



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.

E-book 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 disediakan juga pembahasan *advance* lainnya, **stay tuned!**

Versi e-book: **v1.0.0-beta1.20231011**, dan versi **Python 3.11.3**.

E-book ini aktif dalam pengembangan, kami akan tambah terus konten-kontennya. Silakan cek di [Github repo](#) kami mengenai progress development e-book.

Download Ebook File (pdf)

Ebook ini bisa di-download dalam bentuk file, silakan gunakan link berikut:

[Dasar Pemrograman Python.pdf](#)

Source Code Praktik

Source code contoh program bisa diunduh di 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 Rust.

Kontribusi

Ebook ini merupakan project open source, teruntuk siapapun yang ingin berkontribusi silakan langsung saja cek github.com/novalagung/dasarpemrogramanpython. Cek juga [halaman kontributor](#) untuk melihat list kontributor.

Lisensi dan Status FOSSA

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

[FOSSA Status](#)

Author & Maintainer

Ebook ini dibuat oleh Noval Agung Prayogo. Untuk pertanyaan, kritik, dan saran, silakan drop email ke [\[email protected\]](#).

Author & Contributors

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 teman-teman 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. ... anda :-)
-

Download versi PDF

Ebook ini bisa di-download dalam bentuk file, 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 Status

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

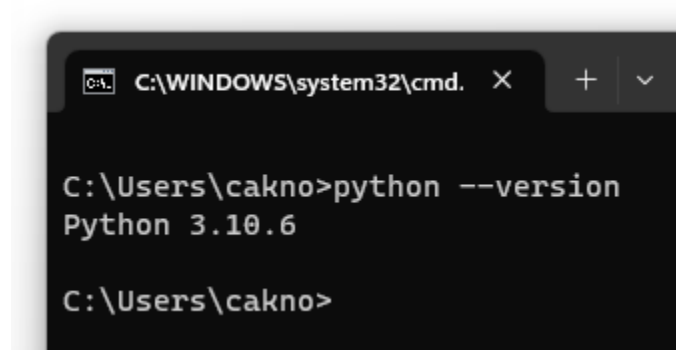
- Tarball source code bisa diunduh di [situs official Python](#)

● 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.



```
C:\WINDOWS\system32\cmd. X + v
C:\Users\cakno>python --version
Python 3.10.6
C:\Users\cakno>
```


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:

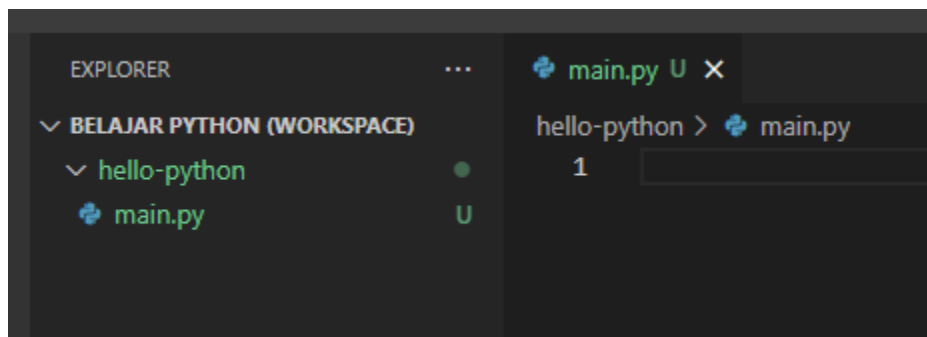
- Extension [Python](#), untuk mendapatkan benefit API doc, autocompletion, linting, run feature, dan lainnya.
- Extension [Jupyter](#), untuk interactive run program via editor.

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`.

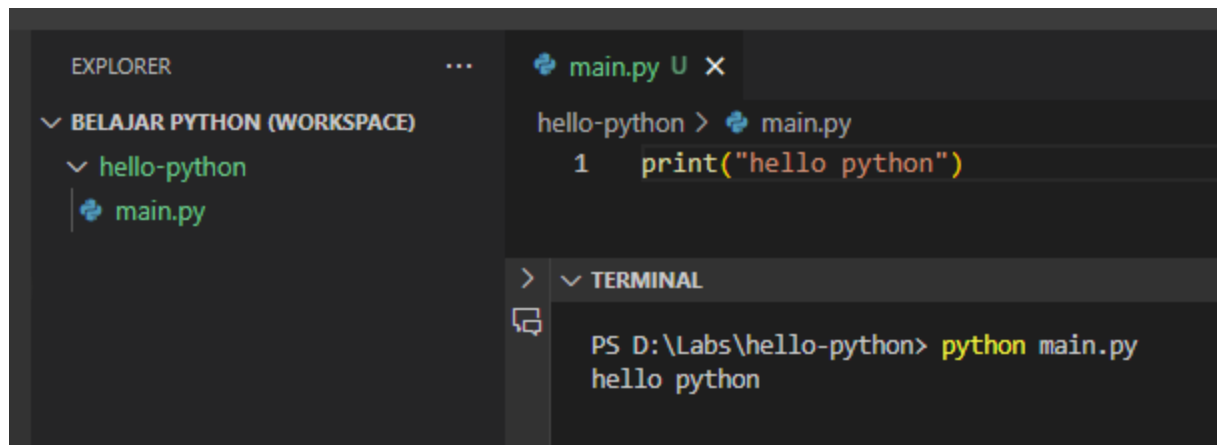


Pada file `main.py`, tuliskan kode berikut:

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

Run program menggunakan command berikut:

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

A screenshot of a code editor interface. On the left, the 'EXPLORER' panel shows a workspace named 'BELAJAR PYTHON (WORKSPACE)' containing a folder 'hello-python' with a file 'main.py'. The main editor area shows the code in 'main.py':

```
hello-python > main.py
1 print("hello python")
```

Below the code editor is a 'TERMINAL' panel showing the command prompt output:

```
PS D:\Labs\hello-python> python main.py
hello python
```

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 disiapkan pada argument pemanggilan fungsi `print()`). Pesan tersebut akan muncul ke layar output stdout (pada contoh ini adalah terminal milik editor

penulis).

- Lebih detailnya mengenai fungsi dibahas pada chapter *Fungsi*
- Lebih detailnya mengenai Python standard library (stdlib) dibahas terpisah pada chapter *Python standard library (stdlib)*

Catatan chapter

● Source code praktik

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

● Referensi

- [https://www.learnpython.org/en/Hello,_World!](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

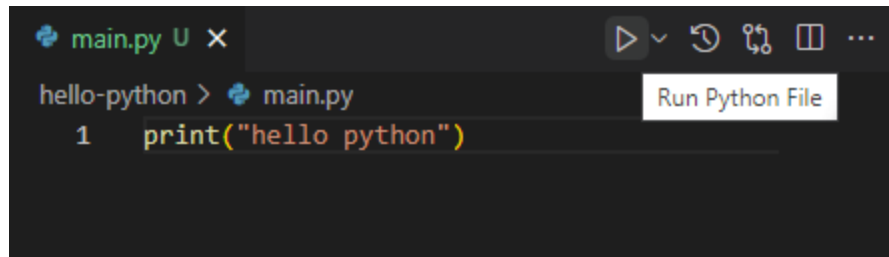
● Menggunakan command `python`

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

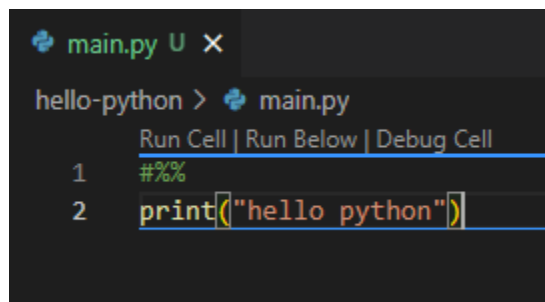
Cara run program ini lebih praktis karena tinggal klik-klik saja. Di toolbar VSCode sebelah kanan atas ada tombol , gunakan tombol tersebut untuk menjalankan program.



```
main.py U x
hello-python > main.py
1 print("hello python")
```

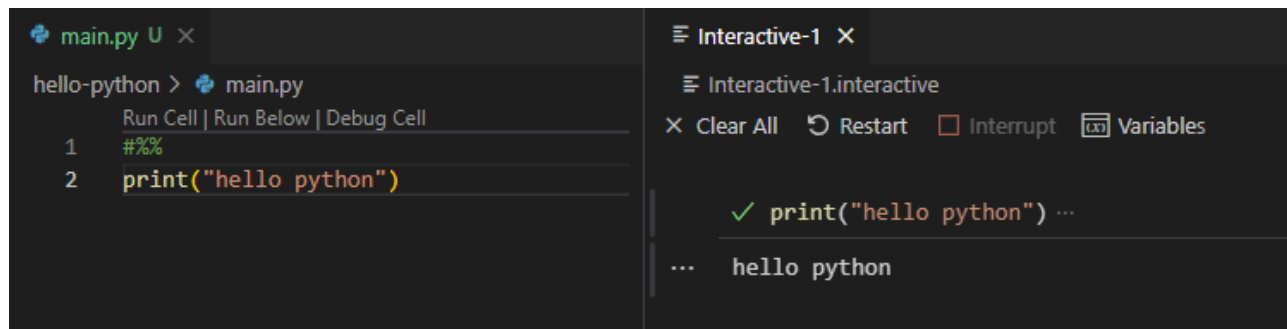
● Menggunakan jupyter code cells

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.



```
main.py U x
hello-python > main.py
1 #%%
2 print("hello python")
```

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



```
main.py U x
Interactive-1 x
Interactive-1.interactive
X Clear All Restart Interrupt Variables
1 #%%
2 print("hello python")
✓ print("hello python") ...
... hello python
```

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.

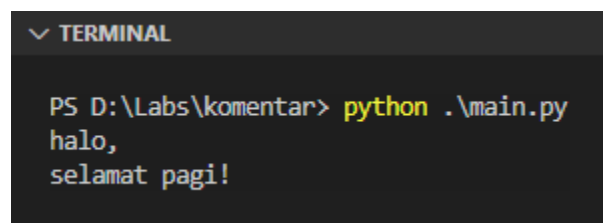
A.3.1. Komentar satu baris

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:



```
✓ TERMINAL

PS D:\Labs\komentar> python .\main.py
halo,
selamat pagi!
```

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"""
```

- Lebih detailnya mengenai string dibahas pada chapter *String*

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 dimasukkan 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 `"noval"` 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))
```

▼ TERMINAL

```
PS D:\Labs\variables> python main.py
==== biodata ====
nama: noval
hobi: makan, umur: 18, laki: True
```

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"
```

Lebih detailnya mengenai string formatting dibahas terpisah pada 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
```

*Lebih detailnya mengenai tipe data dibahas terpisah pada 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:

▼ TERMINAL

```
PS D:\Labs\variables> python main.py  
rata-rata nilai: 24.000
```

Catatan chapter

● Source code praktik

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

● Chapter relevan lainnya

- Tipe Data
- String: formatting
- Number/Bilangan

● 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))
```

▼ TERMINAL

```
PS D:\Labs\variables> python main.py  
pi: 3.140000
```

● Module import

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

di-import.

Lebih detailnya mengenai `import` dan `from` dibahas terpisah pada chapter [Module Import](#)

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).

Lebih detailnya mengenai Python standard library (stdlib) dibahas terpisah pada 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 eksplisit,  
# melainkan didapat dari tipe data nilai  
PI: Final = 3.14  
  
# tipe konstanta TOTAL_MONTH ditentukan secara eksplisit yaitu `int`  
TOTAL_MONTH: Final[int] = 12
```

Lebih detailnya mengenai tipe data dibahas terpisah pada 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
- Module Import
- 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
<code>int</code>	menampung bilangan bulat atau <i>integer</i>	<code>number_1 = 10000024</code>
<code>float</code>	menampung bilangan desimal atau <i>floating point</i>	<code>number_2 = 3.14</code>
<code>complex</code>	menampung nilai berisi kombinasi bilangan real dan imajiner	<code>number_3 = 120+3j</code>

Lebih detailnya mengenai string dibahas pada chapter [Number/Bilangan](#)

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
'''
```

Lebih detailnya mengenai string dibahas 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 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]
```

Lebih detailnya mengenai list dibahas pada chapter [List](#)

A.6.5. Tipe data tuple

Tuple adalah tipe data kolektif yang mirip dengan list, dengan perbedaan 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
```

Lebih detailnya mengenai tuple dibahas pada chapter [Tuple](#)

A.6.6. Tipe data dictionary

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

```
profile_1 = {  
    "name": "Noval",  
    "is_male": False,  
    "age": 16,  
    "hobbies": ["gaming", "learning"]  
}
```

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"] }
```

Lebih detailnya mengenai dictionary dibahas pada chapter *Dictionary*

A.6.7. 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"}
```

Lebih detailnya mengenai set dibahas pada chapter [\[Set\]/basic/set](#))

A.6.8. Tipe data lainnya

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

Catatan chapter

● Source code praktik

[github.com/novalagung/dasarpemrogramanpython-example/./tipe-data](https://github.com/novalagung/dasarpemrogramanpython-example/tree/master/tipe-data)

● Chapter relevan lainnya

- [String](#)
- [List](#)
- [Tuple](#)
- [Dictionary](#)
- [Set](#)

● Referensi

- <https://docs.python.org/3/tutorial/introduction.html>
 - <https://docs.python.org/3/library/stdtypes.html#typeseq>
-

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
<code>+</code>	operasi tambah	<code>num = 2 + 2</code> → hasilnya <code>num</code> nilainya <code>4</code>
unary <code>+</code>	penanda nilai positif	<code>num = +2</code> → hasilnya <code>num</code> nilainya <code>2</code>
<code>-</code>	operasi pengurangan	<code>num = 3 - 2</code> → hasilnya <code>num</code> nilainya <code>1</code>
unary <code>-</code>	penanda nilai negatif	<code>num = -2</code> → hasilnya <code>num</code> nilainya <code>-2</code>
<code>*</code>	operasi perkalian	<code>num = 3 * 3</code> → hasilnya <code>num</code> nilainya <code>9</code>

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
<code>==</code>	apakah kiri sama dengan kanan	<code>res = 4 == 5</code> → hasilnya <code>res</code> nilainya <code>False</code>
<code>!=</code>	apakah kiri tidak sama dengan kanan	<code>res = 4 != 5</code> → hasilnya <code>res</code> nilainya <code>True</code>
<code>></code>	apakah kiri lebih besar dibanding kanan	<code>res = 4 > 5</code> → hasilnya <code>res</code> nilainya <code>False</code>
<code><</code>	apakah kiri lebih kecil dibanding kanan	<code>res = 4 < 5</code> → hasilnya <code>res</code> nilainya <code>True</code>
<code>>=</code>	apakah kiri lebih besar atau sama dengan kanan	<code>res = 5 >= 5</code> → hasilnya <code>res</code> nilainya <code>True</code>
<code><=</code>	apakah kiri lebih kecil atau sama dengan kanan	<code>res = 4 <= 5</code> → hasilnya <code>res</code> nilainya <code>False</code>

A.7.4. Operator logika

Operator	Keterangan	Contoh
<code>and</code>	operasi logika AND	<code>res = (4 == 5) and (2 != 3) →</code> hasilnya <code>res</code> nilainya <code>False</code>
<code>or</code>	operasi logika OR	<code>res = (4 == 5) or (2 != 3) →</code> hasilnya <code>res</code> nilainya <code>True</code>
<code>not</code> atau <code>!</code>	operasi logika negasi (atau NOT)	<code>res = not (2 == 3) →</code> hasilnya <code>res</code> nilainya <code>True</code> <code>res = !(2 == 3) →</code> hasilnya <code>res</code> nilainya <code>True</code>

A.7.5. Operator bitwise

Operator	Keterangan	Contoh
<code>&</code>	operasi bitwise AND	<code>x & y = 0 (0000 0000)</code>
<code> </code>	operasi bitwise OR	<code>x y = 14 (0000 1110)</code>
<code>~</code>	operasi bitwise NOT	<code>~x = -11 (1111 0101)</code>
<code>^</code>	operasi bitwise XOR	<code>x ^ y = 14 (0000 1110)</code>

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)))
```

▼ TERMINAL

```
PS D:\Labs\operator> python.exe main.py
num_1 is num_2 = True
id(num_1): 2545659797168, id(num_2): 2545659797168
```

Di Python ada special case yang perlu kita ketahui perihal penerapan operator `is` untuk operasi perbandingan identitas khusus tipe data

numerik.

Pembahasannya ada di chapter terpisah, yaitu *Object ID & Reference*.

● 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")
```

▼ TERMINAL

```
PS D:\Labs\operator> python.exe main.py
message: hello python learner
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: 19441xxxxxxxxx
```

Nilai kembalian fungsi `id()` bertipe numerik.

Pembahasan versi 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 → False

sample_tuple = ("hello", "python")
is_hello_exists = "hello" in sample_tuple
print(is_hello_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](https://github.com/novalagung/dasarpemrogramanpython-example/blob/master/operator)

● Chapter relevan lainnya

- Variabel
- Tipe Data
- String: formatting

● 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!")
```

▼ TERMINAL

```
PS D:\Labs\if-elif-else> python.exe main.py
perfect
```

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.

```
▼ TERMINAL

PS D:\Labs\if-elif-else> python.exe main.py
Enter your grade: 86
awesome
PS D:\Labs\if-elif-else> python.exe main.py
Enter your grade: 100
perfect
PS D:\Labs\if-elif-else> python.exe main.py
Enter your grade: 40
```

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))
```

```
✓ TERMINAL

PS D:\Labs\if-elif-else> python.exe main.py
Enter your grade: 78
inputan user: 78 <class 'str'>
```

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 menunjukkan bahwa tipe datanya adalah `int`.

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

▼ TERMINAL

```
PS D:\Labs\if-elif-else> python.exe main.py
Enter your grade: 79
inputan user: 79 <class 'int'>
```

Lebih detailnya mengenai mengenai type conversion dibahas pada 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")
```


▼ TERMINAL

```
PS D:\Labs\if-elif-else> python.exe main.py
Enter your grade: 69
passed the exam
but you need to improve it!
```

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`.

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)
```

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

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))
```

```
✓ TERMINAL
r = range(5)
print("r:", list(r))
✓ 0.0s
... r: [0, 1, 2, 3, 4]
```

- Lebih detailnya mengenai list dibahas pada chapter [List](#)
- Lebih detailnya mengenai mengenai type conversion dibahas pada 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:

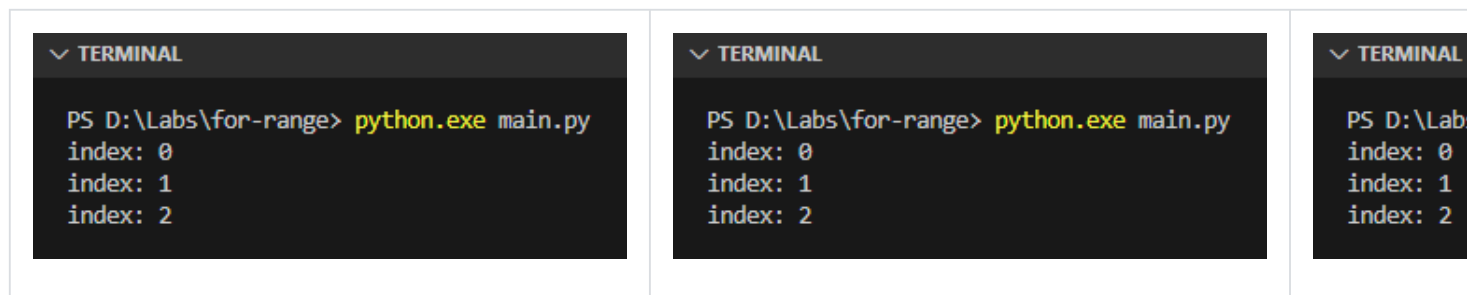
- Menggunakan `range(start, stop)`. Hasilnya data *range* dimulai dari `start` dan hingga `stop - 1`. Sebagai contoh, `range(1, 4)` menghasilkan data range `[1, 2, 3]`.
- Menggunakan `range(start, stop, step)`. Hasilnya data *range* dimulai dari `start` dan hingga `stop - 1`, dengan nilai *increment*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

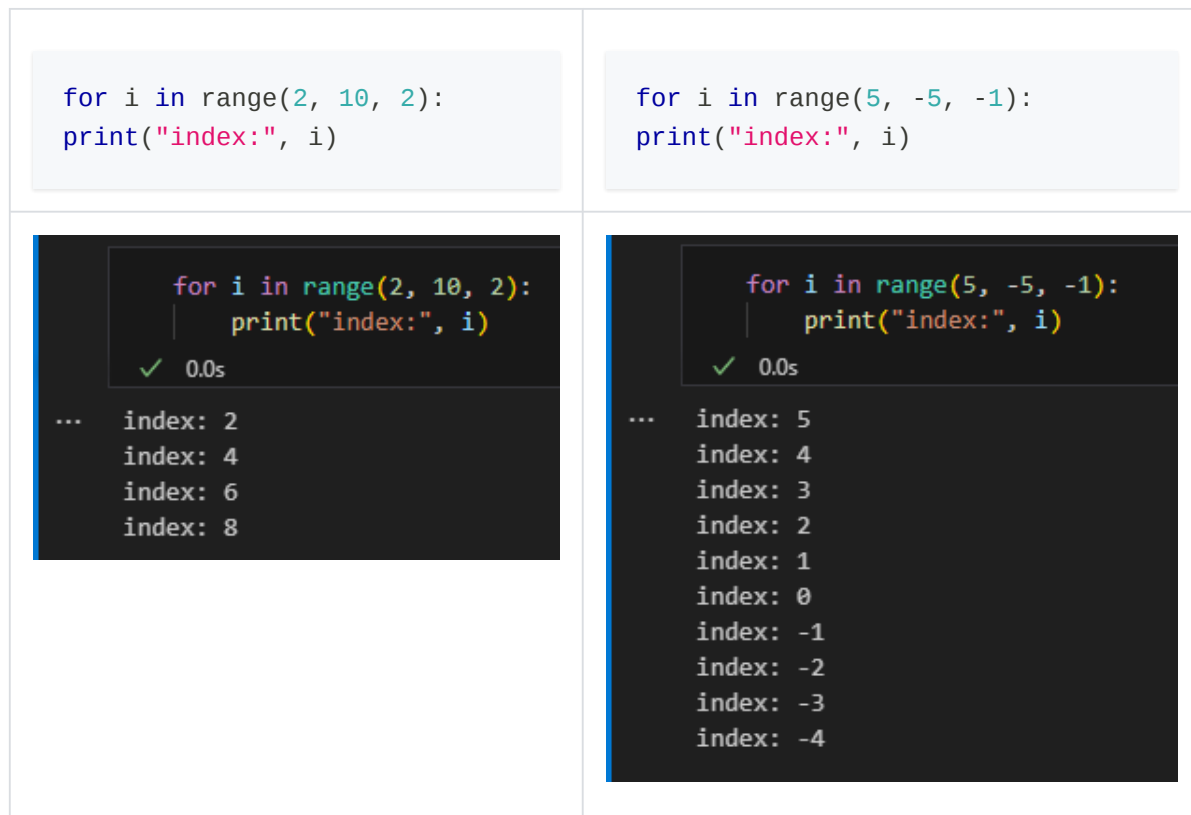
```
for i in range(3):
```

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

```
for i in
```



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



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`. Contoh penerapannya bisa dilihat di bawah ini:

● Iterasi data list

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

▼ TERMINAL

```
PS D:\Labs\for-range> python.exe main.py
morning
afternoon
evening
```

● Iterasi data tuple

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

▼ TERMINAL

```
PS D:\Labs\for-range> python.exe main.py
twenty four
24
```

● Iterasi data string

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

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



```
▼ TERMINAL
PS D:\Labs\for-range> python.exe main.py
h
e
l
l
o

p
y
t
h
o
n
```

● 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]`.

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

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

```
▼ TERMINAL
PS D:\Labs\for-range> python.exe main.py
key: name value: toyota camry
key: year value: 1993
```

● Iterasi data set

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

```
▼ TERMINAL
PS D:\Labs\for-range> python.exe main.py
24
twenty four
```

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()
```

```
▼ TERMINAL
PS D:\Labs\for-range> python.exe main.py
jumlah bintang: 3
* * *
* *
*

PS D:\Labs\for-range> python.exe main.py
jumlah bintang: 5
* * * * *
* * * *
* * *
* *
*
```

● 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).

*Lebih detailnya tentang fungsi dan parameter opsional dibahas pada chapter **Fungsi***

● Fungsi `print()` tanpa parameter

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

Catatan chapter

● Source code praktik

[github.com/novalagung/dasarpemrogramanpython-example/.../for-range](https://github.com/novalagung/dasarpemrogramanpython-example/blob/master/for-range.py)

● Chapter relevan lainnya

- [List](#)
- [String](#)
- [Fungsi](#)

● 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)
```

▼ TERMINAL

```
PS D:\Labs> python.exe main.py
enter an even number greater than 0: 10
number: 10
enter an even number greater than 0: 2
number: 2
enter an even number greater than 0: 8
number: 8
enter an even number greater than 0: 7
7 is not an even number greater than 0
```

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
```

▼ TERMINAL

```
PS D:\Labs> python.exe main.py
enter max data: 6
number 0
number 1
number 2
number 3
number 4
number 5
```

● 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
<i>Increment</i>	<code>i += 1</code>	<code>i = i + 1</code>
<i>Decrement</i>	<code>i -= 1</code>	<code>i = i - 1</code>

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
```



```
▼ TERMINAL

PS D:\Labs> python.exe main.py
enter max data: 4
* * * *
* * *
* *
*
PS D:\Labs> python.exe main.py
enter max data: 2
* *
*
```

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))
```

▼ TERMINAL

```
PS D:\Labs> python.exe main.py
enter a number divisible by 3: 9
9 is divisible by 3
enter a number divisible by 3: 24
24 is divisible by 3
enter a number divisible by 3: 11
```

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.

```
✓ TERMINAL

PS D:\Labs> python.exe main.py
3
4
5
6
7
```

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: "))

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

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

Penjelasan:

- Program yang memiliki perulangan *nested* dengan jumlah perulangan ada 2.
 - Disiapkan sebuah variabel `bool` bernama `outerLoop` untuk kontrol perulangan terluar.
 - Ketika nilai `j` (yang merupakan variabel counter perulangan terdalam) adalah lebih dari atau sama dengan `7`, maka variabel `outerLoop` 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
- 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. Penerapan 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.

Lebih detailnya mengenai fungsi `len()` dibahas setelah ini

● Fungsi `enumerate()`

Fungsi `enumerate()` digunakan untuk membuat data sequence menjadi data enumerasi, yang jika dimasukkan 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()
```

```
▼ TERMINAL

PS D:\labs> python.exe main.py
0 1 0 1 0
1 1 1 0 0
0 0 0 1 1
0 1 1 1 0
```

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]
```

Lebih detailnya mengenai tuple dibahas pada 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']
```

*Lebih detailnya mengenai slice dibahas pada 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]
```

- Lebih detailnya mengenai method dibahas pada chapter *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)
# output → removed element: 20
```

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

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

muncul.

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

✓ TERMINAL

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

```
-----
IndexError                                Traceback (most recent call last)
d:\Labs\Adam Studio\Ebook\dasarpemrogramanpython\dasarpemrogramanpython
  83 # %% A.12.2. © Menghapus element pada index `i`
  84 list_3 = [10, 70, 20, 70]
----> 86 x = list_3.pop(7)
      87 print('list_3:', list_3)
      88 print('removed element:', x)
```

```
IndexError: pop index out of range
```

- 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. Perbedaananya, 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)
```

▼ TERMINAL

```
❌ idx_2nd = list_2.index('kk') ...
```

```
-----
ValueError                                Traceback (most recent call last)
d:\Labs\Adam Studio\Ebook\dasarpemrogramanpython\dasarpemrogramanpython\examples
  36 # %%
----> 37 idx_2nd = list_2.index('kk')
      38 print('idx_2nd: ', idx_2nd)

ValueError: 'kk' is not in list
```

● 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()
```

▼ TERMINAL

```
PS D:\labs> python.exe main.py  
[10, 20, 70]  
['c', 'h', 'z']
```

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

Lebih detailnya mengenai closure/lambda dibahas pada chapter [Closure/Lambda](#)

Catatan chapter

● Source code praktik

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

● Chapter relevan lainnya

- Perulangan → for, range
- List Comprehension
- Slicing
- Closure/lambda

● TBA

- Pack & Unpack with `*` & `**`

● Referensi

- <https://docs.python.org/3/tutorial/datastructures.html>
 - <https://docs.python.org/3/library/stdtypes.html#typeseq>
-

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. Penerapan 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']

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

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

... bisa dituliskan lebih ringkas menjadi seperti berikut:

```
comb = [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

● TBA

- Stack vs Queue

● Referensi

- <https://docs.python.org/3/tutorial/datastructures.html>
 - <https://docs.python.org/3/library/stdtypes.html#typeseq>
-

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	<code>()</code> , atau <code>tuple()</code> , atau elemen ditulis tanpa <code>()</code>	<code>[]</code> , atau <code>list()</code>
Contoh	<pre>x = ()</pre> <pre>x = tuple()</pre> <pre>x = (1, True, "h", 2, 1)</pre> <pre>x = 1, True, "h", 2, 1</pre>	<pre>x = []</pre> <pre>x = list()</pre> <pre>x = [1, True, "h", 2, 1]</pre>
Urutan elemen	urut sesuai index	

	Tuple	List
Pengaksesan elemen	via index dan perulangan	
<i>Mutability</i>	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. Penerapan 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.

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)
```

▼ TERMINAL

```
ultra instinc shaggy  
nightwing  
noob saibot
```

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 dimasukkan 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")
```

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()
```

▼ TERMINAL

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

0 2
0 3
2 2
2 4
```

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))

# vertikal
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 | affiliation")
print("-----")
for row in data:
    for cell in row:
        print(cell, end=" | ")
    print()
```

▼ TERMINAL

```
name | rank | win | affiliation
-----
ultra instinc shaggy | 1 | True | ['detective', 'saiyan'] |
nightwing | 3 | True | ['teen titans', 'bat family'] |
noob saibot | 6 | False | ['brotherhood of shadow'] |
tifa lockhart | 2 | True | ['avalanche'] |
```

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 karakter `(` & `)`.

```
# 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)
```

● 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
```

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

- Pack & Unpack with `*` & `**`
- Zip

● Referensi

- <https://docs.python.org/3/tutorial/datastructures.html>
 - <https://docs.python.org/3/library/stdtypes.html#typeseq>
-

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	<code>set()</code> , atau elemen ditulis diapit <code>{</code> dan <code>}</code>	<code>()</code> , atau <code>tuple()</code> , atau elemen ditulis tanpa <code>()</code>		<code>[]</code> , atau <code>list()</code>
Contoh	<code>x = set()</code>	<code>x = ()</code>	<code>x = []</code>	
	<code>x = {1, True, "h", 2}</code>	<code>x = tuple()</code>	<code>x = list()</code>	
		<code>x = (1, True, "h", 2, 1)</code>	<code>x = [1, True, "h", 2, 1]</code>	
		<code>x = 1, True, "h", 2, 1</code>		
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. Penerapan 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: 3
```

- 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)
```

▼ TERMINAL

```
legolas
aragorn
gimli
```

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)
```

```

▼ TERMINAL
-----
KeyError                                Traceback (most recent call last)
d:\Labs\Adam Studio\Ebook\dasarpemrogramanpython\dasarpemrogramanpython\examples\sets\main_3.py
  40 fellowship.discard('batman')
  41 print("fellowship:", fellowship)
--> 43 fellowship.remove('superman')
     44 print("fellowship:", fellowship)

KeyError: 'superman'

```

● 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 adalah 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 = {'gimli'}
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', 'gimli', '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/nova1agung/dasarpemrogramanpython-example/blob/master/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. Penerapan 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 `Set`, hanya saja bedanya pada tipe dictionary isinya berbentuk **key-value**.

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))
```

▼ TERMINAL

```
{'id': 2, 'name': 'john wick', 'hobbies': ['playing with pencil'], 'is_female': False}  
total keys: 4
```

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 `umur` misalnya (`profile["umur"]`) hasilnya adalah error.

● Urutan item dictionary

Mulai dari Python version 3.7, item dictionary tersimpan secaraurut. 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)
```

▼ TERMINAL

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

- 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))
```

▼ TERMINAL

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

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:

▼ TERMINAL

```
key: id      value: 2
key: name    value: mario
key: hobbies value: ('playing with luigi', 'saving the mushroom kingdom')
key: is_female value: False
```

A.15.4. 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,
    "affiliations": [
        {
            "name": "luigi",
            "affiliation": "brother"
        },
        {
            "name": "mushroom kingdom",
            "affiliation": "protector"
        },
    ],
}

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

Pada kode di atas, key `affiliations` berisi array object dictionary.

Contoh cara mengakses value nested item dictionary:

```
value = profile["affiliations"][0]["name"],
profile["affiliations"][0]["affiliation"]
print("  → %s (%s)" % (value))
# output → luigi (brother)

value = profile["affiliations"][1]["name"],
profile["affiliations"][1]["affiliation"]
print("  → %s (%s)" % (value))
# output → mushroom kingdom (protector)
```

A.15.5. 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
}

print(profile["affiliations"][0]["name"])
# output → luigi

profile["affiliations"][0]["name"] = "luigi steven"

print(profile["affiliations"][0]["name"])
# output → luigi steven
```

A.15.6. 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.

```
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', 'is_female']
```

● Pengaksesan dictionary

Method `values()` digunakan untuk mengakses semua keys 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 keys 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'), ('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)
# output → {'id': 2, 'name': 'mario', 'is_female': False}
```

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

- [Classes & 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. Penerapan 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 `"""` untuk mengapit text. Contoh:

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

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

● 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"
print(text)
# output → 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*.

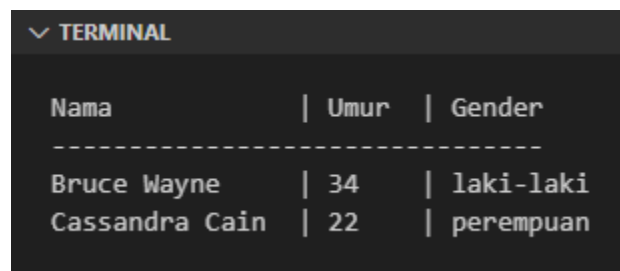
Python mengenal banyak special characters yang masing-masing memiliki kegunaan yang cukup spesifik. Agar lebih jelas silakan lihat tabel berikut:

Special character	Kegunaan
\\	karakter backslash (\)
\'	tanda petik satu (')
\"	tanda kutip (petik dua) (")
\a	bunyi <i>beeb</i> (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)
\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:



```

Nama          | Umur | Gender
-----
Bruce Wayne   | 34   | laki-laki
Cassandra Cain | 22   | perempuan
```

Syntax `0xC548` adalah salah satu penulisan numerik berbasis hexadecimal. Lebih jelasnya dibahas pada chapter [Number/Bilangan](#).

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 salahkan 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)
```

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.

*Lebih detailnya mengenai string formatting dibahas pada chapter **String Formatting***

A.17.4. 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
```

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)
# output → hello 123 True
```

Lebih detailnya mengenai fungsi `str()` dibahas setelah ini pada bagian *Konversi data ke string*

- 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
```
```

## 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 lebar 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])
output → h

print(text[1])
output → e

print(text[2])
output → l
```

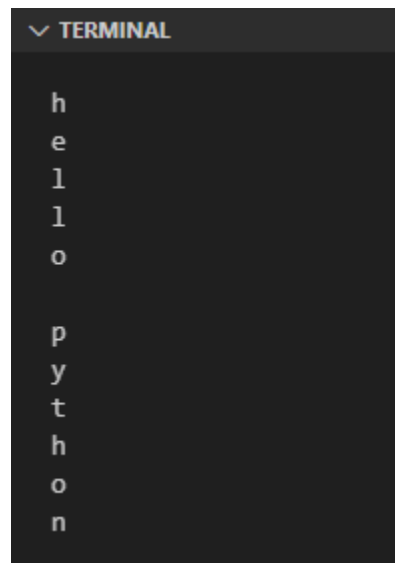
Selain via index, keyword perulangan `for` bisa dimanfaatkan untuk mengiterasi elemen string. Contoh:

```
for c in text:
 print(c)
```

Contoh lain menggunakan `range()` :

```
for i in range(0, len(text)):
 print(text[i])
```

Output:



```
h
e
l
l
o

p
y
t
h
o
n
```

### 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
```

Lebih detailnya mengenai slice dibahas pada 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
```

- 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('2½'.isdigit())
output → False, karena bilangan pecahan memiliki karakter `½` 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('2½'.isdecimal())
```

- 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

print('2½'.isnumeric())
output → True, karena bilangan pecahan termasuk dalam kategori numerik

print('4²'.isnumeric())
output → True, karena bilangan pangkat termasuk dalam kategori numerik

print('٢٨'.isnumeric())
output → True, karena ٢٨ adalah angka numerik arabic

print('4'.isnumeric())
output → True, karena 4 adalah angka 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("12345½".isalnum())
```

## ● 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

print("\n\r".isspace())
output → True, karena string berisi karakter newline

print("hello\n\r".isspace())
output → False, karena string berisi tulisan hello yang tidak termasuk
dalam kategori whitespace
```

## ● 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())
output → HELLO PYTHON

print("Hello Python".lower())
output → hello python

print("Hello Python".swapcase())
output → hELLO pYTHON
```

## 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 dari string lain.

- Menggunakan method `startswith()` untuk mengecek apakah suatu string diawali dengan huruf/kata tertentu.

```
print("hello world".startswith("hell"))
output → True

print("hello world".startswith("ello"))
output → False
```

- Menggunakan method `endswith()` untuk mengecek apakah suatu string diakhiri dengan huruf/kata tertentu.

```
print("hello world".endswith("orld"))
output → True
```

- 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
"""

print(f"--{text}--")
output ↓
#
--
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}
```

*Lebih detailnya mengenai konversi tipe data dibahas pada chapter  
Konversi Tipe Data*

## Catatan chapter

### 🕒 Source code praktik

[github.com/novalagung/dasarpemrogramanpython-example/./string](https://github.com/novalagung/dasarpemrogramanpython-example/./string)

## ● Chapter relevan lainnya

- [Unicode](#)
- [Slicing](#)

## ● 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](https://docs.python.org/3/reference/lexical_analysis.html#f-strings)
-



# A.18. Python Unicode String

Python mengadopsi aturan standar **Unicode** dalam pengolahan karakter dalam 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. Penerapan **Unicode**

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 = "\u00C548\u00B155\u00D558\u00C138\u00C694"
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 hexadesimal, cukup bungkus menggunakan fungsi `hex()`.

```
text = "◇"
codePoint = ord(text)

print(f'code point of {text} in decimal: {codePoint}')
```

Bisa dilihat dari program di atas, unicode code point dari karakter `📄` dalam bentuk hexadecimal adalah `c548`. Jika dicek pada praktek sebelumnya, kode hexadecimal 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 hexadecimal `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](https://github.com/novalagung/dasarpemrogramanpython-example/blob/master/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 hexadecimal: `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

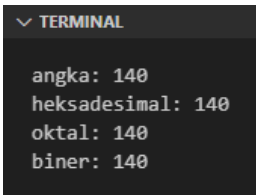
print(f"heksadesimal: {angka_heksadesimal}")
output → heksadesimal: 140

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.



```

▼ TERMINAL

angka: 140
heksadesimal: 140
oktal: 140
biner: 140

```

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:

```
▼ TERMINAL
angka: 140
heksadesimal: 8c
oktal: 214
biner: 10001100
```

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<br>angka2 = 13     | <code>d</code><br>atau<br>tanpa<br>suffix | <code>print(f"angka1: {angka1}")</code><br><code>print(f"angka2: {angka2:d}")</code> |
| Hexadecimal | 16    | <code>0x</code> | hex1 = 0x8c<br>hex2 = 0xff00c0 | <code>x</code>                            | <code>print(f"hex1: {hex1:x}")</code><br><code>print(f"hex2: {hex2:x}")</code>       |
| Octal       | 8     | <code>0o</code> | oct1 = 0o214<br>oct2 = 0o605   | <code>o</code>                            | <code>print(f"oct1: {oct1:o}")</code><br><code>print(f"oct2: {oct2:o}")</code>       |
| Binary      | 2     | <code>0b</code> | bin1 = 0b1010<br>bin2 = 0b110  | <code>b</code>                            | <code>print(f"bin1: {bin1:b}")</code><br><code>print(f"bin2: {bin2:b}")</code>       |

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:

▼ TERMINAL

```
angka 140 sama dengan biner 10001100
```

## ● 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 → angka 140 sama dengan biner 10001100
```

Output program:

▼ TERMINAL

```
total: 560 (hex: 230, oct: 1060, bin: 1000110000)
```

## ● 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}")
```

▼ TERMINAL

```
3.14 + 2.8: 5.9399999999999995
```

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 eksponensial, 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 imajineranya 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](https://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`.

|         |   |   |   |   |   |   |   |   |   |   |    |
|---------|---|---|---|---|---|---|---|---|---|---|----|
| Element | h | e | l | l | o |   | w | o | r | l | d  |
| Index   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 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
```

|         |   |   |   |   |   |   |   |   |   |   |    |
|---------|---|---|---|---|---|---|---|---|---|---|----|
| Element | h | e | l | l | o |   | w | o | r | l | d  |
| Index   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

- 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
```

| Element | h | e | l | l | o |   | w | o | r | l | d  |
|---------|---|---|---|---|---|---|---|---|---|---|----|
| Index   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

- 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]`.

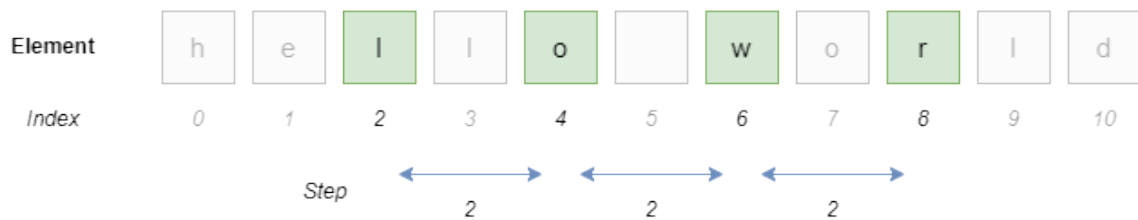
```
slice3 = data_str[:5]
print(slice3)
output → hello
```

| Element | h | e | l | l | o |   | w | o | r | l | d  |
|---------|---|---|---|---|---|---|---|---|---|---|----|
| Index   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |

- 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)]`).

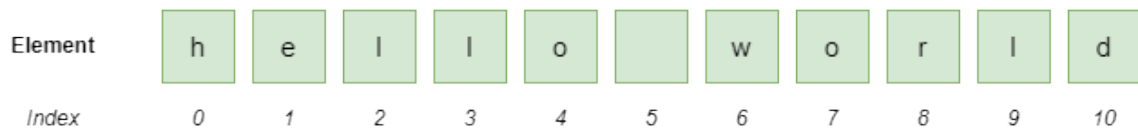
```
slice4 = data_str[4:]
```





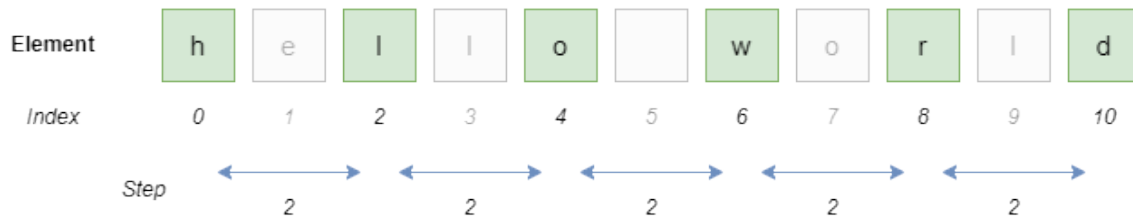
- Slicing seluruh element bisa dilakukan dengan notasi `data_str[:]`. Notasi tersebut adalah ekuivalen dengan `data_str[0:len(data_str)]`.

```
slice7 = data_str[:]
print(slice7)
output → hello world
```



- Slicing seluruh element dengan 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.

*Kita akan bahas lebih detail topik reference pada chapter berikutnya, yaitu **Call by Assignment**.*

## 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]

s1 = slice(2, 6)
list3 = data_list[s1]
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 apa 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:

#### ▼ TERMINAL

```
data: 24
id: 140728206353928
```

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:

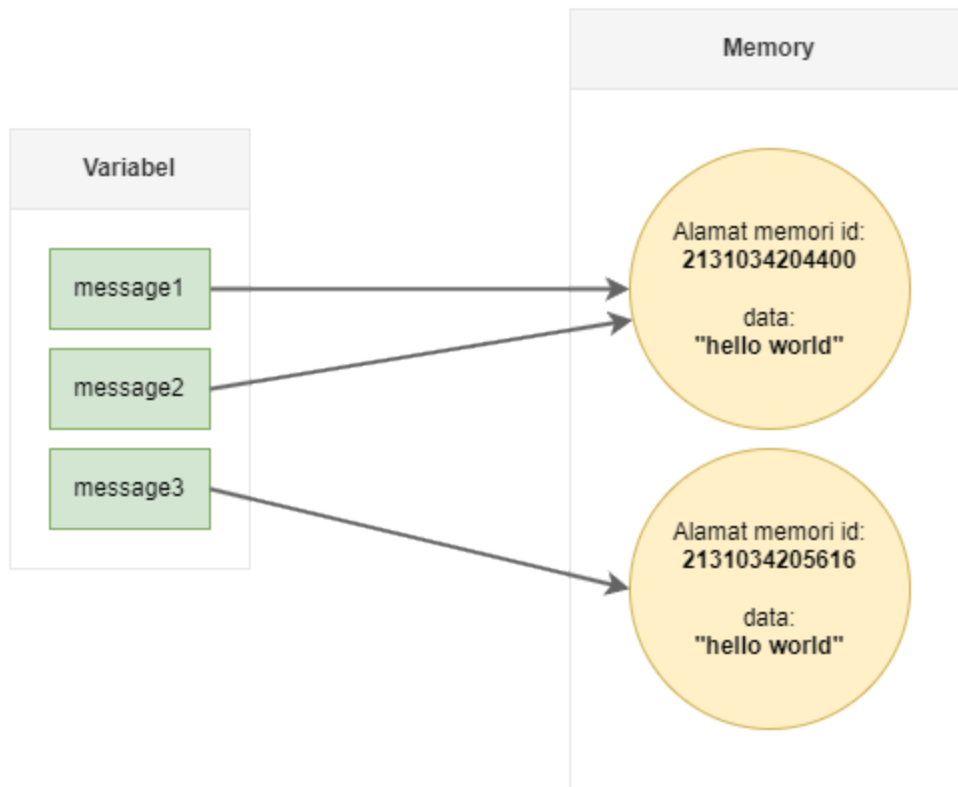
```
▼ TERMINAL

var: message1, data: hello world, id: 2131034204400
var: message2, data: hello world, id: 2131034204400
var: message3, data: hello world, id: 2131034205616
```

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 mengakses 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 statement 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:

```
✓ TERMINAL

message1 (1992124198192) is message2 (1992124198192) → True
message1 (1992124198192) is message3 (1992124191536) → False
message2 (1992124198192) is message3 (1992124191536) → False

message1 (1992124198192) is message2 (1992124200368) → False
message1 (1992124198192) is message3 (1992124191536) → False
message2 (1992124200368) is message3 (1992124191536) → False

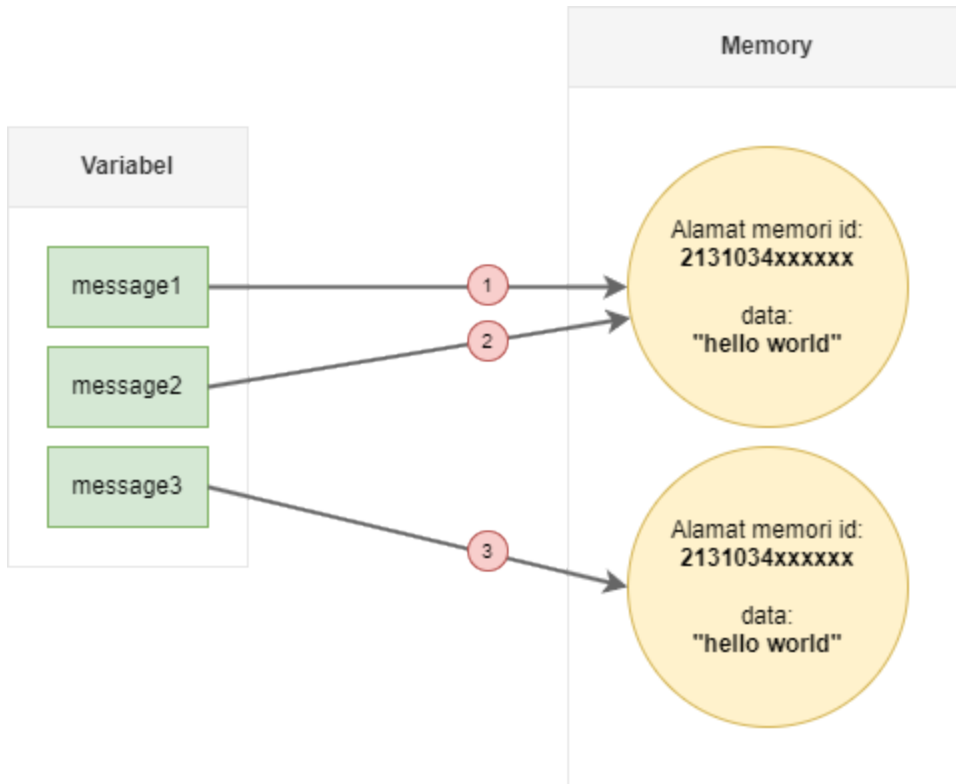
message1 (1992124198192) is message2 (1992124200368) → False
message1 (1992124198192) is message3 (1992124200368) → False
message2 (1992124200368) is message3 (1992124200368) → True
```

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:

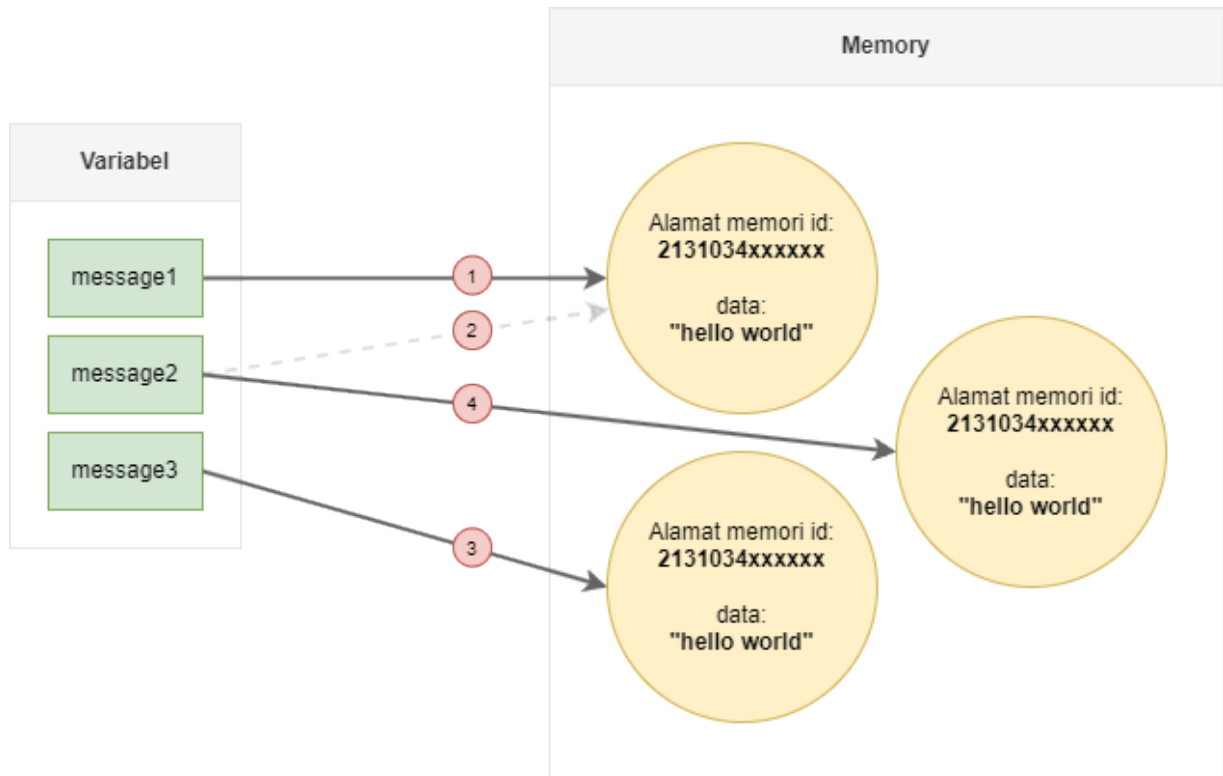
```
message1 = "hello world" # statement 1
message2 = message1 # statement 2
message3 = "hello world" # statement 3
```



- Fase 2:

```
message1 = "hello world" # statement 1
message2 = message1 # statement 2
message3 = "hello world" # statement 3

message2 = "hello world" # statement 4
```

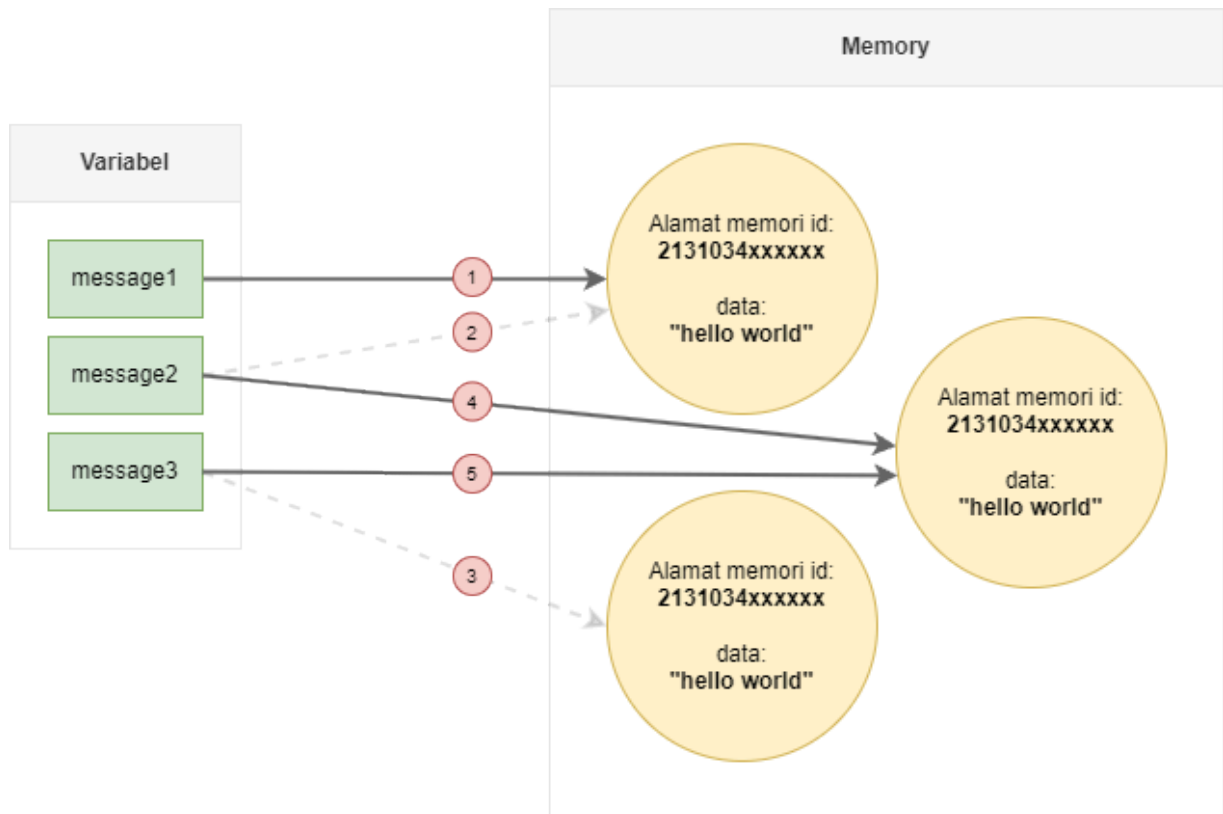


- Fase 3:

```
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 element-nya bisa dimutasi atau diubah tanpa men-*trigger* alokasi alamat memory baru (identfier-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", numbers2, id(numbers2), sys.getsizeof(numbers2))

```

Output program:

#### ▼ TERMINAL

```
numbers1 [1, 2, 3, 4] 2269649128896 88

numbers1 [1, 2, 3, 4, 9] 2269649128896 120
numbers2 [1, 2, 3, 4, 9] 2269649128896 120
```

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 diubah.

### ● 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:

```
✓ TERMINAL

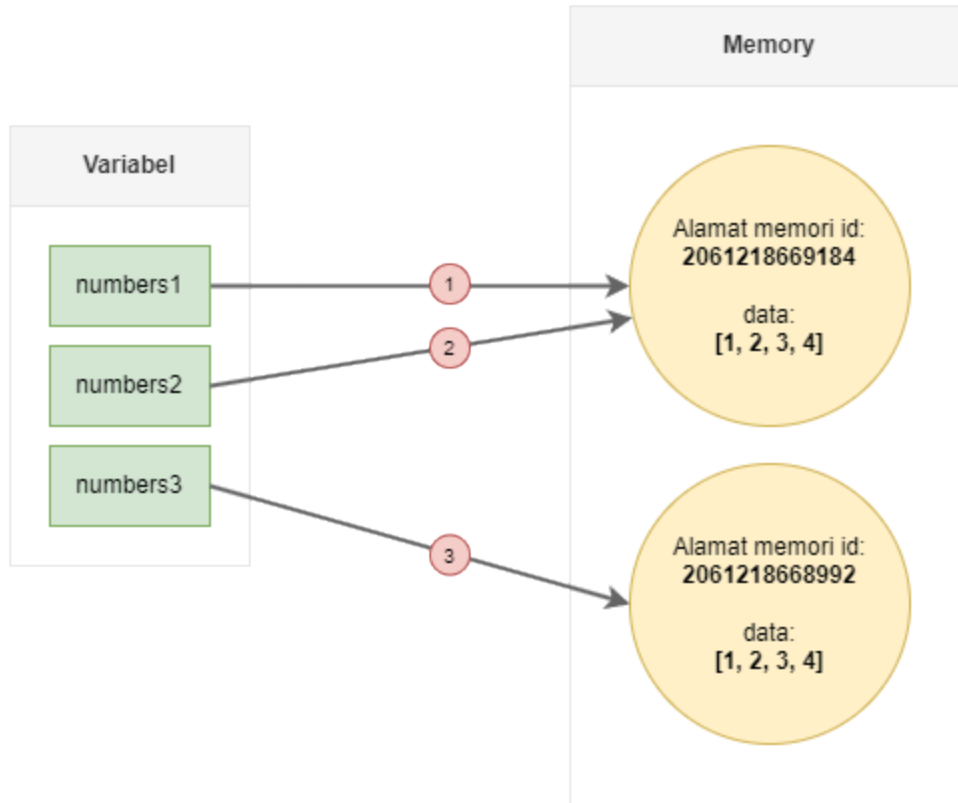
numbers1 [1, 2, 3, 4] 2061218669184
numbers2 [1, 2, 3, 4] 2061218669184
numbers3 [1, 2, 3, 4] 2061218668992

numbers1 [1, 2, 3, 4, 9] 2061218669184
numbers2 [1, 2, 3, 4, 9] 2061218669184
numbers3 [1, 2, 3, 4] 2061218668992
```

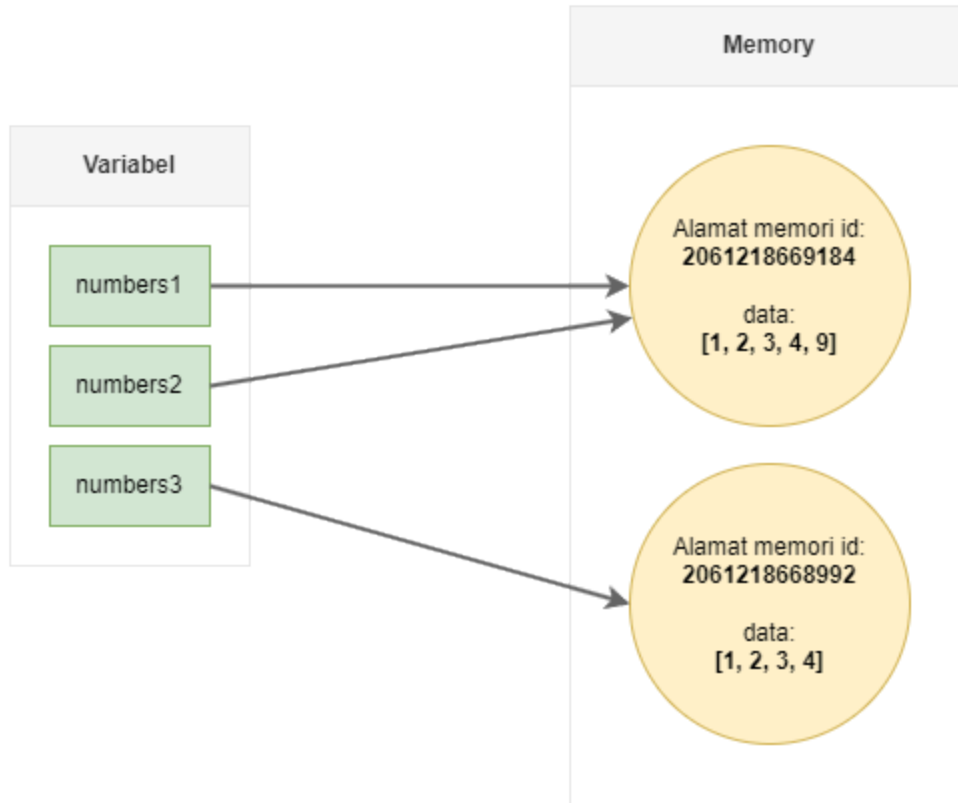
Penjelasan:

- 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

print(f"id n1: {id(n1)}, id n2: {id(n2)}")
output → id n1: 140728206353544, id n2: 140728206353544

print(f"n1 == n2: {n1 == n2}")
output → n1 == n2: True

print(f"n1 is n2: {n1 is n2}")
output → n1 is n2: True
```

---

## Catatan chapter

### ● Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-example/./object-id-reference
```

### ● Chapter relevan lainnya

- Slicing

### ● 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. Penerapan 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:

```
▼ TERMINAL

hello
hello
hello
```

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.

```
main_1.py 1, U x
examples > function > main_1.py > ...
1 say_hello()
2
3 def say_hello():
4 print("hello")
5

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS

PS C:\examples\function> python.exe main_1.py
Traceback (most recent call last):
 File "C:\examples\function\main_1.py", line 1, in <module>
 say_hello()
 ^^^^^^^^^
NameError: name 'say_hello' is not defined
```

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*

Untuk tambahan latihan, buat satu fungsi lagi, lalu isi dengan banyak statement. Misalnya:

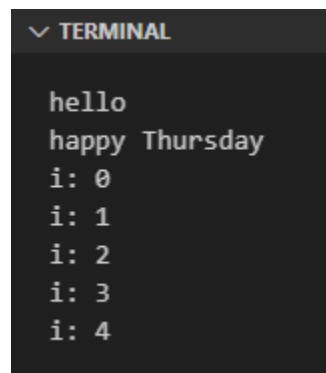
```
def print_something():
 print("hello")

 today = "Thursday"
 print(f"happy {today}")

 for i in range(5):
 print(f"i: {i}")

print_something()
```

Output program:



```
✓ TERMINAL

hello
happy Thursday
i: 0
i: 1
i: 2
i: 3
i: 4
```

Penulisan isi fungsi (statement-statement dalam fungsi) harus disertai dengan *indentation* yang benar. Isi statement posisinya tidak boleh sejajar 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 disisipkan 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 istilah 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**, nantinya kita akan mempelajarinya pada chapter **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`.

Lebih detailnya mengenai tipe data `None` dibahas pada chapter *None*

## A.22.5. 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](https://github.com/novalagung/dasarpemrogramanpython-example/./function)



## ● Chapter relevan lainnya

- Optional, Positional, Keyword Argument
- Args & Kwargs
- Closure
- Lambda

## ● Referensi

- <https://docs.python.org/3/tutorial/controlflow.html#defining-functions>
  - <https://peps.python.org/pep-3102/>
-

# 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 harusurut 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. Sisalnya (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` diset 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 outpunya. 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.

```
▼ TERMINAL

test print matrix 1:
[]

test print matrix 2:
[1, 2]
[5, 6]

test print matrix 3:
[2, 3, 4]
[3, 1, 6]
```

## 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, sebelum penulisan argument yang menggunakan metode *positional argument*. Jika dipaksa ditulis terbalik, maka menghasilkan error. Contohnya seperti pada gambar berikut:

```
43 print(f"matrix * scalar {9}:")
44 res6 = matrix_multiply_scalar(matrix=matrix, 9)
45 print_matrix(res6)
```

PROBLEMS 3 OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS JUPYTER

PS C:\LibsSoftLink\dasarpemrogramanpython\examples\optional-positional-keyword-only-argument> python main\_4.py

File "C:\LibsSoftLink\dasarpemrogramanpython\examples\optional-positional-keyword-only-argument\main\_4.py", line 44

```
res6 = matrix_multiply_scalar(matrix=matrix, 9)
 ^
SyntaxError: positional argument follows keyword argument
```

# Catatan chapter

## ● Source code praktik

```
github.com/novalagung/dasarpemrogramanpython-
example/.../positional-optional-keyword-argument
```

## ● Chapter relevan lainnya

- Function
- Args & Kwargs
- Closure
- 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. 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 dua 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**.

```
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. 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.

```
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 positional semua 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](https://github.com/novalagung/dasarpemrogramanpython-example/blob/master/args-kwargs)

### ● Chapter relevan lainnya

- [Function](#)

- Optional, Positional, Keyword Argument
- Closure
- 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. Penerapan closure / *nested function*

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 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)}")
```

Output program:

```
✓ TERMINAL

call outer_func()
numbers: [1, 2, 3, 4]

call inner_func() within outer_func()
max: 4
min: 1

call inner_func() outside of outer_func()
max: 4
min: 1
```

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-4 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
- Args & Kwargs
- Optional, Positional, Keyword Argument
- 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. Penerapan 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 statement. 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`.

Lebih detailnya mengenai tipe data `None` dibahas pada 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**:

```

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 kompleks 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]]
```

*Lebih detailnya mengenai list comprehensian dibahas pada 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](https://github.com/novalagung/dasarpemrogramanpython-example/..../lambda)

### ● Chapter relevan lainnya

- [Function](#)
- [Optional, Positional, Keyword Argument](#)
- [Args & Kwargs](#)
- [Closure](#)

### ● Referensi

- <https://docs.python.org/3/reference/expressions.html#lambda>
-