karakter `"""` dengan penulisan di awal dan akhir komentar. Contoh: `""" ini komentar """`.

Komentar yang ada di dalam karakter tersebut disebut docstring. DocString memiliki keistimewaan dibanding komentar biasa yang ditulis menggunakan karakter `#`.

Komentar docstring otomatis menempel pada unit dimana komentar ditulis. Misalnya ditulis tepat dibawah deklarasi fungsi bernama `print_random_quote()`, maka komentarnya menembel ke fungsi tersebut. Benefitnya, informasi komentar bisa muncul setidaknya di 2 tempat:

- Komentar bisa diakses menggunakan attribute `__doc__` yang menempel ke fungsi `print_random_quote()`.
- Komentar dimunculkan sewaktu fungsi tersebut di-hover (dengan catatan extension Python/Pylance ter-install di editor).

Perihal unit yang bisa ditempeli docstring bisa berupa fungsi, class, method, atau lainnya.

A.48.2. Praktek penerapan docstring

Mari kita praktekan agar lebih jelas. Siapkan project sederhana dengan struktur seperti ini. Isinya hanya dua file saja:

- File `quote.py` berisi class `Quote`
- File `main.py` berisi kode penggunaan class `Quote`

Project structure

```
praktek-docstring/
├── quote.py
└── main.py
```

Tulis kode berikut di file `quote.py`:

File quote.py

```
quotes = [
    "never let anyone live in your head rent free",
    "if others can do it, then why should I?",
    "\n".join([
        "I'm sick of following my dreams, man.",
        "I'm just going to ask where they're going and hook up with 'em
later."
    ]),
]

from random import randint

# function `print_random_quote()`:
#     print one random quote,
```

Kemudian di file `main.py`, import unit-unit dari module `quote.py` lalu gunakan.

File main.py

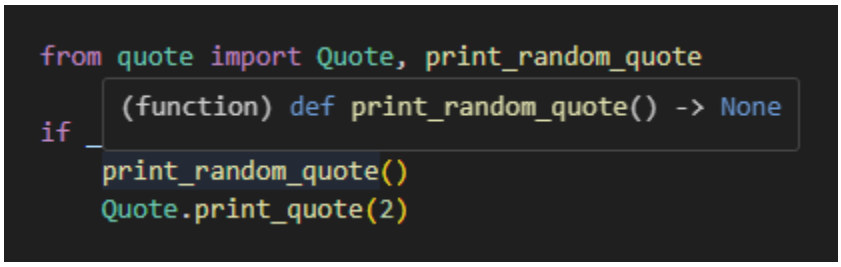
```
from quote import Quote, print_random_quote

if __name__ == '__main__':
    print_random_quote()
    # output → <random quote appears here>

    Quote.print_quote(2)
    # output → I'm just going to ask where they're going and hook up with
'em later.
```

Sampai sini penulis rasa cukup jelas.

Selanjutnya coba hover fungsi atau class yang di-import dari module `Quote.py` yang dipergunakan di `main.py`. Popup muncul tapi isinya hanya informasi deklarasi fungsi itu sendiri. Komentar yang telah ditulis tidak muncul di popup.



Ok, sekarang kita akan modifikasi komentar pada kode yang sudah ditulis dengan mengubahnya menjadi komentar docstring. Komentar hanya dianggap docstring ketika ditulis dengan diapit karakter `"""` `"""` dan penulisannya berada tepat dibawah unit yang ingin dikomentari.

◉ DocString pada class dan fungsi/method

Ubah isi `quote.py` menjadi seperti ini:

File quote.py

```
quotes = [
    "never let anyone live in your head rent free",
    "if others can do it, then why should I?",
    "\n".join([
        "I'm sick of following my dreams, man.",
        "I'm just going to ask where they're going and hook up with 'em
later."
    ]),
]

from random import randint

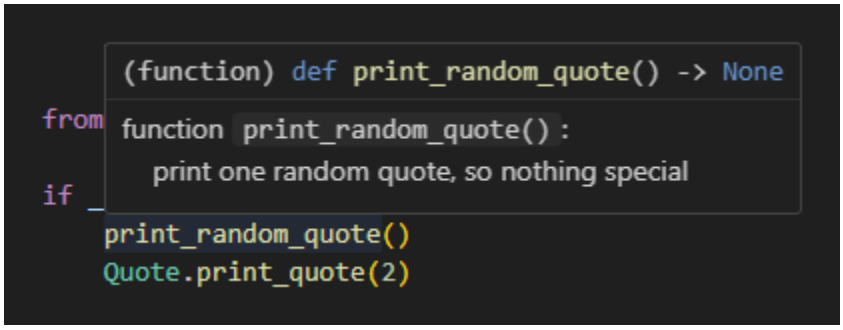
def print_random_quote():
    """
    function `print_random_quote()`:
        print one random quote,
        so nothing special
    """

    i = randint(0, len(quotes)-1)
    print(quotes[i])

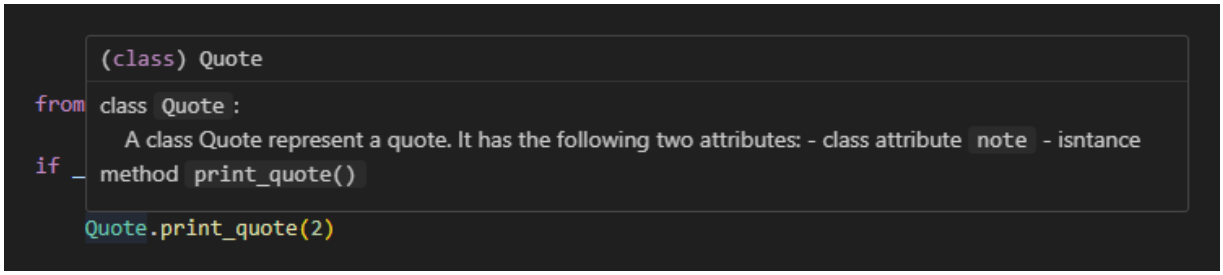
class Quote:
    """
    class `Quote`:
        A class Quote represent a quote.
        It has the following two attributes:
            - class attribute `note`
            - instance method `print_quote()`
    """
```

Sekarang coba hover lagi, lihat isi popup-nya.

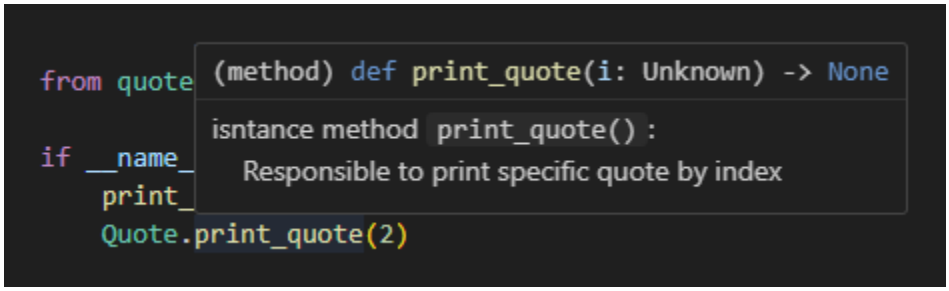
- Hover fungsi `print_random_quote()`



- Hover class `Quote`



- Hover class method `Quote.print_quote()`



Mantab bukan? DocString ini menjadi salah satu hal yang sangat membantu dalam pengembangan.

◉ DocString pada attribute dan variable

Untuk penerapan docstring pada attribute, caranya juga sama, yaitu dengan menuliskan komentar tepat dibawah attribute atau variabel dengan karakter `"""` `"""`.

File quote.py

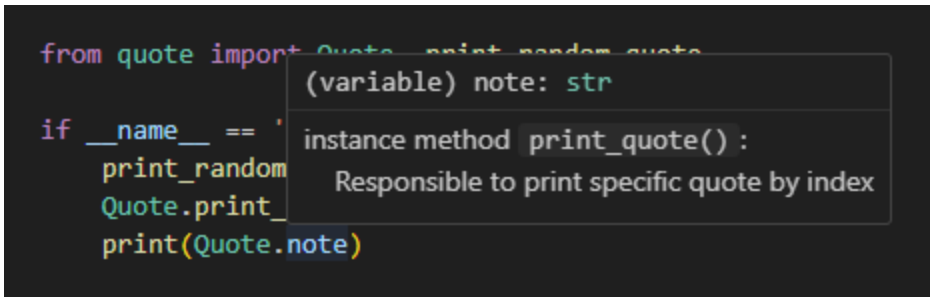
```
# ...

class Quote:
    """
    class `Quote`:
        A class Quote represent a quote.
        It has the following two attributes:
            - class attribute `note`
            - instance method `print_quote()`
    """

    note = "A class to represent quote"
    """
        instance method `print_quote()`:
            Responsible to print specific quote by index
    """

    @classmethod
    def print_quote(cls, i):
        """
        instance method `print_quote()`:
            Responsible to print specific quote by index
        """
        print(quotes[i])
```

Coba sekarang Output ketika di-hover:



A.48.3. Special name → class attribute `__doc__`

Informasi docstring milik fungsi, method, dan class bisa diakses secara excplit menggunakan class attribute `__doc__`. Jika mengacu ke kode yang sudah ditulis, maka pengaksesannya seperti ini:

```
from quote import Quote, print_random_quote

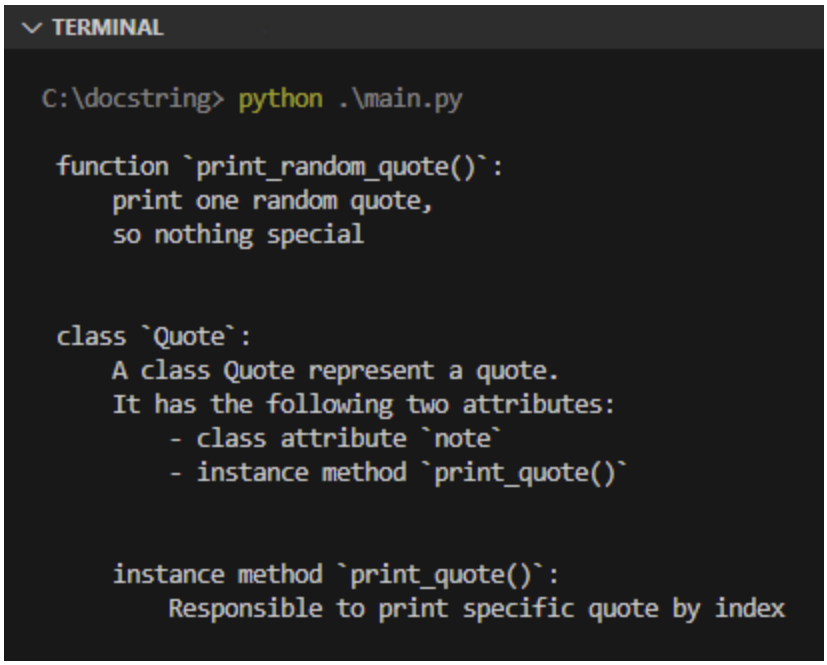
if __name__ == '__main__':

    # menampilkan docstring fungsi `print_random_quote()`
    print(print_random_quote.__doc__)

    # menampilkan docstring class `Quote`
    print(Quote.__doc__)

    # menampilkan docstring class method `Quote.print_quote()`
    print(Quote.print_quote.__doc__)
```

Output program:



Catatan chapter 🗒️

◉ Source code praktik

`github.com/novalagung/dasarpemrogramanpython-example/../docstring`

◉ Chapter relevan lainnya

- Komentar

◉ Referensi

- https://peps.python.org/pep-0257/

A.49. Python File I/O

Pada chapter ini kita akan belajar tentang pengolahan file dan folder, dan beberapa hal relevan lainnya yang masih berhubungan dengan manajemen file & folder.

A.49.1. Membuka stream file

Di Python, hampir semua operasi file diawali dengan pemanggilan fungsi `open()`, dan diakhiri pemanggilan method `close()` (milik object yang dikembalikan oleh fungsi `open()`).

Fungsi `open()` dalam penggunannya membutuhkan pengisian beberapa parameter:

- Parameter ke-1: nama file.
- Parameter ke-2: mode I/O, ada beberapa mode operasi file, diantaranya:
  - `w` untuk mode tulis dengan posisi kursor ada di baris paling awal. Jadi operasi penulisan bisa menimpa konten yang sudah ada. Selain itu, mode ini membuat isi file otomatis menjadi kosong saat fungsi `open()` dipanggil.
  - `a` untuk mode *append*, yaitu mode tulis dengan posisi kursor sudah di baris paling akhir. Jadi penambahan konten tidak akan menimpa konten sebelumnya, tapi ditambahkan di akhir.
  - `r` untuk mode baca.
- Ada juga parameter opsional lainnya, salah satunya `encoding` yang umum diisi dengan nilai `utf-8`.

Bentuk paling sederhana penerapan operasi buka file:

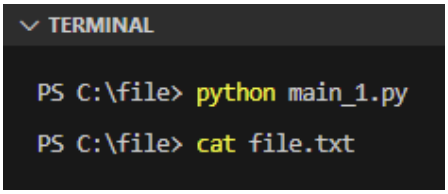
```
f = open("/path/to/file/file.txt", "w", encoding="utf-8")

# ...
```

Kode di atas akan membuka stream I/O file bernama `file.txt`. Jika file tujuan belum ada, maka otomatis dibuatkan oleh Python. Dari object kembalian fungsi `open()` nantinya kita bisa lakukan banyak jenis operasi seperti membaca isi file, menulis, menghapus, dan lainnya.

> *Untuk pengguna Windows, tulis saja path-nya dengan karakter `\` ter-escape. Contoh: `"C:\\Users\\novalagung\\Desktop\\file.txt"`*

Silakan tulis kode di atas, lalu ganti path-nya dengan current path (atau bisa gunakan `.`), kemudian run programnya. Hasil eksekusi program adalah pembuatan sebuah file baru bernama `file.txt` yang isinya kosong.



Mode `w` digunakan disitu, artinya file dibuka dengan mode tulis. Salah satu efek dari penggunaan mode `w` adalah ketika file dibuka isinya pasti dikosongkan terlebih dahulu.

File yang dibuka, wajib untuk selalu ditutup di akhir. Karena membiarkan file tetap terbuka beresiko membuat isi file rusak ketika ada lebih dari 1 pengakses yang melakukan operasi terhadap file tersebut secara bersamaan.

Untuk mengecek apakah file sedang terbuka stream-nya, bisa dengan melihat nilai attribute `closed`.

```
f = open("file.txt", "w", encoding="utf-8")
print("file is closed:", f.closed)
# output → file is closed: False

# ...
```

A.49.2. Keyword `with`

Ada cara yang lebih efisien dalam operasi buka file agar file otomatis ter-close setelah digunakan, yaitu dengan menggunakan keyword `with` diikuti statement `open()` lalu syntax `as nama_variabel`. Kurang lebih seperti ini penulisannya:

```
with open("file.txt", "w", encoding="utf-8") as f:
    print("file is closed:", f.closed)
    # output → file is closed: False
    # ...

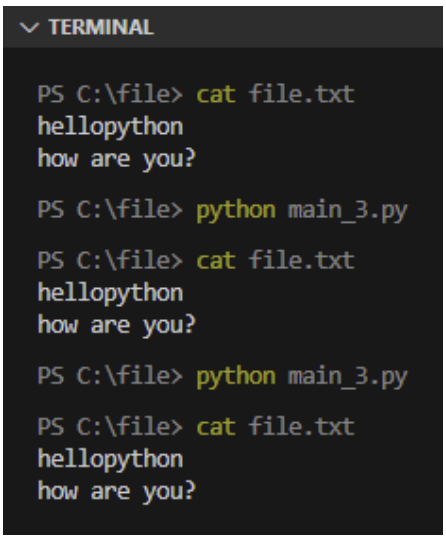
print("file is closed:", f.closed)
# output → file is closed: True
```

A.49.3. Menulis file

Operasi penulisan konten ke file dilakukan via method `write()` milik object file. Contoh penerapannya bisa dilihat pada kode berikut, dimana ada method `write()` digunakan 3x untuk menulis karakter string.

```
with open("file.txt", "w", encoding="utf-8") as f:
    f.write("hello")
    f.write("python\n")
    f.write("how are you?\n")
```

Output program:



Program di-run 3x dan isinya tetap sama (tidak menumpuk), ini karena setiap kali statement `open()` dijalankan dengan mode `w`, file akan dikosongkan terlebih dahulu.

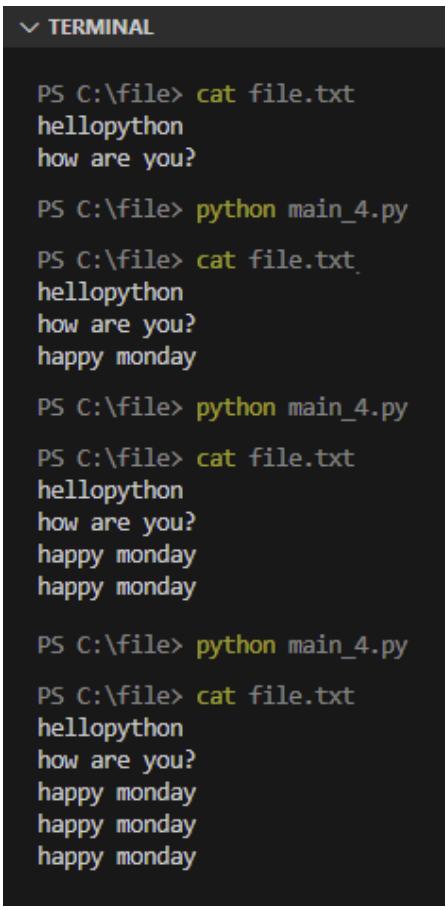
A.49.4. Append konten ke file

Gunakan mode `a` untuk append konten ke file yang isinya bisa saja tidak kosong (agar isi konten tidak ditimpa).

Coba jalankan kode berikut terhadap file `file.txt` yang sebelumnya sudah dibuat. Saat program di-run kondisi file sudah terisi dan tidak dikosongkan terlebih dahulu. Dengan mengeksekusi `write()` disitu maka isi konten akan bertambah terus setiap kali program di-run.

```
with open("file.txt", "a", encoding="utf-8") as f:
    f.write("happy monday\n")
```

Output program:



Bisa dilihat, setiap kali program dieksekusi konten `happy monday\n` bertambah terus.

A.49.5. Membaca file

Method `readline()` dan `read()` milik object file, keduanya digunakan untuk membaca isi file.

- Method `readline()` akan membaca isi file baris per baris. Pembacaan dimulai dari baris paling atas dan proses pembacaan terjadi setiap kali method `readline()` dipanggil. Ketika method ini mengembalikan string kosong, bisa jadi menandakan semua baris konten file sudah terbaca.

  Disini penulis gunakan kata *bisa jadi* karena ada kasus dimana pada beberapa

baris bisa saja memang isinya sengaja kosong.

- Method `read()` akan membaca seluruh isi file. Pemanggilan method ini untuk kedua kalinya pasti mengembalikan string kosong, menandakan semua baris konten file sudah terbaca.

Mari praktekan penggunaan 2 method di atas. Pertama isi file `file.txt` secara manual dengan konten berikut:

```
hellopython
how are you?
happy monday
```

Kemudian baca isinya per baris menggunakan kode berikut:

```
with open("file.txt", "a", encoding="utf-8") as f:
    print(f"line 1: {f.readline()}")
    print(f"line 2: {f.readline()}")
    print(f"line 3: {f.readline()}")
    print(f"line 4: {f.readline()}")
    print(f"line 5: {f.readline()}")
```

Output program:

```
---------------------------------------------------------------------------
UnsupportedOperation                      Traceback (most recent call last)
c:\LibsSoftLink\dasarpemrogramanpython\examples\file\main_5.py in line 4
      1 # %% A.46.5. Membaca file
      3 with open("file.txt", "a", encoding="utf-8") as f:
----> 4     print(f"line 1: {f.readline()}")
      5     print(f"line 2: {f.readline()}")
      6     print(f"line 3: {f.readline()}")

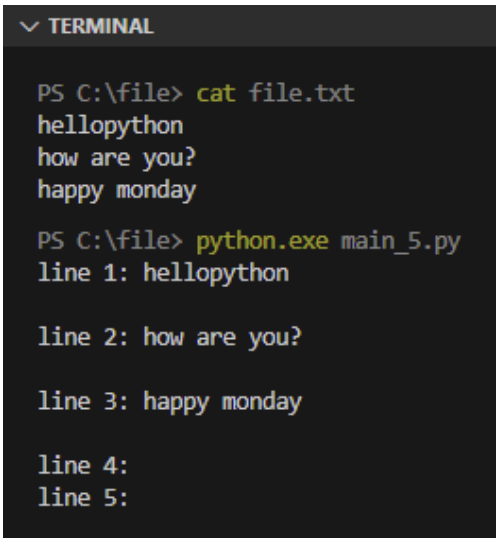
UnsupportedOperation: not readable
```

Malah error? Kok bisa? Error ini disebabkan karena kita menggunakan mode `a`

yang mode tersebut hanya valid untuk operasi append. Kita perlu mengubah mode menjadi `r` untuk operasi pembacaan file.

```
with open("file.txt", "r", encoding="utf-8") as f:
    print(f"line 1: {f.readline()}")
    print(f"line 2: {f.readline()}")
    print(f"line 3: {f.readline()}")
    print(f"line 4: {f.readline()}")
    print(f"line 5: {f.readline()}")
```

Output program:



Bisa dilihat method `readline()` mengembalikan data per baris dari atas ke bawah dengan jumlah sesuai dengan berapa kali baris method tersebut dipanggil.

Dalam penerapannya, dianjurkan untuk menggunakan method ini dalam perulangan kemudian ditambahkan pengecekan isi konten untuk menandai bahwa konten sudah terbaca semua. Contohnya seperti ini:

```
with open("file.txt", "r", encoding="utf-8") as f:
    i = 0
    while True:
        line = f.readline()
```

Kode di atas bisa disederhanakan lagi dengan cara langsung mengiterasi object file-nya. Jadi variabel `f` digunakan secara langsung pada statement perulangan. Hal ini bisa dilakukan karena tipe data kembalian fungsi `open()` adalah `TextIOWrapper` dan tipe ini termasuk tipe data yang *iterable*.

```
with open("file.txt", "r", encoding="utf-8") as f:
    i = 1
    for line in f:
        print(f"line {i}: {line}")
        i += 1
```

> *Lebih detailnya mengenai tipe data iterable dibahas pada chapter Iterator*

Kode yang sudah cukup ringkas di atas bisa disederhanakan lagi dengan cara membungkus tipe data `f` dalam fungsi `enumerate()`. Fungsi ini membuat suatu object yang iterable menjadi memiliki index di setiap element-nya.

```
with open("file.txt", "r", encoding="utf-8") as f:
    for i, line in enumerate(f):
        print(f"line {i+1}: {line}")
```

Jika goal dari program adalah hanya membaca isi file secara menyeluruh, sebenarnya lebih praktis lagi menggunakan method `read()`.

```
with open("file.txt", "r", encoding="utf-8") as f:
    print(f.read())
```

A.49.6. Membaca dan menulis dalam 1 sesi

Di awal chapter telah dijelaskan tentang kegunaan mode `w`, `a`, dan `r`. Lalu bagaimana jika ada kebutuhan untuk membaca dan menulis file dalam satu sesi? Jawabannya adalah dengan menambahkan tanda `+` pada mode (jadinya `w+`, `a+`,

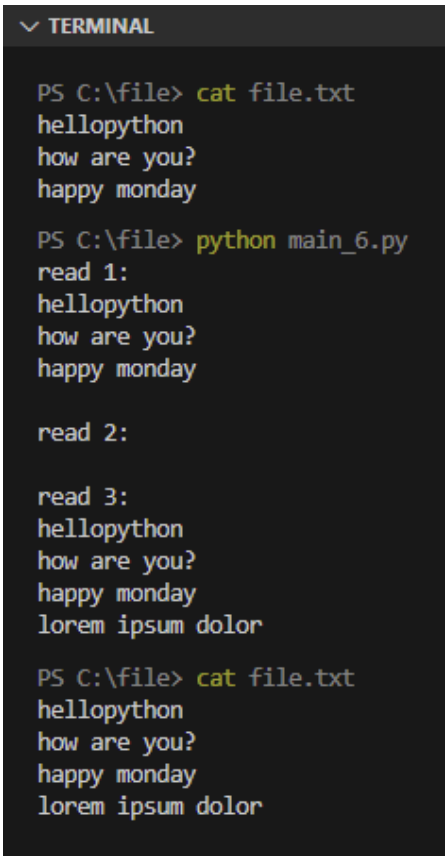
atau `r+`).

Sebagai contoh, pada program berikut, mode `r+` digunakan. O iya, proses pembacaan file dilakukan 2x ya, penjelasan disertakan dibawahnya.

```
with open("file.txt", "r+", encoding="utf-8") as f:
    print(f"read:\n{f.read()}")
    f.write("lorem ipsum dolor\n")
    print(f"read:\n{f.read()}")

with open("file.txt", "r+", encoding="utf-8") as f:
    print(f"read:\n{f.read()}")
```

Output program:



Bisa dilihat di program, block `with` pertama yang berisi operasi baca dan juga tulis

tidak menghasilkan error. Namun ada yang aneh, yaitu tepat setelah `lorem ipsum dolor\n` ditulis ke file, proses baca menghasilkan string kosong. Tapi ketika file dibaca lagi menggunakan block `with` baru, isinya sesuai harapan. Jawabannya adalah karena **pergerakan cursor**.

Flow program di atas kurang lebih seperti ini:

1. Cursor awal pembacaan file ada di baris paling awal, karena mode `r+` digunakan.
2. Setelah method `read()` dipanggil, cursor berada di posisi paling akhir.
3. Kemudian `lorem ipsum dolor\n` ditulis ke file, maka text tersebut ada di baris baru di akhir file.
4. Lalu ketika method `read()` dibaca lagi, isinya kosong, karena cursor posisinya sudah ada di baris akhir file.
5. Kemudian ketika file dibaca ulang menggunakan fungsi `open()` dengan block `with` baru, cursor kembali aktif di baris paling awal.
6. Lalu file dibaca, maka seluruh isi konten yang baru dikembalikan.

Mode `w+`, `a+`, dan `r+` kesemuanya bisa digunakan untuk baca dan tulis dalam 1 sesi, dengan perbedaan ada di posisi kursornya aktif dimana. Jika pembaca tertarik untuk mempelajarinya lebih detail, silakan baca diskusi di StackOverflow berikut:

https://stackoverflow.com/questions/1466000/difference-between-modes-a-a-w-w-and-r-in-built-in-open-function/30566011#30566011

A.49.7. Mengosongkan isi file

Cara mengosongkan file bisa dilakukan dengan mudah menggunakan mode `w`. Baca file menggunakan mode tersebut kemudian langsung `close()` saja. Boleh menggunakan keyword `with` atau bisa langsung sebaris statement. Contoh penerapannya bisa dilihat di kode berikut. 3 block statement di situ semuanya ekuivalen, membuat isi file menjadi kosong.

```
with open("file.txt", "w", encoding="utf-8") as f:
    pass

with open("file.txt", "w", encoding="utf-8"):
    pass

open("file.txt", "w", encoding="utf-8").close()
```

Opsi lainnya adalah menggunakan method `truncate()`.

```
with open("file.txt", "w", encoding="utf-8") as f:
    f.truncate()
```

A.49.8. Menghapus file atau folder

API `os.remove()` digunakan untuk menghapus file, sedangkan `os.rmdir()` untuk menghapus folder. Contoh penerapan:

- Menghapus file:

  ```
  import os

  os.remove("/path/to/something/file.txt")
  ```

- Menghapus folder:

  ```
  import os

  os.rmdir("/path/to/something")
  ```

  Untuk path berbasis windows, pastikan karakter `\` ditulis dengan cara di-escape (ditulis `\\`).

```
import os

os.rmdir("C:\\LibsSoftLink\\dasarpemrogramanpython\\examples")
```

A.49.9. Mengecek apakah file atau folder ada

API `os.path.isfile()` digunakan untuk mengecek apakah suatu file ada.

```
import os.path

if os.path.isfile("/path/to/something/file.txt"):
    print("file.txt is exists")
else:
    print("file.txt is not exists")
```

Untuk pengecekan terhadap folder, gunakan `os.path.exists()`. Fungsi ini bisa digunakan baik untuk pengecekan file ataupun folder.

```
if os.path.exists("/path/to/something"):
    print("something is exists")
else:
    print("something is not exists")
```

Untuk path berbasis windows, pastikan karakter `\` ditulis dengan cara di-escape (ditulis `\\`).

```
if
os.path.exists("C:\\LibsSoftLink\\dasarpemrogramanpython\\examples\\file.txt"):
    print("file.txt is exists")
else:
    print("file.txt is not exists")
```

A.49.10. Membuat folder baru

API `os.makedirs()` digunakan untuk membuat folder baru.

```
import os

os.makedirs("/path/to/somefolder")
```

Untuk path berbasis windows, pastikan karakter `\` ditulis dengan cara di-escape (ditulis `\\`).

```
import os

os.makedirs("C:\\LibsSoftLink\\dasarpemrogramanpython\\examples")
```

A.49.11. Menampilkan isi folder

- Menggunakan `os.listdir()`:

  ```
  import os

  path_location = "C:\\LibsSoftLink\\dasarpemrogramanpython\\examples\\file"
  for f in os.listdir(path_location):
      print(f)
  ```

- Menggunakan `os.walk()`:

  ```
  import os

  path_location = "C:\\LibsSoftLink\\dasarpemrogramanpython\\examples\\file"
  for (dirpath, dirnames, filenames) in os.walk(path_location):
  ```

Penjelasan:

  - Variabel `dirpath` berisi current
  - Variabel `dirnames` berisi folder yang berada dalam current folder
  - Variabel `filenames` berisi file yang berada dalam current folder

- Menggunakan `glob.glob()`:

  API `glob.glob()` ini didesain untuk pencarian. Jadi pada penerapannya perlu ditambahi kondisi *wildcard* pencarian. Misalnya, dengan menambahkan `**` di akhir path, maka pencarian dilakukan terhadap semua jenis file dan folder.

```
import glob

path_location = "C:\\LibsSoftLink\\dasarpemrogramanpython\\examples\\file"
for f in glob.glob(f"{path_location}\\**", recursive=True):
    print(f)
```

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../file
```

◉ Chapter relevan lainnya

- File/Data Format ➜ CSV
- File/Data Format ➜ JSON

◉ TBA

- Pathlib https://docs.python.org/3/library/pathlib.html
- Search pattern

◉ Referensi

- https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files
- https://stackoverflow.com/questions/1466000/difference-between-modes-a-a-w-w-and-r-in-built-in-open-function/30566011#30566011
- https://builtin.com/data-science/python-list-files-in-directory

A.50. Python CLI Arguments & Flags

Chapter ini membahas tentang pengaksesan CLI argument eksekusi program Python. Yang dimaksud dengan argument adalah apapun yang ditulis setelah command `python` (atau `python.exe` di Windows). Umumnya program yang dibuat untuk keperluan tooling ataupun utility lainnya banyak memanfaatkan *command line interface* arguments ini.

Python mengenal 2 jenis CLI arguments, yaitu raw arguments (atau cukup arguments) dan flags, keduanya dibahas di sini.

A.50.1. CLI arguments `sys.argv`

Data CLI argument eksekusi program bisa diakses via `sys.argv`. Variabel `sys.argv` ini berisi data argument bertipe string tersimpan dalam list.

> *Dalam penggunaannya module `sys` harus di-import terlebih dahulu*

Misalnya, pada command berikut ini, pengaksesan variabel `sys.argv` menghasilkan data `["main.py"]`, karena `main.py` merupakan CLI argument yang ditulis setelah command `python`/`python.exe`.

```
python main.py
```

Argument bisa saja ada banyak, misalnya pada kode berikut terdapat 3 argument, `main.py`, `sesuk`, dan `prei`. Penulisan banyak argument ditandai

dengan pembatas karakter spasi.

```
python main.py sesuk prei
```

Mengenai command `python` sendiri, dia hanya membutuhkan informasi argument pertama saja untuk menentukan file mana yang akan dieksekusi. Argument ke-2, ke-3, dan seterusnya tidak dibutuhkan oleh command `python`, tetapi tetap bisa dipergunakan untuk keperluan lainnya.

Silakan coba tulis kode berikut kemudian run program-nya menggunakan command di bawahnya.

```
import sys

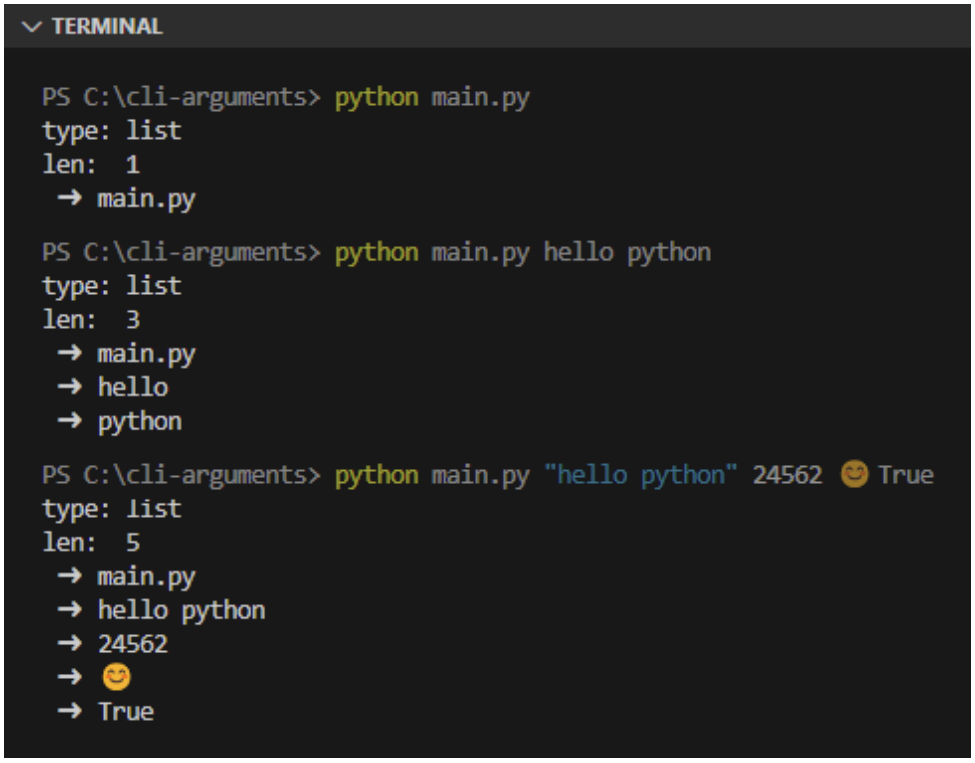
if __name__ == "__main__":
    print(f"type: {type(sys.argv).__name__}")
    print(f"len:  {len(sys.argv)}")

    for arg in sys.argv:
        print(f" ➜  {arg}")
```

Command eksekusi program:

```
python.exe main_1.py
python.exe main_1.py hello python
python.exe main_1.py "hello python" 24562 😊 True
```

Output program:



Silakan lihat perbandingan antara command dengan output. Pada command ke-2, `hello` dan `python` merupakan dua argument berbeda, sedangkan pada command ke-3, `hello python` adalah satu argument karena penulisannya diapit tanda literal string (`"`).

Semua argument, baik itu angka, emoji, ataupun karakter unicode lainnya akan ditampung sebagai elemen list bertipe string di `sys.argv`.

◉ Best practice pengaksesan argument

`sys.argv` merupakan list, maka pengaksesan element tertentu yang index-nya diluar kapasitas adalah menghasilkan error. Karena alasan ini ada baiknya untuk berhati-hati dalam mengakses argument di index tertentu pada variabel tersebut, pastikan untuk menambahkan seleksi kondisi terlebih dahulu untuk mengecek apakah index dari element yang dicari masih dalam kapasitas.

Salah satu solusi aman bisa dengan membuat fungsi terpisah untuk

pengaksesan argument, contohnya bisa dilihat pada kode berikut dimana jika argument yang dicari tidak ada, maka nilai `None` dikembalikan. Metode ini lebih efisien.

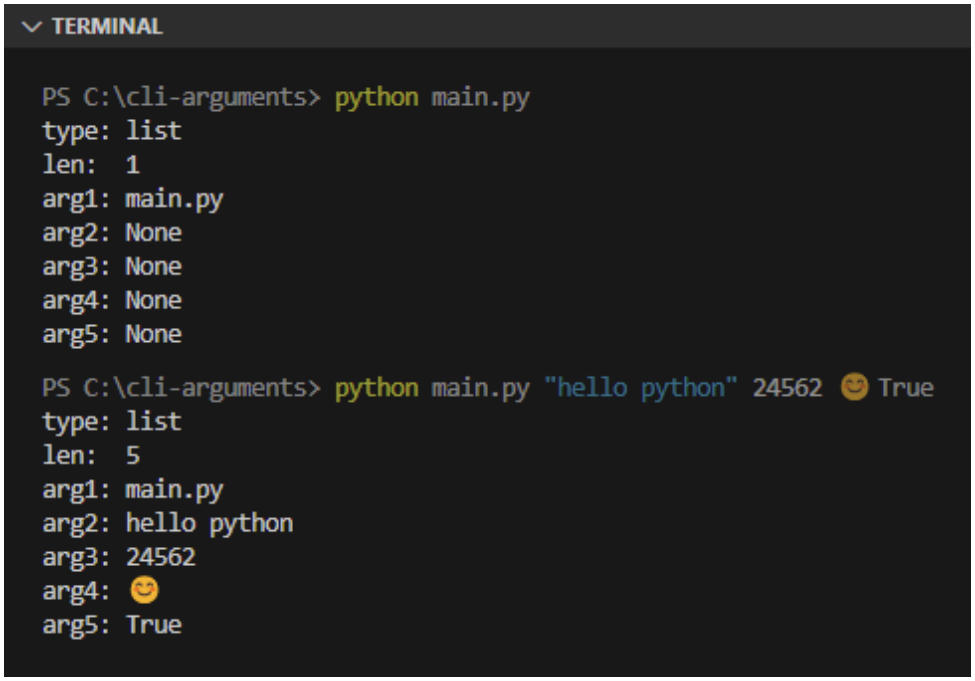
```
import sys

def get_arg(index):
    if len(sys.argv) > index:
        return sys.argv[index]
    else:
        return None

if __name__ == "__main__":
    print(f"type: {type(sys.argv).__name__}")
    print(f"len:  {len(sys.argv)}")

    print(f"arg1: {get_arg(0)}")
    print(f"arg2: {get_arg(1)}")
    print(f"arg3: {get_arg(2)}")
    print(f"arg4: {get_arg(3)}")
    print(f"arg5: {get_arg(4)}")
```

Output program:



A.50.2. CLI flags `argparse`

Flags adalah istilah untuk argument dengan label. Contohnya seperti `python main.py --name Noval` adalah contoh pengaplikasian flag, dengan label adalah `--name` berisi data string `Noval`.

Python menyediakan module `argparse` berisi banyak API untuk keperluan operasi flag argument.

Untuk menerapkan flag, sebuah object parser perlu dibuat menggunakan `argparse.ArgumentParser()`, dengan isi parameter adalah informasi program (seperti nama dan deskripsi). Kemudian dari object tersebut, didaftarkan beberapa flag argument (misalnya `--name`) beserta property-nya. Lalu di akhir, method `parse_args()` wajib ditulis dan pengaksesan nilai flag dilakukan dari object kembalian method tersebut.

Program di bawah ini berisi penerapan flag argument untuk program sederhana

yang kegunaannya untuk pembuatan file. Nama dan path file beserta isinya dikontrol via CLI flag.

```
import argparse

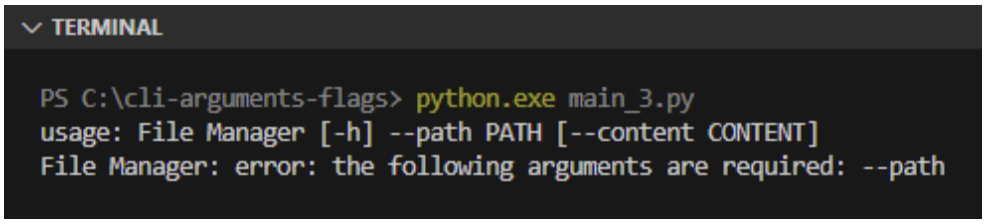
def main():
    parser = argparse.ArgumentParser(
        prog='File Manager',
        description='Managing file easily'
    )

    parser.add_argument("--path", help="path of the file/folder",
default="file.txt", required=True)
    parser.add_argument("--content", help="content of the file",
default="")
    args = parser.parse_args()

    with open(args.path, 'a') as f:
        f.write(args.content)

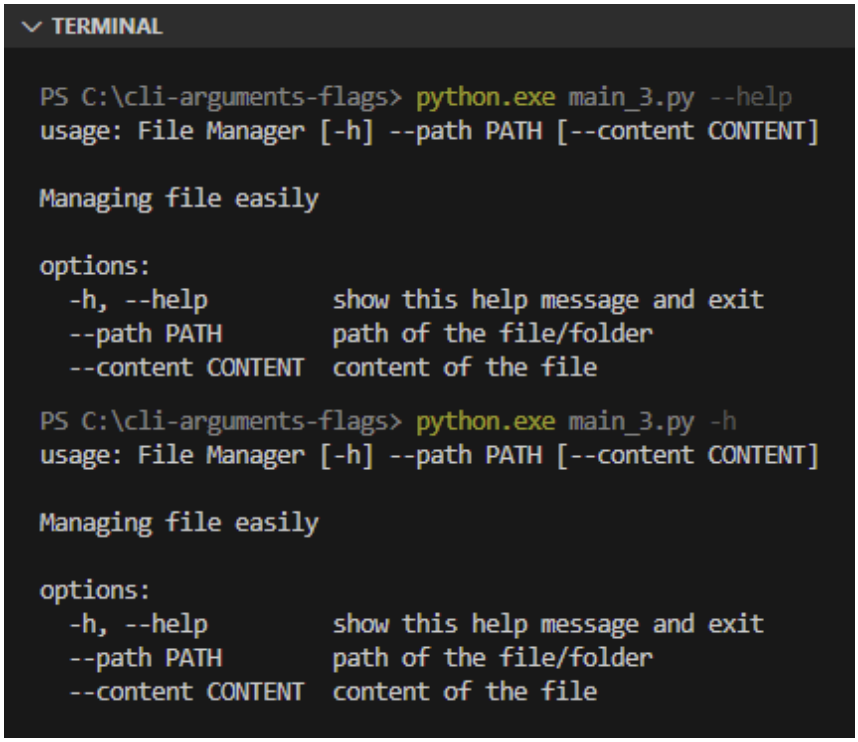
if __name__ == "__main__":
    main()
```

Coba jalankan program di atas dengan perintah standar `python main.py`. Outputnya:



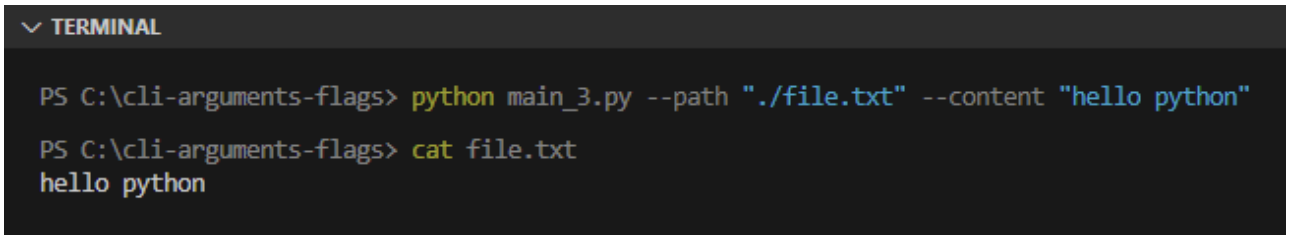
Error muncul, karena ada salah satu flag yang di-setting untuk wajib diisi, yaitu `--path`. Bisa dilihat pada statement `parser.add_argument("--path", ...)` di atas, disitu parameter `required` di-isi dengan `True`, menjadikan pemanggilan program tanpa flag ini memunculkan error.

Untuk melihat flag apa saja yang tersedia, gunakan flag `--help` atau `-h`. Dengan flag tersebut, informasi nama program dan flag yang tersedia dimunculkan.



Sekarang coba jalankan command dengan disertai isi flag `--path` dan `--content`. Program akan berjalan sesuai desain. File terbuat dengan isi sesuai nilai flag `--content`.

```
python main.py --path "./file.txt" --content "hello python"
cat file.txt
```



Kembali ke bagian deklarasi flag menggunakan `parser.add_argument()`, dalam pembuatannya ada beberapa konfigurasi yang bisa dimanfaatkan sesuai kebutuhan, diantaranya:

Parameter	**Keterangan**
Positional param	Diisi dengan nama flag, umumnya menggunakan notasi `--nama` untuk flag dengan label kata atau frasa, dan `-n` saja untuk huruf. Keduanya juga bisa digunakan bersamaan, misalnya `parser.add_argument("--path", "-p", ...)` baru setelahnya diikuti keyword argument.
Keyword param `help`	Diisi dengan informasi keterangan flag. Nantinya muncul saat `--help` digunakan.
Keyword param `default`	Diisi dengan nilai default flag ketika tidak ditulis nilainya secara explicit.
Keyword param `required`	Penanda apakah flag wajib diisi atau opsional. Jika diisi `True` maka wajib untuk diisi dan memunculkan error jika tidak diisi
Keyword param `choices`	Jika diisi dengan nilai list, maka element list menjadi opsi pengisian flag. Jika saat pemanggilan flag diisi dengan nilai yang tidak ada di list maka error muncul.

> *Selain parameter di atas, ada juga beberapa lainnya. Selengkapnya bisa*

> *cek di halaman dokumentasi https://docs.python.org/3/library/argparse.html*

Ok, agar makin paham, mari praktek lagi dengan memodifikasi program sebelumnya menjadi seperti ini:

```
def main():
    parser = argparse.ArgumentParser(
        prog='File Manager',
        description='Managing file easily'
    )

    parser.add_argument("--operation-mode", "-op", help="choose
operation", choices=["write file", "list items"], required=True)
    parser.add_argument("--path", "-p", help="path of the file/folder",
default=".", required=True)
    parser.add_argument("--content", "-c", help="content of the file",
default="")
    args = parser.parse_args()

    if args.operation_mode == "write file":
        with open(args.path, 'a') as f:
            f.write(args.content)

    elif args.operation_mode == "list items":
        for f in glob.glob(f"{args.path}/**", recursive=True):
            print(f)

if __name__ == "__main__":
    main()
```

Perbedaan program terbaru dibanding sebelumnya:

- Flag baru ditambahkan bernama `--operation-mode` atau `-op`, flag ini yang wajib diisi nilai `write file` atau `list items`.

- Flag `--path` dibuatkan *shorthand*-nya yaitu `-p`.
- Flag `--content` dibuatkan *shorthand*-nya yaitu `-c`.
- Ketika flag `-op` bernilai `write file`, maka program melakukan penulisan konten dengan isi dan tujuan file sesuai dengan nilai flag saat eksekusi.
- Ketika flag `-op` bernilai `list items`, maka program menampilkan list files/folders pada path yang ditentukan via flag `--path`.

O iya, perlu diketahui bahwa ketika flag labelnya adalah frasa dan menggunakan karakter `-` sebagai pembatas kata, maka pengaksesannya menggunakan pembatas `_`. Contohnya bisa dilihat pada flag `--operation-mode` yang pengaksesan nilainya dilakukan via `args.operation_mode`.

Jalankan program dengan 3 perintah berikut lalu lihat outputnya:

```
python.exe main_4.py --help
python.exe main_4.py --op "write file" -p "./file.txt" -c "hello python"
python.exe main_4.py --op "list items" -p "./"
```

Output program:

```
TERMINAL

PS C:\cli-arguments-flags> python.exe main_4.py --help
usage: File Manager [-h] --operation-mode {write file,list items} --path PATH [--content CONTENT]

Managing file easily

options:
  -h, --help            show this help message and exit
  --operation-mode {write file,list items}, -op {write file,list items}
                        choose operation
  --path PATH, -p PATH  path of the file/folder
  --content CONTENT, -c CONTENT
                        content of the file

PS C:\cli-arguments-flags> python.exe main_4.py --op "write file" -p "./file.txt" -c "hello python"

PS C:\cli-arguments-flags> cat file.txt
hello python

PS C:\cli-arguments-flags> python.exe main_4.py --op "list items" -p "./"
.\
.\file.txt
.\main_1.py
.\main_2.py
.\main_3.py
.\main_4.py
```

Catatan chapter 📑

◉ Source code praktik

`github.com/novalagung/dasarpemrogramanpython-example/../cli-arguments-flags`

◉ Chapter relevan lainnya

- File I/O

◉ TBA

- Flag without value https://stackoverflow.com/questions/8259001/python-argparse-command-line-flags-without-arguments

◉ Referensi

- https://docs.python.org/3/library/argparse.html
- https://stackoverflow.com/questions/35603729/difference-between-single-dash-and-double-dash-in-argparse

A.51. Python CSV File/ Data Format

Pada chapter ini kita akan belajar tentang penggunaan Python untuk membaca dan mengelola data CSV.

INFO

CSV sendiri merupakan suatu format data yang sangat populer digunakan untuk menyimpan data berbasis tabular. Struktur isi CSV kurang lebih seperti ini:

```
name,email,phone
Noval,[email protected],087877665
John Doe,[email protected],
Garrosh Hellscream,,098878212
```

Penjelasan singkat tentang CSV:

- Baris pertama umumnya berisi informasi header, atau nama kolom.
- Pada contoh di atas, ada 3 buah header, yaitu `name`, `email`, dan `phone`.
- Tanda koma *default*-nya digunakan sebagai pembatas data dan kolom.
- Setiap baris data ditandai dengan *newline* atau baris baru. Umumnya data dimulai pada baris ke-2.
- Di setiap baris, data ditulis sesuai urutan header menggunakan pembatas tanda koma.
- Bisa saja ada data yang isinya tidak lengkap, misalnya di contoh data

> ke-2 tidak memiliki data `phone`, dan data ke-3 tidak memiliki data `email`.

A.51.1. Implementasi CSV di Python

Python menyediakan beberapa API untuk berinteraksi dengan data CSV, ada `csv.DictWriter()` dan `csv.writer()` untuk operasi tulis data, dan `csv.DictReader()` dan `csv.reader()` untuk operasi baca data.

Pembelajaran pada chapter ini dilakukan lewat pembuatan program sederhana yang mengakomodir operasi create, read, dan delete data.

Sebelum mulai, silakan siapkan project baru dengan isi sebuah file `main.py` berisi kode berikut:

```
def control_flow():
    while True:
        print("Choose mode:")
        print("1. Write data")
        print("2. Read data")
        print("3. Delete by row")
        print("4. Exit")

        mode = input('Choice (1/2/3/4): ')
        if mode == '1':
            pass # to be replaced
        elif mode == '2':
            pass # to be replaced
        elif mode == '3':
            pass # to be replaced
        elif mode == '4':
            break
        else:
```

Program yang dibuat adalah aplikasi CLI interaktif, dimana user bisa berinteraksi dengan program sesuai pilihan mode. Ada 4 buah mode:

1. Insert data (ke file CSV)
2. Read data (dari file CSV)
3. Delete data berdasarkan index
4. Exit program

Selanjutnya, program dimodifikasi dengan penambahan implementasi untuk masing-masing mode.

A.51.2. Insert data CSV

Sebuah file CSV akan dibuat dan difungsikan sebagai *database* untuk penyimpanan data yang di-insert via program. File CSV berisi 3 buah kolom data point:

- Name
- Email
- Phone

Pertama, buka file dimana kode ditulis, lalu tambahkan kode berikut di awal program:

```
import csv

filename = 'data.csv'
fieldnames = ['name', 'email', 'phone']

def prepare_csv():
    with open(filename, 'w', newline='') as f:
        writer = csv.DictWriter(f, fieldnames=fieldnames)
```

Fungsi `prepare_csv()` bertugas untuk membuat file csv bernama `data.csv` dengan isi kosong. Di dalam fungsi tersebut proses berikut dilakukan:

- Fungsi `open()` membuka file `data.csv` dengan mode tulis (`w`). Disitu parameter opsional `newline` diisi dengan string kosong agar di akhir file tidak dituliskan baris baru.
- Fungsi `csv.DictWriter()` parameter pertama diisi object file, dan parameter `fieldnames` diisi informasi header dalam bentuk list.
- Method `writeheader()` milik object writer digunakan untuk menulis header ke file.

Sekarang fokus ke fungsi `main()`, panggil fungsi `prepare_csv()` yang telah dibuat. Kode fungsi `main()` setelah perubahan menjadi seperti ini:

```
def main():
    prepare_csv()
    control_flow()
```

Coba jalankan program, lalu cek isi file `data.csv` untuk memastikan operasi penulisan header berjalan sesuai seharusnya.

Output program:

```
PS C:\LibsSoftLink\dasarpemrogramanpython\examples\csv> python main_2.py
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 4
PS C:\LibsSoftLink\dasarpemrogramanpython\examples\csv> cat data.csv
name,email,phone
PS C:\LibsSoftLink\dasarpemrogramanpython\examples\csv>
```

Selanjutnya, buat lagi fungsi baru bernama `write_data()`, gunanya untuk penulisan data `name`, `email`, dan `phone` ke file `data.csv`.

```
def write_data(name, email, phone):
    with open(filename, 'a', newline='') as f:
        writer = csv.DictWriter(f, fieldnames=fieldnames)
        writer.writerow({
            'name': name,
            'email': email,
            'phone': phone
        })
```

Untuk menulis data CSV, cara awalnya kurang lebih sama, yaitu object file dibuat dulu menggunakan fungsi `open()` kemudian dimasukan ke `csv.DictWriter()`. Selain itu, ada beberapa perbedaan dibanding kode sebelumnya:

- Disini file dibuka menggunakan mode append (`a`). Hal ini agar header yang sudah ditulis di file tidak hilang.
- Method `writerow()` milik writer digunakan untuk pengisian data. Penggunaannya mengharuskan data untuk diformat dalam bentuk dictionary.

Data yang bisa dituliskan ke CSV adalah `int`, `float`, dan `str`. Jika ada tipe data lain yang perlu disimpan di CSV, harus dikonversi ke format yang kompatibel terlebih dahulu, misalnya `str`.

Selanjutnya, pada fungsi `control_flow()` pada seleksi kondisi `mode == 1` ganti statement `pass` dengan kode baru yang meminta 3 inputan dari user kemudian menggunakannya sebagai argument pemanggilan fungsi `write_data()`.

```
def control_flow():
    while True:
        print("Choose mode:")
        print("1. Write data")
        print("2. Read data")
        print("3. Delete by row")
        print("4. Exit")

        mode = input('Choice (1/2/3/4): ')
        if mode == '1':
            name = input("Name: ")
            email = input("Email: ")
            phone = input("Phone: ")
            write_data(name, email, phone)
        elif mode == '2':
            pass # to be replaced
        elif mode == '3':
            pass # to be replaced
        elif mode == '4':
            break
        else:
            print('Invalid mode')
```

Jalankan program lalu coba test fungsionalitasnya dengan insert beberapa data. Bisa dilihat pada contoh eksekusi program berikut, file CSV berisi data sesuai operasi insert saat eksekusi program.



A.51.3. Read data CSV

Operasi baca data dilakukan via fungsi `csv.DictReader()` dengan argument parameter adalah object file. Lewat object reader, gunakan fungsi `enumerate()` untuk membungkusnya lalu lakukan iterasi menggunakan keyword `for`. Lebih jelasnya silakan lihat kode `read_data()` berikut, dan tak lupa tambahkan ke program.

```
def read_data():
    with open(filename, 'r') as csvfile:
        reader = csv.DictReader(csvfile)
        for index, row in enumerate(reader):
            print(" ->", f"(row {index + 1})", row['name'], row['email'],
row['phone'])
```

Di fungsi `control_flow()` panggil fungsi `read_data()` pada block seleksi kondisi `mode == 2`.

```
def control_flow():
    while True:
        print("Choose mode:")
        print("1. Write data")
        print("2. Read data")
        print("3. Delete by row")
        print("4. Exit")

        mode = input('Choice (1/2/3/4): ')
        if mode == '1':
            name = input("Name: ")
            email = input("Email: ")
            phone = input("Phone: ")
            write_data(name, email, phone)
        elif mode == '2':
            read_data()
        elif mode == '3':
            pass # to be replaced
        elif mode == '4':
            break
        else:
            print('Invalid mode')
```

Sekarang jalankan program untuk pengetesan.

```
PS C:\LibsSoftLink\dasarpemrogramanpython\examples\csv> python main_3.py
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 2
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 1
Name: Noval
Email: hellc@novalagung.com
Phone: 08282373
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 2
 -> (row 1) Noval hellc@novalagung.com 08282373
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 1
Name: Thrall
Email:
Phone: 083748234
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 2
 -> (row 1) Noval hellc@novalagung.com 08282373
 -> (row 2) Thrall  083748234
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4):
```

Bisa dilihat isi data yang muncul via pemanggilan fungsi `read_data()` sesuai dengan data yang di-insert.

A.51.4. Delete data CSV

Operasi delete data bisa dilakukan dengan menuliskan ulang isi file CSV tetapi meng-exclude data yang ingin dihapus. Kurang lebih kodenya seperti ini:

```
def delete_data(row_index):
    with open(filename, 'r') as csvfile:
        reader = csv.DictReader(csvfile)
        rows = list(reader)

    with open(filename, 'w', newline='') as csvfile:
        writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
        writer.writeheader()

        for index, row in enumerate(rows):
            if index != (row_index - 1):
                writer.writerow({
                    'name': row['name'],
                    'email': row['email'],
                    'phone': row['phone']
                })
```

File CSV dibaca dua kali:

- Pembacaan pertama adalah untuk pengambilan list data untuk disimpan ke variabel `rows`
- Pembacaan ke-2 adalah untuk penulisan ulang header ke file CSV dan juga data terkecuali untuk baris data yang dihapus.

Setelah itu, panggil fungsi `delete_data()` di fungsi `control_flow()`:

```
def control_flow():
    while True:
        print("Choose mode:")
        print("1. Write data")
        print("2. Read data")
        print("3. Delete by row")
        print("4. Exit")

        mode = input('Choice (1/2/3/4): ')
        if mode == '1':
            name = input("Name: ")
            email = input("Email: ")
            phone = input("Phone: ")
            write_data(name, email, phone)
        elif mode == '2':
            read_data()
        elif mode == '3':
            row_index = int(input("Row index: "))
            delete_data(row_index)
        elif mode == '4':
            break
        else:
            print('Invalid mode')
```

Jalankan ulang program, test fungsionalitas mode 1, 2, dan 3.

```
PS C:\LibsSoftLink\dasarpemrogramanpython\examples\csv> python main_4.py
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 1
Name: Noval
Email: hello@novalagung.com
Phone:
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 1
Name: Thrall
Email:
Phone: 14045
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 2
 -> (row 1) Noval hello@novalagung.com
 -> (row 2) Thrall  14045
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 3
Row index: 2
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4): 2
 -> (row 1) Noval hello@novalagung.com
Choose mode:
1. Write data
2. Read data
3. Delete by row
4. Exit
Choice (1/2/3/4):
```

Kurang lebih seperti itu penerapan operasi CSV di Python. Silakan lanjut explore dan praktek dengan pembuatan program sejenis lainnya agar makin paham.

A.51.4. `csv.writer()` dan `csv.reader()`

Selain menggunakan fungsi `csv.DictWriter()` dan `csv.DictReader()`, operasi baca tulis CSV bisa dilakukan menggunakan fungsi `csv.writer()` dan `csv.reader()`.

Caranya mirip namun ada perbedaan di bagian struktur data sewaktu menulis dan membaca data. Di 2 method sebelumnya, data perlu dibentuk dalam format dictionary. Sedangkan pada 2 fungsi ini, data ditulis dalam list.

Program berikut berisi fungsionalitas yang sama persis dengan program sebelumnya, namun menggunakan fungsi `csv.writer()` dan `csv.reader()`. Dari sini silakan pelajari perbedaannya.

```
import csv

def prepare_csv():
    with open('data.csv', 'w', newline='') as f:
        writer = csv.writer(f)
        writer.writerow(['name', 'email', 'phone'])

def write_data(name, email, phone):
    with open('data.csv', 'a', newline='') as f:
        writer = csv.writer(f)
        writer.writerow([name, email, phone])

def read_data():
    with open('data.csv', 'r') as csvfile:
        reader = csv.reader(csvfile)
        for index, row in enumerate(reader):
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../csv
```

◉ Chapter relevan lainnya

- File I/O

◉ Referensi

- https://docs.python.org/3/library/csv.html

A.52. Python JSON File/ Data Format

Pada chapter ini kita akan belajar tentang penerapan tipe data JSON di Python.

INFO

JSON atau JavaScript Object Notation adalah format data yang sangat populer digunakan dalam pembuatan aplikasi. Strukturnya sederhana dan mudah dipahami.

Contoh data JSON:

```
{
    "glossary": {
        "title": "example glossary",
        "GlossDiv": {
            "title": "S",
            "GlossList": {
                "GlossEntry": {
                    "ID": "SGML",
                    "SortAs": "SGML",
                    "GlossTerm": "Standard Generalized Markup
Language",
                    "Acronym": "SGML",
                    "Abbrev": "ISO 8879:1986",
                    "GlossDef": {
                        "para": "A meta-markup language, used to
create markup languages such as DocBook.",
                        "GlossSeeAlso": ["GML", "XML"]
                    },
                    "GlossSee": "markup"
```

Data JSON bisa tersimpan di file berekstensi `.json` atau bisa juga tersimpan di variabel sebagai data string. Contoh:

```
jstr1 = '{ "id": 2, "name": "Noval Agung Prayogo" }'

jstr2 = """
[{
    "id": 2,
    "name": "Noval Agung Prayogo"
}]
"""
```

A.52.1. Implementasi JSON di Python

Operasi JSON di Python umumnya dilakukan terhadap data yang tersimpan di variabel/memory atau data yang tersimpan di file, dengan memanfaatkan module `json` (baawan Python Standard Library) untuk proses decode maupun encode data json.

- Operasi pembacaan data JSON dari file, isi file harus dibaca terlebih dahulu secara menyeluruh, kemudian barulah di-decode ke bentuk dictionary atau slice atau lainnya (sesuai kebutuhan).
- Sedangkan operasi penulisan data JSON ke file bisa, caranya dengan menulis langsung data json dalam bentuk string ke file.

JSON (yang basisnya adalah JavaScript) memiliki kompatibilitas yang cukup bagus dengan tipe data yang tersedia di Python. Lebih jelasnya silakan lihat tabel kompatibilitas tipe data berikut:

Python data type	JSON data type
int	number
float	float
str	string
bool	boolean
list	array
tuple	array
dict	object

◉ Encode data Python ke JSON string

Operasi konversi data ke bentuk JSON string dilakukan menggunakan `json.dumps()`, tersedia di Python Standard Library.

Selama data object yang akan di-encode berisi tipe yang di-support sesuai tabel di atas, maka operasi encode ke JSON string berjalan sukses. Sedangkan jika object berisi tipe yang tidak di-support, maka harus dikonversi ke salah satu tipe data yang kompatibel terlebih dahulu (misalnya `str`) sebelum di-encode menggunakan `json.dumps()`.

Contoh operasi encode data `dict` dengan isi item `str` dan `list`:

```
import json
```

Contoh data lain dengan struktur list berisi elemen dictionary:

```
import json

data = [
    {
        'faction': 'Horde',
        'color': 'red',
        'founding_members': [
            'Orc',
            'Undead',
            'Tauren',
            'Troll',
            'Blood Elf',
            'Goblin'
        ],
        'total_members': 13,
        'active': True,
    },
    {
        'faction': 'Alliance',
        'color': 'blue',
        'founding_members': [
            'Human',
            'Dwarf',
            'Night Elf',
            'Gnome',
            'Draenei',
            'Worgen'
        ],
        'total_members': 13,
        'active': True,
    }
]
jstr = json.dumps(data, indent=4)
print(jstr)
```

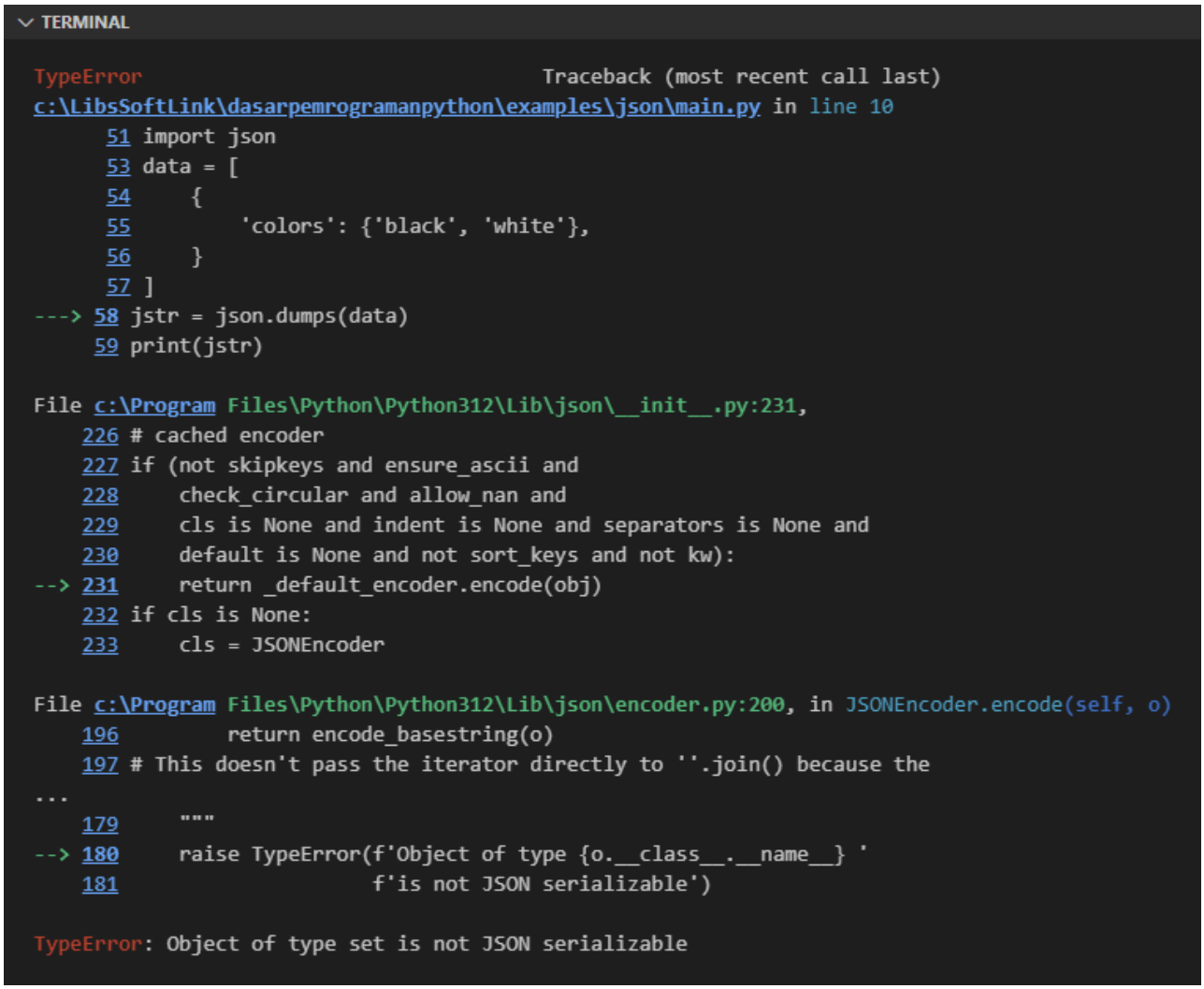
> *Pada contoh di atas, JSON string di-format dengan indent sebanyak 4. Nilai indent bisa dikonfigurasi via parameter opsional* `indent`.

Untuk data dengan isi adalah tipe data yang tidak kompatibel dengan operasi JSON di Python, penggunaannya pada `json.dumps()` menghasilkan error. Contohnya pada kode berikut ada tipe data `set` disitu.

```
import json

data = [
    {
        'colors': {'black', 'white'},
    }
]
jstr = json.dumps(data)
print(jstr)
```

Output program:



◉ Decode JSON string ke data Python

Proses decode data JSON string ke tipe data Python dilakukan menggunakan fungsi `json.loads()`. Fungsi ini mengembalikan data sesuai dengan struktur JSON string-nya (misalnya: JSON string berisi array object, maka data kembalian fungsi `json.loads()` adalah slice berisi dictionary).

```
import json

jstr1 = '{ "name": "Maiev Shadowsong", "affliations": ["Warden",
```

Contoh lain operasi decode data JSON string berisi array object:

```
jstr2 = """
[{
    "name": "Maiev Shadowsong",
    "affliations": ["Warden", "Alliance"],
    "age": 10000
}, {
    "name": "Illidan Stormrage",
    "affliations": ["Illidari", "Armies of Legionfall"],
    "age": 15000
}]
"""
data2 = json.loads(jstr2)

print(f"type: {type(data2).__name__}")
# output → type: list

for row in data2:
    print(f"-> name: {row["name"]}, afflications: {row["affliations"]},
age: {row["age"]}")
# output ↓
#
# -> name: Maiev Shadowsong, afflications: ['Warden', 'Alliance'], age:
10000
# -> name: Illidan Stormrage, afflications: ['Illidari', 'Armies of
Legionfall'], age: 15000
```

◉ Menulis data JSON ke file

Penulisan data JSON ke file sangat mudah, dilakukan menggunakan teknik penulisan standar file (kombinasi keyword `with`, fungsi `open()` dan method `write()` milik object file) dengan syarat: data yang akan ditulis ke file harus berbentuk string.

- Contoh penulisan data JSON string ke JSON file:

```
jstr = """
[{
    "name": "Maiev Shadowsong",
    "affliations": ["Warden", "Alliance"],
    "age": 10000
}, {
    "name": "Illidan Stormrage",
    "affliations": ["Illidari", "Armies of Legionfall"],
    "age": 15000
}]
"""

with open('data.json', 'w') as f:
    f.write(jstr)
```

Outcome program: file `data.json` terbuat dengan isi adalah JSON string yang ada di variabel `jstr`.

- Penulisan data Python ke JSON file dilakukan dengan mengkonversi data ke JSON string terlebih dahulu, kemudian barulah ditulis ke file. Contoh:

```
import json

data = {
    'name': 'Maiev Shadowsong',
    'affliations': ['Warden', 'Alliance']
}
jstr = json.dumps(data)

with open('data.json', 'w') as f:
    f.write(jstr)
```

Outcome program: file `data.json` terbuat dengan isi adalah JSON string hasil proses encoding `json.dumps()` terhadap data di variabel `data`.

◉ Membaca JSON file

Operasi baca JSON file dilakukan dengan membaca file seperti biasa lalu di-decode menggunakan fungsi `json.loads()`.

Contoh penerapannya bisa dilihat pada program di bawah ini. Sebelumnya, pastikan untuk menyediakan sebuah file JSON untuk keperluan testing dengan nama `data.json`. Isi file tersebut dengan data JSON string berikut:

```
{"name": "Maiev Shadowsong", "affliations": ["Warden", "Alliance"]}
```

Lalu tulis kode berikut kemudian run:

```
import json

data = []
with open('data.json', 'r') as f:
    data = json.loads(f.read())

for key in data:
    print(f"{key}: {data[key]}")
# output ↓
#
# name: Maiev Shadowsong
# affliations: ['Warden', 'Alliance']
```

Catatan chapter 🗒️

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../json
```

◉ Chapter relevan lainnya

- File I/O

◉ Referensi

- https://docs.python.org/3/library/csv.html

A.53. Python Date, Time, DateTime, Timezone

Pada chapter ini kita akan belajar tentang tipe data `date`, `time`, `datetime` untuk penyimpanan informasi tanggal dan waktu di Python, serta peran dari tipe data `timezone` untuk penentuan zona waktu.

A.53.1. Pengenalan `date`, `time`, dan `datetime`

Python menyediakan package `datetime` berisi banyak sekali API untuk keperluan operasi data yang berhubungan dengan tanggal dan waktu. Tiga diantaranya adalah tipe data berikut:

- Tipe data class `date` untuk penyimpanan informasi tanggal

```
from datetime import date

data_date = date(year=2020, month=1, day=31)

print("date:", data_date)
print(" → year:", data_date.year)
print(" → month:", data_date.month)
print(" → day:", data_date.day)
# output ↓
#
# date: 2020-01-31
#  → year: 2020
#  → month: 1
#  → day: 31
```

- Tipe data class `time` untuk penyimpanan informasi waktu (jam, menit, detik)

```
from datetime import time

data_time = time(hour=13, minute=14, second=31)

print("time:", data_time)
print(" → hour:", data_time.hour)
print(" → minute:", data_time.minute)
print(" → second:", data_time.second)
print(" → timezone:", data_time.tzinfo)
# output ↓
#
# time: 13:14:31
#  → hour: 13
#  → minute: 14
#  → second: 31
#  → timezone: None
```

- Tipe data class `datetime` untuk penyimpanan informasi tanggal dan waktu

```
from datetime import datetime

data_datetime = datetime(year=2020, month=1, day=31, hour=13,
minute=14, second=31)

print("datetime:", data_datetime)
print(" → year:", data_datetime.year)
print(" → month:", data_datetime.month)
print(" → day:", data_datetime.day)
print(" → hour:", data_datetime.hour)
print(" → minute:", data_datetime.minute)
print(" → second:", data_datetime.second)
```

Pengaksesan informasi unit waktu (seperti hari, tahun, jam, dan lainnya) dilakukan via attribute. Mengenai nama attribute-nya silakan lihat di kode di atas.

> *Tipe data `datetime` berada di package bernama sama yaitu `datetime`. Pada contoh di atas, tipe data `datetime` di-import dari package-nya.*
>
> *Contoh jika yang di-import adalah package `datetime` penggunaan tipe `datetime` seperti ini:*
> *`datetime.datetime(year=2020, month=1, day=31, hour=13, minute=14, second=31)`*
>
> *Jangan sampai bingung ya.*

Khusus untuk tipe data yang juga menyimpan informasi waktu (seperti `time` dan `datetime`), disitu ada attribute bernama `timezone` isinya informasi zona waktu. *By default*, zona waktu adalah kosong atau `None`.

◉ Combining `date` & `time`

Tipe `date` dan `time` bisa dikominasikan untuk menciptakan object baru bertipe `datetime`. Caranya via class method `combine` milik tipe data `datetime`

```
data_datetime = datetime.combine(data_date, data_time)

print("datetime:", data_datetime)
print(" → year:", data_datetime.year)
print(" → month:", data_datetime.month)
print(" → day:", data_datetime.day)
print(" → hour:", data_datetime.hour)
print(" → minute:", data_datetime.minute)
```

◉ Mengambil datetime hari ini / sekarang

Mengambil waktu hari ini dilakukan menggunakan method `today`, ini bisa diterapkan untuk tipe data `datetime` maupun `date`. Sedangkan mengambil waktu sekarang hanya bisa dilakukan pada tipe data `datetime` menggunakan method `now`. Contoh penerapannya bisa dilihat pada kode berikut.

```
data1 = datetime.now()
print("sekarang (datetime):", data1)
# output → sekarang (datetime): 2023-12-23 11:50:45.303421

data2 = datetime.today()
print("sekarang (datetime):", data2)
# output → sekarang (datetime): 2023-12-23 11:50:45.303422

data2 = date.today()
print("sekarang (date):", data2)
# output → sekarang (date): 2023-12-23
```

> *Kode di atas outputnya adalah sesuai dengan output program di local penulis saat program di run. Di tempat pembaca output pastinya berbeda.*

A.53.2. Pengenalan timezone (`tz`)

Setiap tipe data yang menyimpan informasi waktu (seperti `time` dan `datetime`) didalamnya ada informasi timezone yang bisa diakses ataupun ditentukan saat pembuatan data. Informasi timezone direpresentasikan menggunakan `dateutil.tz`, tipe data `tz` dalam package `dateutil`.

Method `tz.gettz()` digunakan untuk membuat data timezone. Contoh pengaplikasiannya:

- Membuat data timezone *New York* atau EST (Eastern Time):

```
tzinfo = tz.gettz("America/New_York")
```

- Membuat data timezone WIB (Waktu Indonesia Barat) atau GMT+7:

```
tzinfo = tz.gettz("Asia/Jakarta")
```

> *Selengkapnya mengenai list timezone identifier silakan cek di https://en.wikipedia.org/wiki/List_of_tz_database_time_zones*

Penerapan `tz` dalam pembuatan data `time` dan `timezone` dilakukan via pengisian parameter `tzinfo`. Contoh penerapannya:

- Timezone pada tipe `time`:

```
from dateutil import tz
from datetime import time

tzinfo = tz.gettz("America/New_York")
data_time = time(hour=13, minute=14, second=31, tzinfo=tzinfo)

print("time:", data_time)
# output → time: 13:14:31

print("timezone:", data_time.tzinfo)
# output → timezone: tzfile('US/Eastern')
```

- Timezone pada tipe `datetime`:

```
from dateutil import tz
from datetime import datetime

tzinfo = tz.gettz("America/New_York")
data_datetime = datetime(year=2020, month=1, day=31, hour=13,
minute=14, second=31, tzinfo=tzinfo)

print("datetime:", data_datetime)
# output → datetime: 2020-01-31 13:14:31-05:00

print("timezone:", data_time.tzinfo)
# output → timezone: tzfile('US/Eastern')
```

- Timezone pada tipe `datetime` hasil kombinasi `date` dan `time`:

```
from dateutil import tz
from datetime import date, time

tzinfo = tz.gettz("America/New_York")
data_date = date(year=2020, month=1, day=31)
data_time = time(hour=13, minute=14, second=31)
data_datetime = datetime.combine(data_date, data_time, tzinfo=tzinfo)

print("datetime:", data_datetime)
# output → datetime: 2020-01-31 13:14:31-05:00

print("timezone:", data_time.tzinfo)
# output → timezone: tzfile('US/Eastern')
```

◉ Local timezone vs. UTC vs. specific timezone vs. no timezone

Selain method `tz.gettz()` ada juga 2 method untuk keperluan timezone lainnya, yaitu:

- Method `tz.tzlocal()` untuk membuat data timezone sesuai waktu local
- Method `tz.tzutc()` untuk membuat data timezone dengan offset UTC

Contoh penerapan:

```
from dateutil import tz
from datetime import date, time, datetime

tzinfo = tz.tzlocal()
data_datetime = datetime(2020, 1, 31, 13, 14, 31, tzinfo=tzinfo)
print("datetime:", data_datetime)
# output → datetime: 2020-01-31 13:14:31+07:00

tzinfo = tz.tzutc()
data_datetime = datetime(2020, 1, 31, 13, 14, 31, tzinfo=tzinfo)
print("datetime:", data_datetime)
# output → datetime: 2020-01-31 13:14:31+00:00

tzinfo = tz.gettz("America/New_York")
data_datetime = datetime(2020, 1, 31, 13, 14, 31, tzinfo=tzinfo)
print("datetime:", data_datetime)
# output → datetime: 2020-01-31 13:14:31-05:00

data_datetime = datetime(2020, 1, 31, 13, 14, 31)
print("datetime:", data_datetime)
# output → datetime: 2020-01-31 13:14:31
```

Silakan perhatikan ke-4 output statement `print` di atas. Kesemuanya berisi informasi datetime yang sama persis namun berbeda timezone. Meskipun informasi tanggal dan waktunya sama persis, jika timezone-nya berbeda maka data tersebut menunjukkan datetime yang benar-benar berbeda.

◉ Konversi datetime antar timezone

Melanjutkan pembahasan section sebelumnya, data `2020-01-31`

`13:14:31+07:00` adalah ekuivalen dengan:

- Data `2020-01-31 06:14:31+00:00` pada zona waktu UTC
- Data `2020-01-31 01:14:31-05:00` pada zona waktu New York

Sampai sini bisa disimpulkan bahwa:

- Dua buah datetime adalah tidak sama jika informasi datetime sama persis namun timezone-nya berbeda.
- Dua buah datetime bisa saja ekuivalen atau sama meskipun informasi datetime dan timezone-nya berbeda.

Cara manual untuk mencari data `datetime` di zona waktu lain adalah dengan melakukan operasi tambah/kurang terhadap timezone-nya. Misalnya: Data `2020-01-31 13:14:31` WIB (GMT+7), kalau dikonversi ke timezone UTC caranya cukup dengan kurangi saja dengan 7 jam, hasilnya: `2020-01-31 06:14:31` UTC (GMT+0).

Atau, praktisnya bisa dengan menggunakan method `astimezone()` milik tipe data `datetime`.

```
def print_dt(d):
    print("datetime:", d, "| tz:", d.tzname())

from dateutil import tz
from datetime import date, time, datetime

data = datetime(2020, 1, 31, 13, 14, 31, tzinfo=None)
print_dt(data)
# output → datetime: 2020-01-31 13:14:31       | tz: None

data_local_tz = data.astimezone(tz.tzlocal())
print_dt(data_local_tz)
# output → datetime: 2020-01-31 13:14:31+07:00 | tz: SE Asia Standard Time
```

Penjelasan:

- Variabel `data` dibuat dengan isi datetime `2020-01-31 13:14:31` tanpa informasi timezone.
- Penerapan method `astimezone()` terhadap data tanpa timezone berefek ke penambahan timezone tanpa perubahan waktu.
    - Variabel `data_local_tz` berisi informasi datetime yang sama persis dengan `data` tapi dengan timezone adalah WIB atau GMT+7.
- Kemudian variabel `data_local_tz` dikonversi ke timezone New York dan hasilnya disimpan di variabel `data_new_york_tz`. Variabel ini berisi data yang mengarah ke suatu waktu yang sama tapi berbeda timezone.
    - Pada tanggal 31 Januari 2020, selisih jam antara GMT+7 dan New York tz (EST) adalah sebanyak `-12 jam`, maka output hasil konversi timezone data `2020-01-31 13:14:31+07:00` WIB adalah `2020-01-31 01:14:31-05:00` EST.
- Lalu variabel `data_new_york_tz` dikonversi ke UTC timezone dengan hasil disimpan di variabel `data_utc_tz`.
    - Pada tanggal 31 Januari 2020, selisih jam antara New York tz (EST) dan UTC adalah sebanyak `+5 jam`, maka output hasil konversi timezone data `2020-01-31 01:14:31-05:00` EST adalah `2020-01-31 06:14:31+00:00` UTC.

Semoga sampai sini cukup jelas.

> **INFO**

> Dalam konversi timezone, komputer membutuhkan setidaknya 2 hal agar konversi timezone terhadap data datetime menghasilkan output yang valid, yaitu:
>
> - Informasi timezone asal dan tujuan
> - Informasi tanggal timezone asal atau tujuan.
>
> Poin ke-2 sangat penting untuk diketahui karena beberapa timezone mengadopsi sistem yang didalamnya ada pergeseran jam sebanyak 2 kali dalam setahun. Contohnya seperti timezone Pacific Time (PT). Selisih atau offset timezone PT terhadap UTC di bulan Juni adalah `-7 jam` sedangkan pada bulan Desember adalah `-8 jam`. Unik bukan?
>
> Selengkapnya https://en.wikipedia.org/wiki/Pacific_Time_Zone

◉ Mengubah timezone tanpa konversi datetime

Untuk mengkonversi suatu data `datetime` ke timezone lain, gunakan method `astimezone()`. Namun untuk merubah timezone tanpa melakukan konversi waktu, maka gunakan method `replace()`. Contoh:

```
def print_dt(d):
    print("datetime:", d, "| tz:", d.tzname())

from dateutil import tz
from datetime import date, time, datetime

data = datetime(2020, 1, 31, 13, 14, 31, tzinfo=None)
print_dt(data)
# output → datetime: 2020-01-31 13:14:31       | tz: None

data_local_tz = data.replace(tzinfo=tz.tzlocal())
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../datetime-timezone
```

◉ Chapter relevan lainnya

- DateTime ➜ Parsing & Formatting

◉ TBA

- Locale

◉ Referensi

- https://docs.python.org/3/library/datetime.html
- https://stackoverflow.com/questions/3305413/
- https://realpython.com/python-datetime/
- https://en.wikipedia.org/wiki/Pacific_Time_Zone

A.54. Python DateTime Parsing & Formatting

Chapter ini merupakan lanjutan dari chapter sebelumnmya. Di sini kita akan belajar tentang teknik parsing dan formatting pada tipe data `datetime`.

A.54.1. Parsing `datetime`

Parsing adalah teknik konversi data date time berbentuk string ke tipe data `datetime`. Ada beberapa method yang tersedia untuk kebutuhan operasi ini:

◉ Via `datetime.strptime()` dan kode format

Class method `datetime.strptime()` digunakan untuk parsing string ke `datetime` dalam kombinasi kode format tertentu. Kode format sendiri merepresentasikan setiap unit waktu, misalnya:

- Kode format tahun adalah `%Y`
- Kode format jam adalah `%H`
- Kode format menit adalah `%M`
- dan lainnya.

Contoh implementasinya seperti ini:

- Contoh ke-1, parsing data `1985-04-12 23:20:50`:

```
from datetime import datetime
```

- Contoh ke-2, parsing data `1985-04-12T23:20:50.52+0700`:

```
from datetime import datetime

date_string = '1985-04-12T23:20:50.52+0700'
format_string = '%Y-%m-%dT%H:%M:%S.%f%z'

data_datetime = datetime.strptime(date_string, format_string)
print("datetime:", data_datetime)
# output → datetime: 1985-04-12 23:20:50.520000+07:00
```

Selengkapnya silakan cek di bagian A.54.3. Tabel kode format.

◉ Via `datetime.fromisoformat()` terhadap data ISO Date Time (ISO 8601)

Untuk data waktu berbentuk string sesuai spesifikasi ISO Date Time atau ISO 8601, konversi ke bentuk `datetime` bisa dilakukan secara praktis menggunakan `datetime.fromisoformat()`.

```
data_datetime = datetime.fromisoformat('1985-04-12T23:20:50.52')
print("datetime:", data_datetime)
# output → datetime: 1985-04-12 23:20:50.520000

data_datetime = datetime.fromisoformat('1985-04-12T23:20:50.52+0700')
print("datetime:", data_datetime)
# output → datetime: 1985-04-12 23:20:50.520000+07:00
```

◉ Via `datetime.fromtimestamp()` terhadap data UNIX timestamp

Data dengan format UNIX timestamp 10 digit bisa dikonversi ke bentuk

`datetime` menggunakan method `fromtimestamp()`.

```
data_datetime = datetime.fromtimestamp(1702980333)
print("datetime:", data_datetime)
# output → datetime: 2023-12-19 17:05:33
```

Untuk timestamp dengan digit lebih dari 10, informasi milisecond-nya bisa ditulis sebagai nilai floating point, misalnya:

```
data_datetime = datetime.fromtimestamp(1702980333.244)
print("datetime:", data_datetime)
# output → datetime: 2023-12-19 17:05:33.244000
```

Informasi timezone juga bisa ditentukan pada saat parsing. Caranya gunakan parameter `tz` method di atas:

```
from dateutil import tz

data_datetime = datetime.fromtimestamp(1702980333.244,
tz=tz.gettz("America/New_York"))
print("datetime:", data_datetime)
# output → datetime: 2023-12-19 05:05:33.244000-05:00
```

◉ Via `dateutil.parser.parse()`

Method `parse()` milik `dateutil.parser` cukup cerdas untuk mengenali banyak jenis format (selama format tersebut umum digunakan). Contoh penerapannya:

```
from dateutil import parser
```

Selengkapnya mengenai format yang didukung oleh parser ini bisa di cek di halaman dokumentasi Python https://dateutil.readthedocs.io/en/stable/parser.html

A.54.2. Formatting `datetime`

Formatting adalah teknik untuk memformat data `datetime` menjadi bentuk string dengan susunan/format sesuai keinginan. Beberapa method untuk operasi formatting datetime bisa dilihat di bawah ini:

◉ Method `datetime.strftime()` dan kode format

Method formatting `datetime.strftime()` tersedia pada tipe data datetime, pemanggilannya disertai dengan kode format yang diinginkan untuk output. Perihal kode format sendiri adalah sama dengan yang digunakan pada teknik parsing via `strptime()`.

Contoh penerapannya:

```
from datetime import datetime

data_datetime = datetime.fromtimestamp(1702980333, tz=None)
print(data_datetime.strftime('%Y-%m-%dT%H:%M:%S.%f%z'))
# output → 2023-12-19T17:05:33.000000

data_datetime = datetime.fromisoformat('1985-04-12T23:20:50.52+0700')
print(data_datetime.strftime('%m/%d/%Y %H:%M:%S %z'))
# output → 04/12/1985 23:20:50 +0700
```

Selengkapnya silakan cek di bagian A.54.3. Tabel kode format.

◉ Method `datetime.isoformat()` → ISO Date Time (ISO

8601)

Method ini menghasilkan output berbentuk string sesuai format ISO Date Time (ISO 8601). Contoh penggunaan:

```
data_datetime = datetime.now()
print(data_datetime.isoformat())
# output → 2023-12-23T14:01:03.088274
```

◉ Method `datetime.timestamp()` → UNIX timestamp

Method ini menghasilkan output berbentuk UNIX timestamp bertipe float. Angka dibelakang koma merepresentasikan miliseconds. Contoh penggunaan:

```
data_datetime = datetime.now()
print(data_datetime.timestamp())
# output → 1703314863.975431
```

A.54.3. Tabel kode format

Tabel kode format sesuai 1989 C standard:

Kode format	Penjelasan	Contoh
`%a`	Weekday as locale’s abbreviated name	Sun, Mon, …, Sat (en_US); So, Mo, …, Sa (de_DE)

Kode format	Penjelasan	Contoh
`%A`	Weekday as locale’s full name	Sunday, Monday, ..., Saturday (en_US); Sonntag, Montag, ..., Samstag (de_DE)
`%w`	Weekday as a decimal number, where 0 is Sunday and 6 is Saturday	0, 1, ..., 6
`%d`	Day of the month as a zero-padded decimal number	01, 02, ..., 31
`%b`	Month as locale’s abbreviated name	Jan, Feb, ..., Dec (en_US)
`%B`	Month as locale’s full name	January, February, ..., December (en_US)
`%m`	Month as a zero-padded decimal number	01, 02, ..., 12
`%y`	Year without century as a zero-padded decimal number	00, 01, ..., 99

Kode format	Penjelasan	Contoh
`%Y`	Year with century as a decimal number	0001, 0002, ..., 2013, 2014, ..., 9998, 9999
`%H`	Hour (24-hour clock) as a zero-padded decimal number	00, 01, ..., 23
`%I`	Hour (12-hour clock) as a zero-padded decimal number	01, 02, ..., 12
`%p`	Locale’s equivalent of either AM or PM	AM, PM (en_US)
`%M`	Minute as a zero-padded decimal number	00, 01, ..., 59
`%S`	Second as a zero-padded decimal number	00, 01, ..., 59
`%f`	Microsecond as a decimal number, zero-padded to 6 digits	000000, 000001, ..., 999999
`%z`	UTC offset in the form ±HHMM[SS[.ffffff]] (empty string if the object is naive)	(empty), +0000, -0400, +1030, +063415, -030712.345216

Kode format	Penjelasan	Contoh
`%Z`	Time zone name (empty string if the object is naive)	(empty), UTC, GMT
`%j`	Day of the year as a zero-padded decimal number	001, 002, ..., 366
`%U`	Week number of the year (Sunday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Sunday are considered to be in week 0	00, 01, ..., 53
`%W`	Week number of the year (Monday as the first day of the week) as a zero-padded decimal number. All days in a new year preceding the first Monday are considered to be in week 0	00, 01, ..., 53
`%c`	Locale’s appropriate date and time representation	Tue Aug 16 21:30:00 1988 (en_US)
`%x`	Locale’s appropriate date representation	08/16/88 (None); 08/16/1988 (en_US); 16.08.1988 (de_DE)

Kode format	Penjelasan	Contoh
`%X`	Locale’s appropriate time representation	21:30:00 (en_US); 21:30:00 (de_DE)
`%%`	A literal '%' character	%

Tabel kode format sesuai C89 standard:

Kode format	Penjelasan	Contoh
`%G`	ISO 8601 year with century representing the year that contains the greater part of the ISO week (%V)	0001, 0002, ..., 2013, 2014, ..., 9998, 9999
`%u`	ISO 8601 weekday as a decimal number where 1 is Monday	1, 2, ..., 7
`%V`	ISO 8601 week as a decimal number with Monday as the first day of the week. Week 01 is the week containing Jan 4	01, 02, ..., 53
`%:z`	UTC offset in the form ±HH:MM[:SS[.ffffff]] (empty string if the object is naive)	(empty), +00:00, -04:00, +10:30, +06:34:15,

Kode format	Penjelasan	Contoh
		-03:07:12.345216

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../datetime-parsing-formatting
```

◉ Chapter relevan lainnya

- Date, Time, DateTime, Timezone

◉ TBA

- Locale

◉ Referensi

- https://docs.python.org/3/library/datetime.html
- https://dateutil.readthedocs.io/en/stable/parser.html
- https://stackoverflow.com/questions/3305413/how-to-preserve-timezone-when-parsing-date-time-strings-with-strptime
- https://stackoverflow.com/questions/13182075/how-to-convert-a-timezone-aware-string-to-datetime-in-python-without-dateutil
- https://stackoverflow.com/questions/1941927/convert-an-rfc-3339-time-to-a-standard-python-timestamp

- https://realpython.com/python-datetime/

A.55. Python Delete Object

Pada sekian chapter yang lalu kita telah belajar tentang object None, fungsinya untuk menandai variabel agar bernilai kosong.

Kali ini yang kita pelajari adalah keyword `del` untuk operasi delete object. Hasil penggunaan keyword `del` benar-benar menghapus variabel beserta isinya, jadi tidak hanya mengosongkan nilainya.

A.55.1. Keyword `del`

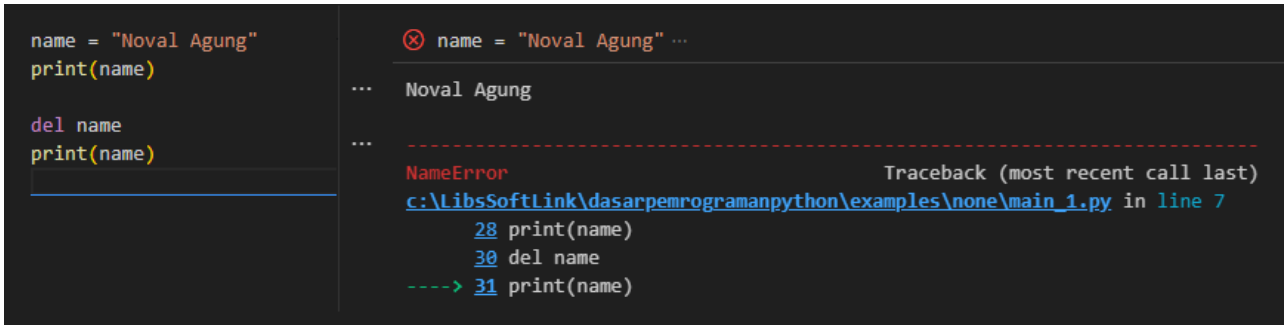
Penggunaan `None` sebagai nilai suatu variabel menjadikan variabel tersebut tetap ada tapi nilainya berubah menjadi `None`. Jadi penggunaan `None` disini berguna untuk mengosongkan variabel tanpa menghapus variabel itu sendiri.

Bagaimana jika kita ingin benar-benar menghapus suatu variabel? Caranya menggunakan keyword `del`. Contoh penggunaannya bisa dilihat di bawah ini:

```
name = "Noval Agung"
print(name)

del name
print(name)
```

Operasi print pertama memunculkan nilai variabel `name` tanpa kendala. Namun di statement print ke-2, variabel `name` tidak dikenal dan menghasilkan error karena variabel tersebut telah dihapus menggunakan keyword `del`.



Jadi keyword `del` ini benar-benar menghapus variabel ya.

Tak hanya ke variabel saja, keyword ini bisa digunakan untuk menghapus attribute, item list, class, fungsi, dan banyak lainnya!

◉ Delete list item

```
obj = ["Noval", "Malang", "Chad"]
print(obj)
# output → ['Noval', 'Malang', 'Chad']

del obj[1]
print(obj)
# output → ['Noval', 'Chad']
```

◉ Delete dictionary item

```
obj = {
    "name": "Noval",
    "city": "Malang",
    "gender": "Chad"
}
print(obj)
# output → {'name': 'Noval', 'city': 'Malang', 'gender': 'Chad'}

del obj["city"]
```

◉ Delete class/object property

Keyword `del` dalam penggunaannya terhadap property class/object membuat property tersebut menjadi tidak ada.

Pada contoh berikut statement delete dibungkus block `try` untuk menangkap error yang muncul karena mencoba mengakses property yang tidak terdaftar.

```
class Person:
    def __init__(self, name, city, gender):
        self.name = name
        self.city = city
        self.gender = gender

p = Person("Noval", "Malang", "Chad")
print(p.name, p.city, p.gender)
# output → Noval Malang Chad

try:
    del p.city
    print(p.name, p.city, p.gender)
    # raise exception ⬇

except Exception as err:
    print(err)
    # output → 'Person' object has no attribute 'city'
```

◉ Delete function

Tak hanya object, fungsi juga bisa dihapus.

```
def say_hello():
    print("hello world")
```

◉ Delete class

```
class Person:
    def __init__(self, name, city, gender):
        self.name = name
        self.city = city
        self.gender = gender

p = Person("Noval", "Malang", "Chad")
print(p.name, p.city, p.gender)
# output → Noval Malang Chad

try:
    del Person
    p = Person("Noval", "Malang", "Chad")
    # raise exception ⬇

except Exception as err:
    print(err)
    # output → name 'Person' is not defined
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../del
```

◉ Chapter relevan lainnya

- None

◉ Referensi

- https://docs.python.org/3/tutorial/datastructures.html#the-del-statement
- https://docs.python.org/3/reference/simple_stmts.html#del

A.56. Python Enumeration

Enumeration (Enum) adalah nilai konstan, umumnya disiapkan untuk merepresentasikan sekumpulan data konstan yang konteksnya masih sama. Misalnya, **warna** yang isinya bisa merah, biru, kuning, atau warna lain.

Pembuatan enum di Python sangat mudah. Caranya dengan mendeklarasikan class baru yang meng-*inherit* class `enum.Enum`, kemudian menuliskan kumpulan data sebagai class property-nya.

A.56.1. Enum / Enumeration

Di bawah ini, disiapkan sebuah enum bernama `City` dengan isi ada 4 buah pilihan kota.

```
from enum import Enum
class City(Enum):
    MALANG = 1
    SURABAYA = 2
    YOGYAKARTA = 3
    JAKARTA = 4

print(list(City))
# output → [<City.MALANG: 1>, <City.SURABAYA: 2>, <City.YOGYAKARTA: 3>,
<City.JAKARTA: 4>
```

Nilai property enum bisa diisi dengan data apapun. Pada contoh di atas, nilai property enum `City` diisi dengan angka numerik.

> *Property enum bisa diakses dalam bentuk `list` dengan cukup*

> *membungkusnya menggunakan fungsi* `list()`.

Selanjutnya, buat satu variabel, isi nilainya dengan salah satu property enum (misalnya `City.YOGYAKARTA`), kemudian print.

```
city1 = City.YOGYAKARTA

print(city1)
# output → City.YOGYAKARTA

print(f"name {city1.name}")
# output → name YOGYAKARTA

print(f"value {city1.value}")
# output → value 3
```

Nama enum property bisa diambil menggunakan property `name` sedangkan value-nya diambil via property `value`.

A.56.2. Naming convention enum

Sesuai penjelasan di halaman dokumentasi Python, nama class enum dianjurkan ditulis menggunakan **CamelCase**, sedangkan nama class property dituliskan **UPPERCASE**.

A.56.3. Notasi penulisan pengaksesan enum property

Dalam penggunaannya setidaknya ada 3 notasi pengaksesan enum property yang bisa digunakan.

- Cara pertama menggunakan notasi `Enum.PROPERTY`.

```
city1 = City.YOGYAKARTA

print(city1)
# output → City.YOGYAKARTA
print(f"name {city1.name}")
# output → name YOGYAKARTA
print(f"value {city1.value}")
# output → value 3
```

- Cara ke-2 menggunakan notasi pengaksesan dictionary dengan key berupa string.

```
city2 = City["SURABAYA"]
print(city2)
# output → City.SURABAYA
print(f"name {city2.name}")
# output → name SURABAYA
print(f"value {city2.value}")
# output → value 2
```

  Metode ini cukup berguna pada case ketika key enum yang perlu diambil yang sudah diketahui nilainya.

- Cara ke-3 menggunakan notasi pengaksesan `Enum(value)`.

```
city3 = City(4)
print(city3)
# output → City.JAKARTA
print(f"name {city3.name}")
# output → name JAKARTA
print(f"value {city3.value}")
```

Metode ini pas digunakan ketika value enum sudah diketahui nilainya.

A.56.4. Nilai property enum

Property enum nilainya bisa berisi data numerik, string, atau tipe data lainnya.

Di atas telah dicontohkan enum `City` yang property-nya berisi data numerik. Pada contoh ke-2 ini, enum `Color` property-nya berisi data string.

```
from enum import Enum
class Color(Enum):
    RED = "red"
    BLUE = "blue"

print(list(Color))
# output → [<Color.RED: 'red'>, <Color.BLUE: 'blue'>]

color1 = Color("red")
color2 = Color.RED
color3 = Color["BLUE"]
print(color1, color2, color3)
# output → Color.RED Color.RED Color.BLUE
```

Selain menggunakan class `enum.Enum` saat deklarasi, bisa juga menggunakan salah satu class turunan enum lainnya, diantaranya ada:

- Class `enum.Enum` ➡ property boleh berisi data bertipe apapun
- Class `enum.StrEnum` ➡ property hanya boleh berisi tipe data string
- Class `enum.IntEnum` ➡ property hanya boleh berisi tipe data integer

Cara penerapan variant enum yang telah disebut di atas adalah masih sama. Cukup ganti `Enum` dengan turunan class yang diinginkan, lalu sesuaikan tipe

data nilai property di dalamnya.

◉ `StrEnum`

Misalnya, enum `Color` kita ubah dari menggunakan `Enum` ke `StrEnum`, maka kodenya seperti ini:

```
from enum import StrEnum
class Color(StrEnum):
    RED = "red"
    BLUE = "blue"

print(list(Color))
# output → [<Color.RED: 'red'>, <Color.BLUE: 'blue'>]

color1 = Color("red")
color2 = Color.RED
color3 = Color["BLUE"]
print(color1, color2, color3)
# output → Color.RED Color.RED Color.BLUE
```

◉ `IntEnum` dan fungsi `auto()`

Untuk enum dengan property bertipe numerik, bisa menggunakan fungsi `auto()` untuk generate nilai numerik unik secara otomatis. Contoh:

```
from enum import IntEnum, auto
class Size(IntEnum):
    S = auto()
    M = auto()
    L = auto()
    XL = auto()

print(list(Size))
```

A.56.5. Pengecekan nilai enum

Operator identitas `is` dan operator perbandingan `==` dan `!=` bisa digunakan untuk pengecekan nilai enum.

```
from enum import Enum
class City(Enum):
    MALANG = 1
    SURABAYA = 2
    YOGYAKARTA = 3
    JAKARTA = 4

def say_anything(c):
    if c is City.MALANG:
        print("Oyi sam")
    elif c == City.JAKARTA:
        print("lo gue lo gue")
    elif c == City.SURABAYA:
        print("coeg")
    elif c == City.YOGYAKARTA:
        print("monggo")

city1 = City.YOGYAKARTA
say_anything(city1)
# output → monggo
```

A.56.6. Perulangan enum

Enum merupakan tipe yang bisa langsung digunakan pada perulangan. Contoh penggunaannya:

```
for c in City:
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../enum
```

◉ Chapter relevan lainnya

- Konstanta

◉ TBA

- `IntFlag` and `Flag` https://docs.python.org/3/howto/enum.html#intflag

◉ Referensi

- https://docs.python.org/3/library/enum.html
- https://docs.python.org/3/howto/enum.html

A.57. Python Eval & Exec

Pada chapter ini kita akan mempelajari fungsi `eval()` untuk eksekusi *expression*, fungsi `exec()` untuk eksekusi syntax Python, dan fungsi `compile()`.

A.57.1. Fungsi `eval()`

String berisi ekspresi seperti `PI * r * r` bisa dieksekusi kemudian diambil hasilnya menggunakan fungsi `eval()`. Cara penggunaannya sangat mudah, tulis saja ekspresi sebagai argument pemanggilan fungsi. Misalnya:

```
a = 10
b = 5
c = 8
area = eval('(a + b) / 20 + (c * c)')
print(area)
# output → 64.75
```

Variabel `a`, `b`, dan `c` di ekspresi `'(a + b) / 20 + (c * c)'` nilainya terisi oleh nilai variabel `a`, `b`, dan `c` yang sudah dideklarasikan sebelumnya.

Ekspresi string berisi apapun bisa digunakan pada fungsi `eval()` ini, asalkan ekspresinya hanya 1 baris.

Jika ekspresi tidak valid, maka eksekusi fungsi `eval()` menghasilkan error, dan ketika itu terjadi ada baiknya di-handle dengan baik. Contoh:

```
try:
    area = eval('(a + b asd / 20 + (c * c)')
```

By default, variabel dalam ekspresi terisi oleh nilai variabel yang sudah dideklarasikan. Selain itu, nilai variabel juga bisa ditentukan secara eksplisit via parameter ke-2 atau ke-3 fungsi `eval()`.

- Jika ekspresi berisi variabel yang kesemuanya ditentukan secara eksplisit, bisa gunakan parameter ke-2.

```
area = eval('PI * r * r', { "PI": 3.14, "r": 10 })
print(area)
# output → 314.0
```

- Jika eskresi berisi variabel yang hanya sebagaian saja yang nilainya ditentukan secara eksplisit, maka gunakan `globals()` sebagai argument parameter ke-2 dan tulis variabel pada parameter ke-3.

```
PI = 3.14
area = eval('PI * r * r', globals(), { "r": 10 })
print(area)
# output → 314.0
```

> *Pemanggilan `eval('expr')` tanpa parameter ke-2 dan ke-3 adalah ekuivalen dengan statement `eval('expr', globals(), locals())`*

A.57.2. Fungsi `exec()`

Fungsi `exec()` berguna untuk eksekusi string berisi kode Python. Cara penggunaannya mirip seperti `eval()`. Contoh:

- Eksekusi statement dengan operasi *assignment*

```
a = 10
b = 5
c = 8
res = 0
exec('res = (a + b) / 20 + (c * c)')
print(res)
# output → 64.75
```

Eksekusi `(a + b) / 20 + (c * c)` nilainya disimpan ke variabel `res` yang efeknya mengubah nilai variabel `res` yang telah dideklarasikan.

- Eksekusi perulangan

```
r = 4
stmt = """
for x in range(r):
    print(x)
"""
exec(stmt)
# output ↓
#
# 0
# 1
# 2
# 3
```

- Eksekusi kode berisi statement import dan fungsi

```
r = 10
res = 0
stmt = """
from typing import Final

PI: Final = 3.14
```

Selama string berisi kode Python dengan syntax valid, maka bisa dijalankan via fungsi `exec()` ini.

A.57.3. Fungsi `compile()`

Fungsi `compile()` digunakan untuk kompilasi expression maupun kode Python, untuk kemudian digunakan pada fungsi `eval()`, `exec()`, ataupun untuk keperluan lainnya (seperti parse AST dan lainnya).

Fungsi ini memiliki 3 parameter yang harus diisi saat pemanggilan:

- Parameter ke-1 diisi dengan nilai string *expression* atau kode Python
- Parameter ke-2 diisi string `'<string>'`
- Parameter ke-3 diisi `eval` atau `exec`

Dengan mengkompilasi string terlebih dahulu, kita bisa mengantisipasi syntax error ataupun expression error yang ada dalam string tanpa harus mengeksekusinya terlebih dahulu.

Pengaplikasian fungsi `compile()` dalam penggunaannya dibedakan berdasarkan isi string yang di-compile.

◉ Compile *expression*

Penggunaan fungsi `compile()` untuk *expression*, parameter ke-3 perlu diisi nilai `'eval'`.

```
compiled = compile('expr', '<string>', 'eval')
res = eval(compiled)
```

Contoh:

```
a = 10
b = 5
c = 8
compiled = compile('(a + b) / 20 + (c * c)', '<string>', 'eval')
area = eval(compiled)
print(area)
# output → 64.75
```

◉ Compile *Python source code*

Sedangkan untuk eksekusi kode Python, parameter ke-3 diisi nilai `'exec'`.

```
compiled = compile(stmt, '<string>', 'exec')
res = exec(compiled)
```

Contoh:

```
r = 10
res = 0
stmt = """
from typing import Final

PI: Final = 3.14

def calculate_area_of_circle():
    print(f"calculating area of circle with r: {r}")
    return PI * r * r

res = calculate_area_of_circle()
"""
compiled = compile(stmt, '<string>', 'exec')
area = exec(compiled)
print(res)
# output ↓
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../eval-exec
```

◉ Referensi

- https://stackoverflow.com/questions/2220699/whats-the-difference-between-eval-exec-and-compile

A.58. Python Iterable & Iterator

Pada chapter ini kita akan belajar tentang konsep sekaligus penerapan iterable dan iterator di Python.

A.58.1. Apa itu iterable?

Iteration atau iterasi adalah proses pengaksesan item-item suatu data kolektif sesuai urutannya satu-per-satu (menggunakan metode perulangan). Penerapan keyword `for in` pada data string, list, atau data kolektif lainnya adalah contoh dari iterasi.

```
condition = "weteng luwe"
for c in condition:
    print(c)

numbers = [10, 12, 32, 44]
for n in numbers:
    print(n)
```

Di atas, variabel `condition` di-iterasi menggunakan keyword `for in`, variabel `c` disetiap perulangan terisi dengan huruf string satu-per-satu, bergantian, mulai huruf pertama hingga terakhir.

Begitu juga dengan variabel `numbers`, nilai numerik dalam list di-iterasi dengan setiap nilai ditampung variabel `n` untuk kemudian di-print.

Iterable adalah object yang bisa di-iterasi, contohnya seperti tuple, list, string,

dan tipe data kolektif lainnya. Sederhananya, setiap object yang bisa digunakan pada keyword `for in` adalah pasti iterable.

A.58.2. Apa itu iterator?

Iterator berbeda dibanding iterable. Iterator adalah object iterable yang bisa mengingat *state* perulangannya. Jadi object tersebut tau informasi seperti perulangan sedang berada di indeks ke berapa, elemen berikutnya apa, dan info mengenai kapan perulangan berhenti.

Object iterable bisa dikonversi menjadi iterator, caranya dengan membungkusnya menggunakan fungsi `iter()`. Contoh:

```
condition = "weteng luwe"
for c in iter(condition):
    print(c)

numbers = [10, 12, 32, 44]
for n in iter(numbers):
    print(n)
```

Ok, lalu apa benefitnya? apa bedanya dengan iterable biasa? Jika mengacu ke contoh di atas, tidak akan terlihat perbedaannya dimana, karena object iterable dan iterator keduanya pasti bisa digunakan pada perulangan menggunakan keyword `for in`.

Agar lebih jelas, kita berkenalan dulu dengan fungsi baru bernama `next()`. Fungsi ini berguna untuk mengembalikan data iterasi berikutnya. Contoh penerapannya silakan lihat kode berikut ini:

```
numbers = [10, 12, 32, 44]
```

Data list `numbers` dikonversi ke bentuk iterator menggunakan fungsi `iter()`, kemudian digunakan sebagai argument pemanggilan fungsi `next()` 4 kali, hasilnya:

- Pemanggilan ke-1 mengembalikan item index ke-0, yaitu angka `10`
- Pemanggilan ke-2 mengembalikan item index ke-1, yaitu angka `12`
- Pemanggilan ke-3 mengembalikan item index ke-2, yaitu angka `32`
- Pemanggilan ke-4 mengembalikan item index ke-3, yaitu angka `44`

Dari sini terlihat bahwa object iterator benar-benar mengingat informasi state-nya. Setelah pemanggilan fungsi `next()` pertama, object `numbers_iter` tau bahwa item indeks ke-1 sudah diakses, dan pengaksesan berikutnya adalah indeks ke-2, dan seterusnya.

> *Penggunaan fungsi `next()` terhadap object iterator yang sudah terakses semua itemnya menghasilkan error. Manfaatkan keyword `try except` jika diperlukan.*

A.58.3. Custom iterator

Selain menggunakan fungsi `iter()` pembuatan iterator juga bisa dilakukan via custom class. Caranya dengan mendesain custom class agar memiliki 2 hal berikut:

1. Custom class harus *inherit* class `Iterator` milik module `collections.abc`.
2. Di dalam class harus ada fungsi `__next__(self)` yang otomatis dipanggil setiap kali object digunakan pada fungsi `next()`.

Pada contoh berikut, disiapkan class bernama `LoopReverse` dengan tugas

adalah menghasilkan object iterator yang mengembalikan item data kolektif berurutan dari belakang (terbalik).

```
from collections.abc import Iterator

class LoopReverse(Iterator):

    def __init__(self, data):
        self.data = data
        self.i = 0

    def __next__(self):
        if (self.i+1) > len(self.data):
            raise StopIteration
        else:
            r = self.data[len(self.data)-self.i-1]
            self.i = self.i + 1
            return r

numbers = [23, 2, 1, 54]
for n in LoopReverse(numbers):
    print(n)

# output ↓
#
# 54
# 1
# 2
# 23
```

Penjelasan:

- Sewaktu class dinisialisasi, instance attribute `i` disiapkan dengan nilai `0` dan attribute `data` berisi data yang disisipkan via argument (pada konteks ini yang dimaksud adalah data `[23, 2, 1, 54]`).
- Di setiap perulangan, fungsi `__next__(self)` dipanggil, ketika nilai `i`

masih di bawah jumlah item `data`, maka item nomor `i` dari belakang dikembalikan. Selebihnya, exception `StopIteration` di-raise untuk menandai bahwa iterasi sudah dihentikan.

Coba test juga pengaksesan item iterator via fungsi `next()`. Contoh:

```
numbers = [23, 2, 1, 54]
numbers_iter = LoopReverse(numbers)

n = next(numbers_iter)
print(n) # output → 54

n = next(numbers_iter)
print(n) # output → 1

n = next(numbers_iter)
print(n) # output → 2

n = next(numbers_iter)
print(n) # output → 23
```

Selain menggunakan class `Iterator` sebagai superclass, iterator custom class juga bisa dibuat dengan cukup menambahkan fungsi `__iter__(self)` yang mengembalikan data `self` tanpa perlu meng-inherit class `Iterator`. Contoh:

```
class LoopReverse:

    def __init__(self, data):
        self.data = data
        self.i = 0

    def __iter__(self):
        return self
```

Method `__iter__(self)` harus mengembalikan object iterator yang dimana harus memiliki method `__next__(self)`. Pada contoh di atas, method `__next__(self)` kebetulan memang posisinya satu level dengan method `__iter__(self)`, oleh karena itu object `self` dijadikan nilai balik.

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../iterable-iterator
```

◉ Chapter relevan lainnya

- Perulangan ➜ for & range

◉ Referensi

- https://docs.python.org/3/glossary.html#term-iterator
- https://docs.python.org/3/library/functions.html#iter
- https://docs.python.org/3/library/functions.html#next

A.59. Python Generator & Yield

Pada chapter ini kita akan belajar tentang konsep generator serta pengaplikasiannya menggunakan keyword `yield` dan *generator expression*.

A.59.1. Generator object

Generator object adalah object iterator yang dibuat via fungsi generator (menggunakan keyword `yield`) atau via *generator expression*.

> *Mengenai apa itu iterator, pembahasannya ada di chapter sebelumnya, yaitu Iterable & Iterator*

A.59.2. Generator function

Generator function adalah salah satu cara pembuatan generator object. Ciri khasnya di dalam fungsi ada statement berisi keyword `yield`.

Mari praktekan. Di bawah ini, fungsi `random_messages()` didesain untuk mengembalikan generator object.

```
def random_messages():
    yield "hello python"
    yield "how are you"
    yield "nice to meet you"
```

Variabel `messages` menampung nilai balik pemanggilan fungsi `random_messages()` (data generator object), ketika di-print nilainya yang muncul adalah *generator object*.

Selayaknya iterator, generator (yang juga merupakan iterator) pengaksesan itemnya dilakukan via perulangan `for in`, atau menggunakan fungsi `next()`.

Pada contoh berikutnya ini, generator `random_messages()` digunakan pada perulangan. Hasilnya, di setiap iterasi `for in`, data yang dikembalikan via keyword `yield` di-print.

```
def random_messages():
    yield "hello python"
    yield "how are you"
    yield "nice to meet you"

for message in random_messages():
    print(message)
# output ↓
#
# hello python
# how are you
# nice to meet you
```

Karena ada 3 statement `yield` disitu, maka generator ketika digunakan di perulangan hasilnya iterasi sebanyak 3x.

- Iterasi ke-1, variabel `message` berisi statement yield ke-1, yaitu `hello python`
- Iterasi ke-2, variabel `message` berisi statement yield ke-2, yaitu `how are you`
- Iterasi ke-3, variabel `message` berisi statement yield ke-3, yaitu `nice to`

`meet you`

Contoh implementasi penerapan generator via fungsi `next()`:

```
gen = random_messages()

message = next(gen)
print(f"message 1: {message}")
# output → message 1: hello python

message = next(gen)
print(f"message 2: {message}")
# output → message 2: how are you

message = next(gen)
print(f"message 3: {message}")
# output → message 3: nice to meet you
```

Berikut adalah 2 contoh tambahan penerapan fungsi generator dan keyword `yield`.

◉ Infinite counter

Keyword `yield` bisa dikombinasikan dengan perulangan `while` untuk membuat logic *infinite counter*.

```
def infinite_counter():
    i = 0
    while True:
        yield i
        i = i + 1

c = infinite_counter()
print(next(c))
```

◉ Operasi baca file

Contoh operasi baca file yang setiap baris konten file dikembalikan sebagai data iterasi via keyword `yield`.

```
def file_reader(file_name):
    for row in open(file_name, "r"):
        yield row

for row in open('content.txt'):
    print(row.strip())
```

A.59.3. Generator expression

Generator comprehension (atau yang umumnya dikenal sebagai generator expression) adalah ekspresi statement yang ditulis dalam format tertentu yang otomatis terdeteksi sebagai *generator object*.

Penulisan ekspresi generator mirip seperti penulisan *list comprehension*, perbedaannya ada pada karakter yang digunakan untuk membungkus ekspresi. Pada list comprehension menggunakan tanda `[ ]`, pada generator comprehension/expression menggunakan tanda `( )`.

Contoh penerapan:

```
exp1 = [num**2 for num in range(5)]
print(exp1)
# output → [0, 1, 4, 9, 16]

exp2 = (num**2 for num in range(5))
print(exp2)
```

Catatan chapter 📑

◉ Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/../generator-yield
```

◉ Chapter relevan lainnya

- List Comprehension
- Iterable & Iterator

◉ Referensi

- https://docs.python.org/3/reference/expressions.html#generator-expressions
- https://docs.python.org/3/reference/expressions.html#yield-expressions