

BAB 03

CASAR **PEMROGRAMAN PYII-ION** **(BAGIAN KEDUA)**

Pokok bahasan pada bab ini mencakup hal-hal berikut:

- indentasi di Python;
- struktur kontrol;
- pembuatan fungsi;
- pembuatan kelas;
- pengenalan struktur data;
- argumen baris perintah;
- penanganan eksepsi;
- pengenalan numpy.

3.1 Indentasi di Python

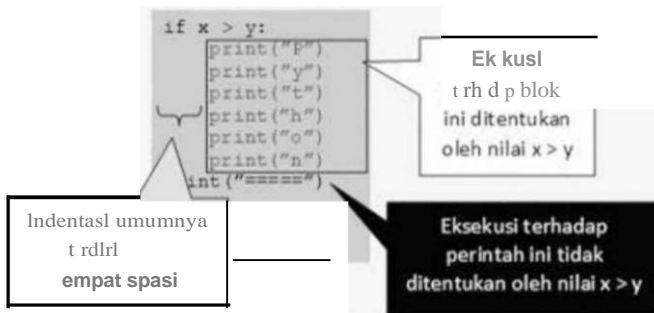
Di Python, indentasi atau penulisan teks yang menjorok ke kanan mempunyai peran yang sangat penting karena digunakan sebagai penanda blok kode pada pernyataan seperti `if`, `while`, dan `for`.

Sebagai contoh, perhatikan kode berikut:

```
if x > y:
    print("P")
    print("y")
    print("t")
    print("h")
    print("o")
    print("n")
print("=====")
```

Deretan pernyataan `print ()` yang diletakkan menjorok ke kanan terhadap baris `if` akan dieksekusi bergantung pada kondisi `x > y`, sedangkan pernyataan `print ("=====")` selalu dijalankan, tidak tergantung pada kondisi `if`.

Tidak ada aturan pasti yang menentukan jumlah karakter spasi yang digunakan untuk mengatur indentasi. Namun, umumnya empat karakter spasi digunakan oleh para pemrogram untuk mengatur indentasi. Gambar 3.1 memberikan penjelasan secara visual mengenai hal ini.



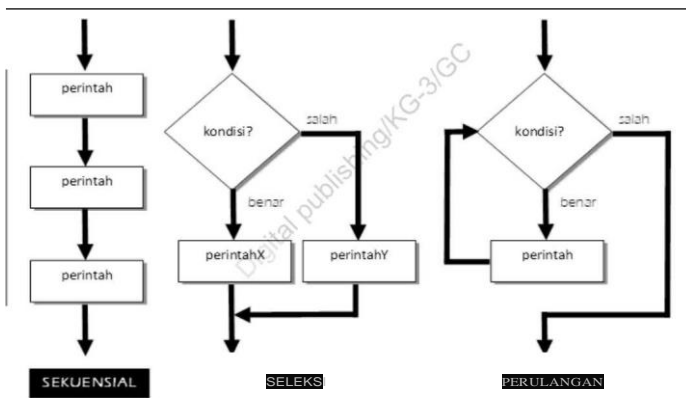
Gambar 9.1 Indentasi untuk menyatakan blok kode

3.2 Struktur Kontrol

Struktur kontrol pada python dapat dibagi menjadi tiga kategori:

- 1) sekuensial;
- 2) seleksi;
- 3) perulangan.

Struktur sekuensial menyatakan deretan perintah yang dijalankan secara berurutan. Struktur seleksi bermanfaat untuk melakukan pengambilan keputusan. Adapun struktur perulangan berguna untuk mengulang suatu tindakan beberapa kali. Gambar 3.2 memberikan gambaran ketiga struktur kontrol tersebut dalam bentuk diagram alir.



Gambar 9. Tiga jenis struktur kontrol di Python

Catatan

Struktur seleksi dan perulangan biasa digunakan pada skrip.

3.2.1 Struktur if-else

Struktur `if-else` berguna untuk melakukan pengambilan keputusan, yakni untuk memilih salah satu tindakan terhadap lebih dari satu pilihan. Bentuknya seperti berikut:

```

if kondisi:
    perintah_1
    perintah_2

    perintah_n

```

Perhatikan bahwa:

- 1) setelah *kondisi* terdapat tanda titik-dua (:);
- 2) *perintah_1* hingga *perintah_n* ditulis menjorok ke kanan, misalnya diawali empat karakter spasi.

Pada bentuk seperti itu, perintah-perintah yang ditulis menjorok tersebut akan dijalankan kalau *kondisi* bernilai `True`.

Untuk memahami if, silakan untuk mencoba skrip berikut:



```

# Penentuan genap dan ganjil

nilai = input('Bilangan bulat      ')
bilangan = int(nilai)

keterangan = 'GANJIL'
if bilangan % 2 == 0:
    keterangan = 'GENAP'

print(bilangan, 'adalah bilangan', keterangan)

Akhir berkas

```

Hasil pengujian skrip ini diperlihatkan berikut ini:

```

C:\LatOpenCV>python genap.py
Bilangan bulat = 4
4 adalah bilangan GENAP

C:\LatOpenCV>

```

Pada skrip ini, bagian

```
nilai = input('Bilangan bulat = ')
bilangan = int(nilai)
```

digunakan untuk mendapatkan bilangan bulat dari *keyboard*. Lalu, mula-mula keterangan diisi "GANJIL". Kemudian, pernyataan *if* dijalankan. Awalnya, kondisi `bilangan % 2 == 0` dievaluasi. Ekspresi `bilangan % 2` menghasilkan 0 mengingat sisa pembagian 4 dengan 2 adalah 0. Lalu, nilai ekspresi `0 == 0` memberikan hasil `True`. Dengan demikian, pernyataan `keterangan = 'GENAP'` dijalankan. Oleh karena itu, nilai keterangan tidak lagi berupa 'GANJIL', melainkan berupa 'GENAP'.

Sekarang, data yang dimasukkan adalah 3, seperti yang diperlihatkan berikut ini:

```
C:\LatOpenCV>python genap.py
Bilangan bulat = 3
4 adalah bilangan GANJIL

C:\LatOpenCV>
```

Hal ini disebabkan ekspresi `bilangan % 2 == 0` menghasilkan `False` untuk bilangan berupa 3 sehingga pernyataan

```
keterangan = 'GENAP'
```

tidak dieksekusi. Dengan demikian, keterangan tetap berisi 'GANJIL'.

Apabila pernyataan yang mengikuti *if* hanya satu, pernyataan tersebut dapat diletakkan satu baris dengan *if*. Sebagai contoh,

```
if bilangan % 2 == 0:
    keterangan = 'GENAP'
```

boleh ditulis menjadi

```
if bilangan % 2 == 0: keterangan = 'GENAP'
```

Adapun perintah `if` yang mengandung `else` mempunyai bentuk seperti berikut:

```
if kondisi:
    perintah_al

    perintah_an
else:
    perintah_bl

    perintah_bn;
```

Pada bentuk seperti itu,

- `perintah_al` hingga `perintah_an` hanya dijalankan kalau kondisi di `if` bernilai `True`;
- `perintah_bl` hingga `perintah_bn` hanya dijalankan kalau kondisi di `if` bernilai `False`.

Contoh berikut memperlihatkan penggunaan `if..else` untuk menentukan suatu bilangan berupa bilangan genap atau bilangan ganjil.



Berkas : genap2.py

```
#Penentuan genap dan ganjil
# menggunakan if..else

nilai = input('Bilangan bulat : ')
bilangan = int (nilai)

if bilangan % 2 == 0:
    keterangan = 'GENAP'
else:
    keterangan = 'GANJIL'

print(bilangan, 'adalah bilangan', keterangan)
```

Akhir berkas

Pada skrip ini,

```
if bilangan % 2 == 0:
    keterangan    'GENAP'
else:
    keterangan    'GANJIL'
```

digunakan untuk menggantikan

```
keterangan = 'GANJIL'
if bilangan % 2 == 0:
    keterangan = 'GENAP'
```

yang digunakan pada contoh sebelumnya.

3.2.2 Struktur if-else Bertingkat

Apabila pengambilan keputusan melibatkan lebih dari dua pilihan, if bertingkat bisa digunakan. Bentuknya seperti berikut:

```
if kondisi 1:
    perintah_a1

    perintah_an
el.if kondisi 2:
    perintah_b1

    perintah_bn
el.if kondisi x:

el.se:
    perintah_m1

    perintah_mn
```

Skrip berikut dimaksudkan untuk menentukan suatu bilangan berupa bilangan positif, negatif, atau nol.



```
# Contoh penentuan bilangan bilangan
#     positif, negatif, atau nol
#     menggunakan if-elif
```

```
nilai = input('Bilangan bulat      ')
bilangan = int(nilai)
```

```
if bilangan == 0:
    keterangan    'NOL'
elif bilangan > 0:
    keterangan    'POSITIF'
else:
    keterangan    'NEGATIF'
```

```
print(bilangan, '->', keterangan)
```

Akhir berkas

Mula-mula, kondisi bilangan `== 0` diperiksa. Jika bernilai `True`, pernyataan berikut dieksekusi:

```
keterangan = "NOL"
```

Jika `bilangan == 0` bernilai `False`, kondisi `bilangan > 0` dievaluasi. Apabila kondisi ini bernilai `True`, pernyataan berikut dijalankan:

```
keterangan    "POSITIF"
```

Jika `bilangan > 0` bernilai `False`, pernyataan berikut dieksekusi:

```
keterangan    "NEGATIF"
```

Dengan cara seperti itu, ketiga kemungkinan pada bilangan bisa tertangani.

Contoh hasil pengujian untuk tiga keadaan diperlihatkan berikut ini.


```

C:\LatOpenCV>python kategbi1.py
Bilangan bulat = 3
3 -> POSITIF

C:\LatOpenCV>python kategbi1.py
Bilangan bulat = -2
-2 -> NEGATIF

C:\LatOpenCV>python kategbi1.py
Bilangan bulat = 0
0 -> NOL

C:\LatOpenCV>

```

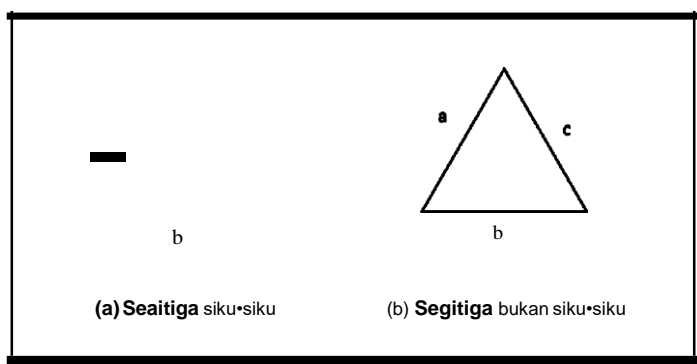
3.2.3 Struktur if-else Bersarang

Pernyataan `if` yang berada di dalam `if` biasa dinamakan `if` bersarang. Pernyataan `if-elif` merupakan kasus khusus pernyataan `if` bersarang dan secara khusus disebut `if` bertingkat. Kasus `if` bersarang yang dibahas di sini bukan yang berbentuk seperti itu. Terkadang dijumpai `if` bersarang yang tidak berbentuk `if` bertingkat. Sebagai contoh, hal ini dijumpai pada persoalan untuk menentukan suatu segitiga tergolong sebagai segitiga siku-siku atau tidak.

Suatu segitiga siku-siku mempunyai sifat seperti berikut:

$$c = \sqrt{a^2 + b^2}$$

Dalam hal ini, c adalah sisi yang terpanjang (Gambar 3.3(a)). Dengan memasukkan nilai ketiga sisi, bagaimana cara untuk menentukan segitiga tersebut sebagai segitiga siku-siku atau bukan?



Gambar 3.3 Jenis segitiga

Berdasarkan pernyataan bahwa pada ketiga sisi pada segitiga siku-siku mempunyai hubungan seperti berikut:

$$c = \sqrt{a^2 + b^2}$$

Algoritma untuk menyelesaikan masalah ini adalah seperti berikut:

Masukan: a, b, c

Keluaran: segitigaSikuSiku (berupa nilai TRUE atau FALSE)

1. segitigaSikuSiku FALSE
2. toleransi 0.00001
3. jumlahKuadrat a * a + b * b
4. cKuadrat c * c

JIKA (jumlahKuadrat >= cKuadrat - Toleransi DAN
jumlahKuadrat <= CKuadrat + Toleransi)
segitigaSikuSiku TRUE

SEBALIKNYA

jumlahKuadrat A* A+ C * C
bKuadrat B * B

JIKA (jumlahKuadrat >= bKuadrat - toleransi DAN
jumlahKuadrat <= bKuadrat + toleransi)
segitigaSikuSiku TRUE

SEBALIKNYA

jumlahKuadrat b * b + c * c

```

aKuadrat = a * a
JIKA (jumlahKuadrat >= aKuadrat - toleransi &&
    jumlahKuadrat <= aKuadrat + toleransi)
    segitigaSikuSiku = TRUE

```

AKHIR-JIKA

AKHIR-JIKA

AKHIR-JIKA

Penggunaan toleransi (yang berupa nilai yang sangat kecil) di sini dimaksudkan untuk mengantisipasi penghitungan $c = \sqrt{a^2 + b^2}$ yang belum tentu menghasilkan nilai yang secara eksak sama dengan c^2 mengingat komputer tidak dapat menyimpan nilai real secara eksak pada memori komputer. Oleh karena itu, $c = \sqrt{a^2 + b^2}$ berkemungkinan besar tidak sama dengan c^2 , tetapi berupa nilai yang sangat dekat sekali dengan c^2 . Dengan perkataan lain, $c = \sqrt{a^2 + b^2}$ akan bernilai antara $c^2 - \text{toleransi}$ dan $c^2 + \text{toleransi}$.

Algoritma di depan dapat diubah ke skrip Python seperti berikut:

11!1 **Berkas : :eno1112.11y**

```

# Contoh penentuan segitiga berupa
# segitiga siku-siku atau bukan

print('Pernasukan tiga sisi dalam segitiga')
nilai = input('Sisi pertama = ')
sisiA = float(nilai)

nilai = input('Sisi kedua = ')
sisiB = float(nilai)

nilai = input('Sisi ketiga = ')
sisiC = float(nilai)

segitigaSikuSiku = False
toleransi = 0.00001
jumlahKuadrat = sisiA * sisiA + sisiB * sisiB
cKuadrat = sisiC * sisiC
if (jumlahKuadrat >= cKuadrat - toleransi) and\

```

```

    (jumlahKuadrat <= cKuadrat + toleransi):
        segitigaSikuSiku = True
else:
    jumlahKuadrat <- sisiA * sisiA + sisiC * sisiC
    bKuadrat = sisiB * sisiB
    if (jumlahKuadrat >= bKuadrat - toleransi) and\
        (jumlahKuadrat <= bKuadrat + toleransi):
        segitigaSikuSiku = True
    else:
        jumlahKuadrat = sisiB * sisiB + sisiC * sisiC
        aKuadrat = sisiA * sisiA
        if (jumlahKuadrat >= aKuadrat - toleransi) and\
            (jumlahKuadrat <= aKuadrat + toleransi):
            segitigaSikuSiku = True

# Tampilkan hasil
if segitigaSikuSiku:
    print('Segitiga siku-siku')
else:
    print('Segitiga bukan siku-siku')

```

Akhir berkas

Skrip ini menguji dua segitiga, yang pertama adalah yang memiliki sisi 3, 4, dan 5, dan yang kedua mempunyai sisi 3, 5, dan 6. Hasilnya diperlihatkan berikut ini:

```

C:\LatOpenCV>python sikusiku.py<P
Pemasukan tiga sisi dalam segitiga
Sisi pertama = 3<P
Sisi kedua = 4<P
Sisi ketiga = 5<P
Segitiga siku-siku

```

```

C:\LatOpenCV>python sikusiku.py<P
Pemasukan tiga sisi dalam segitiga
Sisi pertama = 3<P
Sisi kedua = 4<P
Sisi ketiga = 6<P
Segitiga bukan siku-siku

```

```

C:\LatOpenCV>

```


3.2.4 Perulangan dengan while

Perintah while memiliki bentuk pemakaian seperti berikut:

```
while kondisi:  
    pernyataan_l  
  
    pernyataan_n  
else:  
    pernyataan_ml  
  
    pernyataan_mn ]
```

Dalam hal ini, *pernyataan_l* hingga *pernyataan_n* akan dijalankan secara berulang selama *kondisi* pada while bernilai True. Akan tetapi, karena kondisi diuji di depan, ada kemungkinan bahwa pernyataan-pernyataan tersebut tidak dijalankan sama sekali. Bagian else bersifat opsional. Jika bagian ini disertakan, *pernyataan_ml* hingga *pernyataan_mn* akan dijalankan terakhir kali tepat sebelum while berakhir.

Contoh penggunaan while dapat dilihat pada skrip berikut:

 **Berkas : genap2.py**

```
# Penggunaan while  
#   untuk menampilkan bilangan bulat  
#   1 hingga 10  
  
bilangan = 1  
while bilangan <= 10:  
    print(bilangan)  
    bilangan = bilangan + 1  
  
Akhir berkas
```

Skrip ini memberikan contoh penggunaan while untuk menampilkan bilangan bulat 1 hingga 10. Hasilnya seperti berikut:

```
.....  
C:\LatOpenCV>python bil1sd10.py<P
```

```
1  
2  
3  
4  
5  
6  
7  
8  
9  
10
```

```
C:\LatOpenCV>
```

Secara prinsip, terdapat tiga bagian pengendali dalam perulangan menggunakan `while` pada contoh di depan. Bagian pertama adalah variabel yang bertindak sebagai pencacah. Variabel yang dimaksud adalah `bilangan`. Variabel ini disebut pencacah karena akan menghitung dari 1, 2, 3, dan seterusnya. Variabel ini perlu diberi nilai awal untuk mencacah. Perintah yang digunakan untuk keperluan ini adalah:

```
bilangan = 1
```

Bagian kedua yaitu kondisi yang digunakan untuk meneruskan pengulangan atau menghentikan perulangan. Bagian ini terletak pada kondisi `while`. Pada contoh di depan berupa:

```
bilangan <= 10
```


Eksekusi pada bagian berikut dilakukan terus-menerus selama kondisi ini bernilai `True`:

```
print(bilangan)  
bilangan = bilangan + 1
```

Adapun bagian ketiga adalah bagian yang digunakan untuk menaikkan variabel `pencacah`. Pada contoh di depan, bagian ini berupa:

```
bilangan = bilangan + 1
```

Dengan memanipulasi ketiga bagian pengendali `while`, dimungkinkan untuk menampilkan bilangan bulat dari 10 menuju 1. Perwujudannya seperti berikut:

 Berkas : `genap2.py`

```
# Penggunaan while
#   untuk menampilkan bilangan bulat
#   10, 9, 8, 7, 6, 5, 4, 3, 2, 1

bilangan = 10
while bilangan > 0:
    print(bilangan)
    bilangan = bilangan - 1

Akhir berkas
```

Mula-mula, variabel `bilangan` diisi dengan 1. Selanjutnya, kondisi yang digunakan pada `while` berupa `bilangan > 0`. Artinya, perulangan dilakukan selama `bilangan` bernilai lebih besar daripada 0. Adapun bagian yang mengontrol pencacahan secara menurun berupa:

```
bilangan = bilangan - 1
```

Perintah ini digunakan untuk mengurangi nilai pada `bilangan` sebesar satu.

Hasil pemanggilan skrip `bil10sd1.py` adalah seperti berikut:

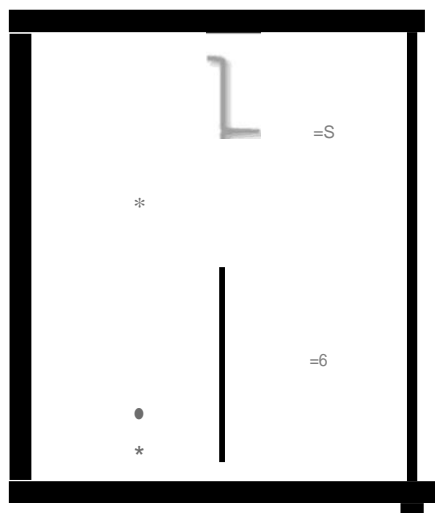
```

.....
C:\LatOpenCV>python billOsd1.py
10
9
8
7
6
5
4
3
2
1

C:\LatOpenCV>

```

Dalam praktik, sering dijumpai persoalan yang melibatkan while yang ditulis di dalam while, atau disebut while bersarang. Sebagai contoh, hal ini dijumpai pada persoalan untuk menampilkan segitiga bintang yang tingginya bisa ditentukan, sebagaimana diperlihatkan pada Gambar 3.4.



Gambar 9.4 Segitiga bintang

Dengan melihat contoh yang diberikan, terlihat bahwa tinggi segitiga sebanyak N. Pada baris pertama, jumlah bintang sebanyak 1. Pada baris kedua, jumlah bintang sebanyak 2, dst. Dengan demikian, jumlah bintang dalam suatu baris bergantung pada nomor baris, yakni sebanyak nomor baris. Berdasarkan hal ini, terdapat dua `while` untuk menyelesaikan masalah ini. Perintah `while` pertama menangani baris dan `while` kedua menangani jumlah bintang dalam setiap baris.



dannya adalah seperti berikut:

Berkas : segitiga.py

```
# Penggunaan while bersarang
#   untuk menyusun segitiga bintang

nilai = input('Tinggi segitiga =')
tinggi = int(nilai)

baris = 1
while baris <= tinggi:
    kolom = 1;
    while kolom <= baris:
        print( '*', end='')
        kolom = kolom + 1

    #Pindah baris
    print()
```

```
baris = baris + 1
```

Akhir berkas

Pembuatan segitiga bintang ditangani dengan menggunakan `while` di dalam `while`. Perintah yang digunakan untuk menangani semua baris yaitu:

```
baris = 1;
while baris <= tinggi:

    baris    baris + 1
```

Perintah yang digunakan untuk menampilkan sejumlah bintang dalam satu baris yaitu:

```
kolom = 1;
while kolom <= baris:
    print('*', end= ' ')
    kolom = kolom + 1
```

Perintah

```
print('*', end= ' ')
```

digunakan untuk menampilkan * tanpa berpindah baris. Perlakukan ini diatur oleh argumen `end = ' '`.

Adapun perintah berikut digunakan untuk menambahkan karakter pindah baris yang diperlukan setelah semua bintang dalam satu baris ditampilkan:

```
print()
```

Berikut menunjukkan contoh hasil pemanggilan skrip `segitiga.py`:

```
.....
C:\LatOpenCV>python segitiga.py
Tinggi segitiga = 6
*
**
***
****
*****
*****

C:\LatOpenCV>
```

Sekarang, bagaimana cara membuat segitiga dengan bentuk berikut?

```
*****
*****
*****
***
*
```

Hasil tersebut adalah untuk tinggi segitiga sebesar 5.

Persoalan ini dapat disajikan dalam bentuk algoritma seperti berikut:

1. Masukkan Tinggi
2. UNTUK Baris= 1 HINGGA Tinggi
 Sisipkan sebanyak Baris-1 karakter spasi pada baris yang sama
 Sisipkan sebanyak $2 * (\text{Tinggi} - \text{Baris}) + 1$ tanda *
 Pindah baris
AKHIR-UNTUK

Perwujudannya dengan while adalah seperti berikut:



```
# Penggunaan while bersarang
# untuk menyusun segitiga bintang

nilai = input('Tinggi segitiga = ')
tinggi = int(nilai)

baris = 1
while baris <= tinggi:
    # Tampilkan sebanyak baris - 1 spasi
    pencacah = 1;
    while pencacah <= baris - 1:
        print(chr(32), end= '')
        pencacah = pencacah + 1

    # Tampilkan 2 x (tinggi - baris) + 1 tanda *
    pencacah = 1;
    while pencacah <= 2 * (tinggi - baris) + 1:
        print('*', end= '')
        pencacah = pencacah + 1

    #Pindah baris
    print()
```

```
baris = baris + 1
```

Akhir berkas

Pada skrip ini, chr (32) menyatakan karakter spasi. Hasil pengujiannya diperlihatkan berikut ini:

```
C:\LatOpenCV>python segitiga2.py
Bilangan bulat = 6
*****
*****
*****
*****
***
*
```

C:\LatOpenCV>

Contoh di depan menunjukkan penggunaan `while` berdasarkan algoritma yang diberikan. Dalam praktik, berdasarkan algoritma tersebut dapat dibentuk skrip yang lebih kompak. Bagian

Sisipkan sebanyak Baris-1 karakter spasi pada baris yang sama
 Sisipkan sebanyak $2 * (\text{Tinggi} - \text{Baris}) + 1$ tanda `*`

dapat diselesaikan dengan menggunakan operator `*` yang dikenakan pada string. Berikut adalah kode seperti berikut:

11!1 Berkas: segitiga3.py

```
# Penggunaan while bersarang
# untuk menyusun segitiga bintang
# Versi 2

nilai = input('Tinggi segitiga ')
tinggi = int(nilai)

baris = 1;
while baris <= tinggi:
    # Tampilkan sebanyak baris - 1 spasi
    print(chr(32) * (baris - 1), end = ' ')

    # Tampilkan 2 x (tinggi - baris) + 1 tanda *
    print('*' * (2 * (tinggi - baris) + 1))

    baris = baris + 1
```

Akhir berkas

Catatan

Jika karena sesuatu hal membuat perulangan tidak pernah berhenti, tekanlah

CTR+C

untuk menghentikan perulangan yang tidak pernah berhenti.

3.2.5 Perulangan dengan for

Pernyataan `for` berguna untuk melakukan pengulangan suatu proses yang didasarkan pada nilai elemen-elemen senarai (Subbab 3.5.1).

Bentuk penggunaannya seperti berikut:

```
for variabel in list:
    pernyataan_al

    pernyataan_an
[else:
    pernyataan_bl

    pernyataan_bn
```

Bagian `pernyataan_a1` hingga `pernyataan_an` akan dijalankan sebanyak nilai elemen dalam `list`. Dalam hal ini, untuk setiap iterasi (satu putaran eksekusi pada `pernyataan_a1` hingga `pernyataan_an`), `variabel` akan bernilai sama dengan salah satu elemen dalam `list`. Bagian `pernyataan_b1` hingga `pernyataan_bn` akan dijalankan terakhir kali.

Contoh penggunaan `for` dapat diuji secara interaktif seperti berikut:

```
>>> for x in [4, 5, 8, 9]: 4)
      print(x) 4)

4
5
8
9
>>>
```

Pada contoh ini terdapat empat iterasi mengingat jumlah elemen dalam senarai terdapat 4.

- Pada iterasi pertama, x bernilai 4.
- Pada iterasi pertama, x bernilai 5.
- Pada iterasi pertama, x bernilai 8.
- Pada iterasi pertama, x bernilai 9.

Nilai pada senarai [4, 5, 8, 9] menyatakan daftar nilai untuk iterasi masing-masing.

Pernyataan `for` dapat diterapkan pada string. Setiap karakter dapat diambil melalui `for`. Contoh:

```
>>> for x in "ABCDE":4)
      print(x) 4)

A
B
C
D
E
>>>
```

3.2.6 Perintah `break` dan `continue`

Pernyataan `break` tidak hanya digunakan pada `repeat`, tetapi juga bisa dikenakan pada `for` maupun `while` untuk mengakhiri

perulangan. Jika diterapkan pada kalang yang bersarang, menyebabkan kalang paling dalam diakhiri.

break

skrip berikut menunjukkan penggunaan break pada for:



Berkas : !:e itio.11'

```
#Contoh break pada for
```

```
for bilangan in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:  
    if bilangan == 5:  
        break
```

```
print(bilangan)
```



Akhir berkas

Pada skrip ini, perintah berikut membuat perulangan for ketika bilangan bernilai 5:

diakhiri

```
if bilangan == 5:  
    break
```

Dengan demikian, nilai terakhir yang ditampilkan via `print()` adalah 4.

Hal ini diperlihatkan pada contoh berikut:

```
>>> for x in "ABCDE":  
        print(x)
```

A

B

C

D

E

>>>

Pada Python, pernyataan `continue` disediakan untuk mengabaikan iterasi yang sedang berlangsung pada perulangan. Untuk memahami hal ini, skrip berikut yang menunjukkan penggunaan `continue` pada `for` bisa dicoba.

#Contoh next pada for

```
for bilangan in [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]:
    if bilangan == 5:
        continue
```

Akhir berkasngan)

Pada skrip ini, perintah berikut membuat iterasi yang sedang berlangsung pada perulangan `for` diabaikan ketika `bilangan` bernilai 5:

```
if bilangan == 5:
    continue
```

Dengan demikian, nilai 5 tidak pernah ditampilkan mengingat `print()` diabaikan ketika `bilangan` bernilai 5.

```
.....11.....
C:\LatOpenCV>python nextfor.py
1
2
3
4
6
7
8
9
10

C:\LatOpenCV>
```

3.3 Pembuatan Fungsi

Kadangkala, fungsi perlu dibuat sendiri. Hal ini dimaksudkan untuk mempersingkat kode terutama kalau sejumlah perintah yang sama dijalankan di beberapa tempat.


```
.....  
C:\LatOpenCV>python robot1.py
```

Robot pertama:

```
xx      xx
```

```
ll      ll  
ll      ll
```

Robot kedua:

```
xx      xx
```

```
ll      ll  
ll      ll
```

```
C:\LatOpenCV>
```

Baris-baris berikut terdapat dua kali pada skrip tersebut:

```
print('----- ')  
print(' |  xx      xx |')  
print(' |              |')  
print(' |              |')  
print(' |              |')  
print('----- ')  
print('  ||          || |')  
print('  ||          || |')  
print('  |              |')
```

Nah, baris-baris tersebut dapat dijadikan sebagai fungsi. Penulisannya seperti berikut:

Dengan adanya definisi fungsi `tampilkanRobot()`, fungsi tersebut dapat dipanggil dengan cara seperti berikut:

```
tampilkanRobot()
```

Dengan menggunakan fungsi `tampilkanRobot()`, skrip robot.l



seperti berikut:

```
# Robot versi 2
```

```
def tampilkanRobot():
    print('-----')
    print(' |  xx      xx  |')
    print(' |                      |')
    print(' |                      |')
    print(' |                      |')
    print('-----')
    print('  ||          ||  ')
    print('  ||          ||  ')
    print('                      •')
```

```
print('Robot pertama:')
tampilkanRobot()
```

```
print()
```

```
print('Robot kedua:')
tampilkanRobot()
```

Akhir berkas

Definisi fungsiberupa:

```
def tampilkanRobot():
    print('-----')
    print(' |  xx      xx  |')
    print(' |                      |')
    print(' |                      |')
    print(' |                      |')
    print('-----')
    print('  ||          ||  ')
    print('  ||          ||  ')
    print('                      •')
```

Secara prinsip, definisi fungsi paling dasar diawali kata-tercadang `def` dan diikuti dengan nama fungsi dan `()` : . Selebihnya berupa blok kode yang akan dijalankan ketika fungsi dipanggil.

Hasil skrip ini sama dengan hasil skrip `robot1.r`.

3.3.2 Pembuatan Argumen

Contoh fungsi `tampilkanRobot()` di depan tidak melibatkan argumen. Nah, bagaimana seandainya dikehendaki untuk menyertakan argumen dalam fungsi? Untuk mempraktikkan hal ini, dapat kita tambahkan argumen bernama `judul` yang digunakan untuk melewati judul robot. Fungsi yang diperlukan adalah seperti berikut:

```
def tampilkanRobot(judul):
    print (judul)
    print(" ----- ")
    print("I   xx   xx  I")
    print(" |                               |")
    print(" |                               |")
    print("I                               I")
    print(" ----- ")
    print("    11          11  ")
    print("    11          11  ")
    print("                    ")
```

Perhatikan bahwa di dalam `def` tanda kurung tepat setelah nama fungsi terdapat `judul`, yang merupakan argumen fungsi. Tambahan lain berupa baris berikut:

```
    print (judul)
```

Bagian inilah yang menampilkan judul robot.



ng digunakan untuk mewujudkan hal ini dapat dilihat berikut ini.

Berkas : robot3.py

```
# Robot versi 3

def tampilkanRobot(judul):
    print(judul)
    print('-----')
    print(' |  xx      xx  |')
    print(' |              |')
    print(' |              |')
    print(' |              |')
    print(' |              |')
    print('-----')
    print('  ||          ||  ')
    print('  ||          ||  ')
    print('                •')

tampilkanRobot('Robot pertama:')

print()
```

tampilkanRobot('Robot kedua:')

Akhir berkas

Tampak bahwa karena fungsi `tampilkanRobot()` memiliki satu argumen, pemanggilannya adalah semacam berikut:

```
tampilkanRobot('Robot pertama')
```

3.3.3 Penyertaan Nilai Balik

Nilai balik suatu fungsi ditentukan melalui `return()`. Sebagai contoh, fungsi `ganjil()` memberikan nilai balik `True` kalau argumen bilangan berupa bilangan ganjil.



Berkas : segitiga.py

```
# Perwujudan fungsi dengan nilai balik
```

```
def ganjil(bilangan):
    hasil = bilangan % 2
```

```
return(hasil == 1)
```

```
# Bagian pemanggilan fungsi  
print("5 adalah ganjil?", ganjil(5))  
print("6 adalah ganjil?", ganjil(6))
```

Akhir berkas

Pada definisi fungsi ganjil (), perintah berikut digunakan untuk menghasilkan sisa pembagian bilangan dengan 2:

```
hasil = bilangan % 2
```

Dengan demikian, nilai di variabel **hasil** akan selalu berupa 0 atau 1. Nilai 0 diperoleh kalau bilangan berisi bilangan genap dan nilai 1 diperoleh jika bilangan berisi bilangan ganjil. Berdasarkan fakta ini, maka `hasil == 1` menghasilkan True kalau **hasil** berisi 1 atau False kalau **hasil** berisi 0. Itulah sebabnya, perintah

```
return(hasil == 1)
```

dijadikan sebagai nilai balik fungsi ganjil (), yang memberikan nilai balik True kalau argumen bilangan berisi bilangan ganjil atau False kalau bilangan berisi bilangan genap. Hasil pengujian diperlihatkan berikut ini.

```
.....  
C:\LatOpenCV>python ganjil.py<P  
5 adalah ganjil? True  
6 adalah ganjil? False
```

```
C:\LatOpenCV>
```

3.3.4 Nilai Bawaan Argumen

Nilai bawaan untuk argumen bisa ditentukan dengan menyebutkan nilainya pada definisi fungsi. Contoh diperlihatkan pada skrip berikut.



```
# Contoh argumen dengan nilai bawaan

def ulang(st, jumlah = 10): print(st * jumlah)

# Bagian utama
ulang('-')
ulang('-', 15)
ulang('+', 5)
```

Akhir berkas

Pada definisi fungsi `ulang()`, `jumlah = 10` pada bagian argumen menyatakan bahwa kalau argumen ini tidak disebutkan, nilai yang digunakan adalah 10. Dengan demikian,

```
ulang('-')
```

identik dengan

```
ulang('-', 10)
```

Pada definisi fungsi tersebut, perintah berikut digunakan untuk mendapatkan string yang merupakan perulangan isi `st` sebanyak `jumlah`:

```
st* jumlah
```

Berikut menunjukkan hasil pemanggilan skrip bawaan .py:

```
.....
C:\LatOpenCV>python bawaan.py<P

+++++

C:\LatOpenCV>
```

Catatan

Apabila suatu fungsi mengandung hanya satu pernyataan, pernyataan tersebut bisa

dijadikan satu baris dengan baris yang berisi
def. Hal itu terlihat pada:

```
def ulang(st, jumlah 10): print(st  
* jumlah)
```

3.3.5 Variabel Lokal dan Global

Variabel yang terdapat pada fungsi dengan sendirinya berkedudukan sebagai variabel lokal, yakni variabel yang hanya dikenali di fungsi tempat variabel tersebut berada. Variabel seperti ini tidak mempengaruhi nilai variabel di luar fungsi walaupun bernama sama.



memahami hal ini, perhatikan skrip berikut:

Berkas : lokal.py

```
#Contoh untuk melihat efek variabel lokal

def gantiMobil ():
    mobil = 'Honda Jazz'

    print('Isi mobil di gantiMobil():', mobil)

#Bagian utama
mobil = 'Toyota Avanza'

print('Isi mobil semula : ' + mobil)

gantiMobil()
print('Isi mobil sekarang: ' + mobil)
```



Akhir berkas

Perhatikan bahwa variabel mobil mula-mula diisi dengan 'Toyota Avanza'. Kemudian, fungsi gantiMobil () dipanggil. Di dalam fungsi tersebut terdapat perintah:

```
mobil = 'Honda Jazz'
```

Namun, apa yang terjadi setelah perintah ini dijalankan? Apakah nilai 'Toyota Avanza' yang disimpan di variabel mobil di luar fungsi akan

berubah? Contoh sesudah ini memperlihatkan kenyataannya. Tampak bahwa nilai `mobil` yang semula berupa 'Toyota Avanza' tidak berubah. Hal ini terjadi karena `mobil` pada fungsi `gantiMobil()` dan di luar fungsi menyatakan dua variabel yang berbeda. Pada fungsi `gantiMobil()`, `mobil` pada perintah

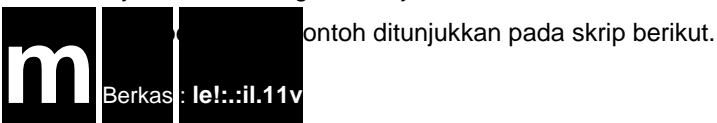
```
mobil = 'Honda Jazz'
```

menyatakan variabel lokal, sedangkan yang berada di luar fungsi adalah variabel global.

```
C:\LatOpenCV>python lokal.py<P
Isi mobil semula : Toyota Avanza
Isi mobil di gantiMobil(): Honda Jazz
Isi mobil sekarang: Toyota Avanza
```

```
C:\LatOpenCV>
```

Pertanyaan berikutnya yang terkadang muncul adalah "Apakah berarti bahwa fungsi tidak bisa mengakses variabel global?" Jawabannya adalah bisa, walaupun cara seperti ini tidak dianjurkan untuk digunakan. Lalu, bagaimana cara mengakses variabel global dari suatu fungsi? Jawabannya adalah dengan menyebutkan variabel tersebut di fungsi



```
# Contoh untuk melihat efek
#   pengubahan variabel global

def gantiMobil():
    global mobil
    mobil = 'Honda Jazz'

    print('Isi mobil di gantiMobil(): ' + mobil)

# Bagian utama
mobil = 'Toyota Avanza'
```

```
print('Isi mobil semula  :', mobil)

gantiMobil()
print('Isi mobil sekarang:', mobil)
```

Akhir berkas

Pada skrip ini, perintah

```
mobil = 'Honda Jazz'
```

terdapat pada definisi fungsi `gantiMobil()` sama seperti pada skrip `lokal.py`. Akan tetapi, sebelum pernyataan tersebut terdapat pernyataan:

```
global mobil
```

Perintah ini menyatakan bahwa `mobil` adalah variabel global. Dengan demikian,

```
mobil = 'Honda Jazz'
```

menyatakan bahwa 'Honda Jazz' ditugaskan ke variabel global bernama `mobil`, bukan variabel lokal bernama `mobil`. Dengan demikian, variabel `mobil` yang terletak di luar definisi fungsi `gantiMobil()` bisa diubah. Hal ini ditunjukkan pada hasil eksekusi skrip `global.py` berikut ini.

```
-----
C:\LatOpenCV>python global.py<P
Isi mobil semula  : Toyota Avanza
Isi mobil di gantiMobil():  Honda Jazz
Isi mobil sekarang: Honda Jazz

C:\LatOpenCV>
```

3.3.6 Pengubahan Nilai Argumen di dalam Fungsi

Nilai argumen fungsi bisa diubah atau tidak bergantung pada jenis objek yang dilewatkannya. Jika objek yang dilewatkan bertipe seperti `int`, `float`, `bool`, `str`, `tuple`, `unicode`, nilai argumen tidak bisa

diubah dari dalam fungsi. Objek seperti itu dikenal sebagai objek yang tidak dapat diubah setelah diciptakan (*immutable object*). Akan tetapi, argumen berupa objek bertipe **set**, **list**, dan **dict** (yang tergolong sebagai *mutable object* atau objek yang dapat diubah setelah diciptakan) dapat diubah di dalam fungsi.

Skrip berikut menunjukkan bahwa argumen string tidak dapat diubah di dalam fungsi:



Berkas : lek.il.11y

```
# Contoh untuk melihat efek
#   pengubahan nilai argumen

def ubah(bunga):
    bunga = 'Melati'

    print('Isi bunga di ubah :', bunga)
    return(bunga)

# Bagian utama
bunga = 'Mawar'

print('Isi bunga semula  :', bunga)
ubah(bunga)
print('Isi bunga sekarang:', bunga)
```

Akhir berkas

Pada definisi fungsi `ubah()`, perintah berikut digunakan untuk mengubah nilai argumen `bunga`:

```
bunga = 'Melati'
```

Namun, apa yang terjadi ketika skrip `ubaharg.py` dieksekusi? Berikut menunjukkan hasilnya.

```
.....  
C:\LatOpenCV>python ubahstr.py
```

```
Isi bunga semula : Mawar  
Isi bunga di ubah(): Melati  
Isi bunga sekarang: Mawar
```

```
C:\LatOpenCV>
```

Tampak bahwa variabel `bunga` yang digunakan pada pemanggilan fungsi `ubah()` tidak dapat diubah oleh fungsi. Tampak bahwa nilai `bunga` sebelum dan sesudah pemanggilan fungsi `ubah()` tetap sama.

Skip berikut menunjukkan bahwa argumen bertipe `list` dapat diubah di dalam fungsi:

11!1 Berkas : l.k.il.11y

```
# Contoh untuk melihat efek  
# pengubahan nilai argumen 0  
  
def ubah(bunga):  
    bunga[1] = 'Kana'  
  
    print('Isi bunga di ubah():', bunga)  
    return(bunga)
```

```
# Bagian utama  
bunga = ['Melati', 'Mawar', 'Kenanga']  
  
print('Isi bunga semula :', bunga)  
ubah(bunga)  
t('Isi bunga sekarang:', bunga)
```

Akhir berkas

Pada definisi fungsi `ubah()`, perintah berikut digunakan untuk mengubah nilai salah satu elemen di `bunga`:

```
bunga[1] = 'Kana'
```

Perintah ini digunakan untuk mengubah elemen pada `bunga` yang berindeks 1 dengan 'Kana'. Berikut menunjukkan hasilnya.

```
C:\LatOpenCV>python ubahlist.py<P
Isi bunga semula : ['Melati', 'Mawar', 'Kenanga']
Isi bunga di ubah(): ['Melati', 'Kana', 'Kenanga']
Isi bunga sekarang: ['Melati', 'Kana', 'Kenanga']

C:\LatOpenCV>
```

Tampak bahwa variabel `bunga` di fungsi dan sesudah pemanggilan fungsi bernilai sama. Artinya, nilai variabel yang dilewatkan sebagai argumen di fungsi bisa diubah di dalam fungsi.

3.4 Pembuatan Kelas

Subbab ini mengenalkan cara untuk membuat kelas.

3.4.1 Dasar Pemhuatan Kelas

Suatu kelas dapat dibuat sendiri dengan format seperti berikut:

```
class NamaKelas:
    pernyataan_l

    pernyataan_n
```

Secara konvensional, setiap kata pada nama kelas ditulis dengan huruf kapital, sedangkan yang lain berhuruf kecil.

Sebagai contoh, Anda bisa menuliskan perintah seperti berikut untuk membuat kelas yang mengandung properti `nomorPolisi` dan `warna`:

```
>>> class Mobil:<P
        nomorPolisi = "AB2345BA"<P
        warna = "Biru"<P

>>>
```

Setelah kelas Mobil dibuat, objek berkelas Mobil bisa diciptakan.

Contoh:

```
>>> mobA = Mobil()
>>> print(mobA.nomorPolisi)
AH2345BA
>>> print(mobA.warna)
Biru
>>>
```

Pada contoh ini, pernyataan berikut digunakan untuk menciptakan objek bernama mobA. Setelah itu, properti nomorPolisi dan warna secara berturut-turut dapat diakses melalui objek mobA dengan menuliskan seperti berikut:

```
mobA.nomorPolisi
mobA.warna
```

Sebagai contoh,

```
print(mobA.nomorPolisi)
```

digunakan untuk menampilkan nilai nomorPolisi pada objek mobA.

Nilai pada properti juga bisa diubah. Hal ini diperlihatkan pada contoh berikut:

```
.....
• >>> mobA.warna = "Merah"
  >>> print(mobA.warna)
  Merah
  >>>
;.....
```

3.4.2 Pemhuatan Kelas dengan Melewatkan Argumen untuk Properti

Suatu kelas umumnya dibuat dengan memungkinkan untuk memberikan nilai kepada properti-propertinya sewaktu objek diciptakan. Sebagai contoh, suatu kelas bernama Orang dan

mempunyai properti bernama nama dan usia. Nah, objek dapat dibuat dengan menuliskan perintah semacam berikut:

```
pemrogram = Orang('Karyadi', 23)
staffHRD = Orang('Sinta', 19)
```

Bentuk seperti itu memerlukan konstruktor. Dalam hal ini, Python menyediakan konstruktor bernama `__init__()`.

Nah, cara mewujudkan konstruktor untuk kasus di atas beserta definisi kelas ditunjukkan pada seperti berikut:

m

Berkas : l.k.il.11.1

```
# Contoh kelas dengan konstruktor

# Definisi kelas
class Orang:
    def __init__(self, nama, usia):
        self.nama = nama
        self.usia = usia

    def info(self):
        print('-----');
        print('Nama:', self.nama)
        print('Usia:', self.usia)

# Bagian utama

pemrogram = Orang('Karyadi', 23)
staffHRD = Orang('Sinta', 19)

pemrogram.info()
staffHRD.info()
```

Akhir berkas

Pada definisi kelas `Orang`, terdapat dua definisi metode. Metode pertama berupa `__init__()` dan kedua berupa `info()`. Metode `__init__()` berguna untuk memberikan nilai nama dan usia ketika objek dibentuk, sedangkan metode `info()` digunakan untuk menampilkan data nama dan usia. Pada metode `__init__()`,

`self.nama` menyatakan properti nama dan `self.usia` menyatakan properti usia. Hal yang sama berlaku pada metode `info ()`.

Berikut adalah contoh hasil eksekusi skrip `orang.py`:

```
.....  
C:\LatOpenCV>python orang.py<P
```

```
Nama: Karyadi
```

```
Usia: 23
```

```
Nama: Sinta
```

```
Usia: 19
```

```
C:\LatOpenCV>
```

3.5 Pengenalan Struktur Data

Subbab ini memperkenalkan struktur senarai, tupel, kamus, dan himpunan. Perlu diketahui, struktur data adalah suatu objek yang digunakan untuk menyimpan banyak data.

3.5.1 Senarai

Senarai (list) adalah struktur data yang bisa mengandung banyak elemen yang berbeda tipe dan memungkinkan penambahan atau pengurangan elemen-elemennya. Bahkan, suatu senarai bisa mengandung elemen senarai.

Senarai ditulis dalam tanda kurung siku (`[]`) dan antareleman ditulis dengan pemisah berupa koma. Contoh:

```
[1, 2, 4, 5, 7]
```

Senarai ini memiliki lima elemen dengan masing-masing berupa bilangan. Contoh lain:

```
['Avanza', 'Biru', 2018, 1300]
```


Contoh ini digunakan untuk mencatat data kendaraan yang mencakup tipe, warna, tahun produksi, dan CC. Perhatikan bahwa tipe elemen-elemennya tidak sama.

Penugasan dan Penampilan Senarai

Senarai dapat ditugaskan ke variabel seperti kalau menugaskan bilangan ke variabel. Selain itu, senarai juga bisa ditampilkan melalui `print()`.

Contoh:

```

.....
:>>> dataMobil = ['Avanza', 'Biru', 2018, 1300] //
>>> print(dataMobil) <P
['Avanza', 'Biru', 2018, 1300]
:>>>
.....

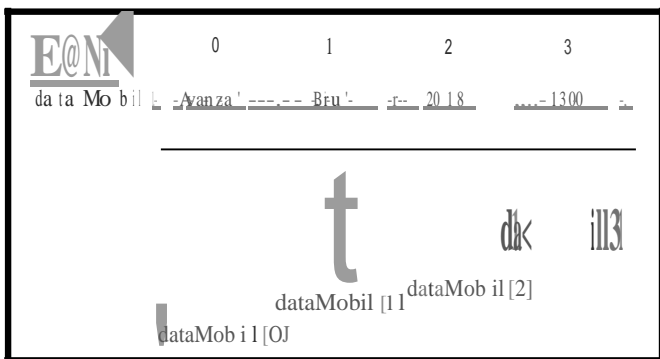
```

Pengaksesan Elemen Senarai

Elemen pada senarai dapat diakses secara individual melalui notasi seperti berikut:

variabel_list[indeks]

Dalam hal ini, *indeks* pada senarai dimulai dari nol. Gambar 3.5 memperlihatkan senarai `dataMobil`, elemen-elemennya, beserta indeks elemen masing-masing.



Gambar 9.5 Senarai dan indeks elemen

Contoh berikut menunjukkan pengaksesan setiap elemen pada senarai `dataMobil`:

```
>>> dataMobil = ['Avanza', 'Biru', 2018, 1300] <P
>>> dataMobil[0] <P
'Avanza'
>>> dataMobil[1] <P
'Biru'
>>> dataMobil[2] <P
2018
>>> dataMobil[3] <P
1300
>>> dataMobil[0] = 'Xenia' <P
>>> print(dataMobil) <P
['Xenia', 'Biru', 2018, 1300]
>>>
```

Elemen senarai dapat diakses dengan nilai negatif. Indeks -1 berarti data terakhir, indeks -2 berarti data kedua sebelum yang terakhir, dst.

Contoh:

```
>>> dataMobil = ['Avanza', 'Biru', 2018, 1300] <P
>>> dataMobil[-1] <P
1300
>>> dataMobil[-2] <P
2018
```

Pengubahan Data pada Elemen Senarai

Di depan telah dijelaskan mengenai cara mengubah elemen senarai, yaitu berupa pernyataan:

```
dataMobil[0] = 'Xenia'
```

Hal ini merupakan contoh untuk mengubah elemen berindeks 0 dengan string 'Xenia'. Dalam praktik, pengubahan dapat dilakukan terhadap sejumlah data sekaligus melalui irisan. Notasi irisan berupa:

```
m:n
```

Notasi ini menyatakan dari indeks m hingga indeks n-1. Contoh:

```
>>> dataMobil = ['Avanza', 'Biru', 2018, 1300] <P
>>> print(dataMobil) <P
['Avanza', 'Biru', 2018, 1300]
>>> dataMobil[0:2] = ['Xenia', 'Hitam']<P
>>> print(dataMobil) <P
['Xenia', 'Hitam', 2018, 1300]
>>>
```

Pada contoh ini, elemen dengan indeks 0 dan 1 (dinyatakan dengan irisan 0:2) secara berturut-turut diganti dengan 'Xenia' dan 'Hitam'.

Cara Mendapatkan Jumlah Elemen

Fungsi `len()` biasa digunakan untuk mendapatkan jumlah elemen dalam suatu senarai. Contoh:

```
>>> dataMobil = ["Avanza", "Biru", 2018, 1300] <P
>>> len(dataMobil) <P
4
>>>
```

Angka 4 yang dihasilkan `len()` menyatakan jumlah elemen di `dataMobil`.

Penambahan dan Penghapusan Elemen

Elemen di senarai dapat ditambahi ataupun dikurangi. Penambahan elemen dilakukan dengan menggunakan metode `append()`. Elemen yang ditambahkan disisipkan di akhir senarai. Contoh:

```
>>> s = [27, 28, 29, 30]
>>> s.append(5)
>>> print(s)
[27, 28, 29, 30, 5]
```

Penambahan elemen ke senarai juga bisa dilakukan menggunakan metode `insert ()`. Format penggunaannya seperti berikut:

senarai.insert(indeks,nilai)

Argumen pertama menyatakan indeks tempat elemen baru akan disisipkan dan nilai menyatakan nilai elemen yang akan disisipkan.

Contoh:

```
>>> s = [27, 28, 29, 30] <Ji
>>> s.insert(2, 7) <Ji
>>> print(s) <Ji
[27, 28, 7, 29, 30]
>>>
```

Di contoh ini, perintah `s.insert (2, 7)` digunakan untuk menyisipkan bilangan 7 pada posisi indeks 2.

Untuk menghapus elemen dalam senarai, fungsi yang digunakan adalah `del ()`. Contoh:

```
.....
>>> s = [27, 28, 29, 30] <Ji
>>> del (s [2]) <Ji
>>> print(s) <Ji
[27, 28, 30]
>>>
```

Pada contoh ini, `del (s [2])` digunakan untuk menghapus elemen di senarai `s` pada indeks 2 (elemen ketiga). Itulah sebabnya, elemen 29 tidak terlihat saat `print (s)` dieksekusi.

Penghapusan sejumlah elemen dapat dilakukan melalui irisan. Contoh:

```
>>> s = [27, 28, 29, 30] <Ji
>>> del(s[1:3]) <Ji
>>> print(s) <Ji
[27, 30]
>>>
```

Perintah `del (s [1: 3 J)` digunakan untuk menghapus elemen pada `s` pada indeks 1 dan 2. Itulah sebabnya, elemen dengan nilai 28 dan 29 (indeks 1 dan 2) tidak terlihat ketika `print (s)` dieksekusi.

Metode Lain-Lain

Senarai mempunyai sejumlah metode yang berguna untuk melakukan berbagai operasi. Tabel 3.1 dapat dijadikan sebagai referensi awal.

Tabel 9.1 Metode-metode pada objek senarai

Metode	Keterangan
<code>count(x)</code>	Metode ini digunakan untuk menghitung frekuensi elemen <code>x</code> pada senarai. Contoh: <pre>>>> z = ['A', 'B', 'A', 'D', 'A'] <9 >>> z.count('A') <9 3 >>></pre>
<code>extend(d)</code>	Metode ini menambahkan elemen-elemen dalam senarai <code>d</code> ke akhir senarai. Contoh: <pre>>>> x = [1, 2, 3] <9 >>> z = [88, 89] <9 >>> z.extend(x) <9 >> print(z) <9 [88, 89, 1, 2, 3] >>></pre>
<code>index(x)</code>	Metode ini mendapatkan indeks dari nilai <code>x</code> yang pertama kali dijumpai pada senarai. Contoh : <pre>>>> z = ['A', 'B', 'A', 'D', 'A'] <9 >>> z.index('B') <9 1 >>></pre>
<code>pop([i])</code>	Metode ini menghapus elemen yang terletak pada indeks <code>i</code> dan memberikan nilai balik berupa nilai elemen tersebut. Jika tidak ada argumen, elemen yang dihapus adalah elemen yang paling belakang. Contoh: <pre>>>> z = ['A', 'B', 'A', 'D', 'A'] <9 >>> z.pop(2) <9</pre>

Metode	Keterangan
	<pre>'A' >>> print(z) <9 ['A', 'B', 'D', 'A'] >>></pre>
remove(x)	<p>Metode ini menghapus elemen <i>x</i> yang dijumpai pertama kali pada senarai. Contoh:</p> <pre>>>> z = ['A', 'B', 'A', 'D', 'A'] <9 >>> z.remove('A') <9 >>> print(z) <9 ['B', 'A', 'D', 'A'] >>></pre>
reverse()	<p>Metode ini berguna untuk membalik urutan senarai. Contoh:</p> <pre>>>> a = ['A', 'B', 'C', 'D', 'E'] <9 >>> a.reverse() <9 >>> print(a) <9 ['E', 'D', 'C', 'B', 'A'] >>></pre>
sort()	<p>Metode ini bermanfaat untuk mengurutkan elemen-elemen dalam senarai. Contoh :</p> <pre>>>> a = ['A', 'B', 'C', 'D', 'E'] <9 >>> a.sort() <9 >>> print(a) <9 ['A', 'B', 'C', 'D', 'E'] >>></pre>

Fungsi-Fungsi yang Berhubungan dengan Senarai

Sejumlah fungsi berikut mempunyai hubungan dengan senarai: `range`, `filter()`, `map()`, `sorted()`, `max()`, dan `min()`. Penjelasan dan contoh fungsi-fungsi ini dapat dilihat pada Tabel 3.2.

Tabel A.11 Fungsi-fungsi untuk senarai

Fungs	Keterangan
range()	<p>Fungsi ini berguna untuk mendapatkan senarai yang mengandung deretan nilai yangurut seperti:</p> <p>0 (1 2 3 4 5 6 7 8 9 10)</p> <p>fJ (135791113 15)</p> <p>€) (10 9 8 7 6 5 4 3 2 1)</p> <p>Perintah untuk contoh0 :</p> <pre>list(range(1, 11))</pre> <p>Pada contoh ini, angka 11 menyatakan bahwa angka terakhir adalah 11-1.</p> <p>Perintah untuk contoh @ :</p> <pre>list(range(1, 16, 2))</pre> <p>Angka 2 menyatakan kenaikan dari satu bilangan ke bilangan berikutnya.</p> <p>Perintah untuk contoh €) :</p> <pre>list(range(10, 0, -1))</pre> <p>Angka minus menyatakan penurunan dari satu angka ke angka berikutnya.</p>
filter()	<p>Fungsi ini berguna untuk menapis suatu senarai dengan menggunakan suatu fungsi. Bentuk penggunaannya seperti berikut:</p> <pre>list(filter(namaFungsi, senarai))</pre> <p>Contoh berikut digunakan untuk mendapatkan senarai berdasarkan senarai s dengan elemen-elemennya berupa bilangan ganjil (yang diatur melalui fungsi f()) :</p> <pre>>>> def f(x): return x % 2 != 0 ... >>> s = [4, 3, 7, 2, 5, 8, 1, 6] >>> list(filter(f, s)) [3, 7, 5, 1]</pre>
map()	<p>Fungsi ini berguna untuk mendapatkan senarai dari suatu senarai dengan setiap elemen senarai asal dikenai suatu fungsi. Bentuk pemakaiannya seperti berikut :</p> <pre>list(map(namaFungsi, senarai))</pre> <p>Contoh berikut menghasilkan senarai dengan setiap elemen bernilai kuadrat elemen asalnya :</p> <pre>>>> def f(x): return x * x</pre>

Fungs	Keterangan
	<pre>... :9 >>> s = [4' 3, 7' 2, 5, 8' 1, 6] :9 >>> list(map(f, s)) :9 [1 6, 9, 49, 4' 25, 64, 1, 36] >>></pre>
sum()	<p>Fungsi ini menghasilkan penjumlahan nilai semua elemen dari suatu senarai. Contoh :</p> <pre>>>> s = [4' 3, 7' 2, 5, 8' 1, 6] :9 >>> sum(s) :9 36 >>></pre>
max()	<p>Fungsi ini menghasilkan nilai terbesar pada suatu senarai. Contoh:</p> <pre>>>> s = [4' 3, 7' 2, 5, 8' 1, 6] :9 >>> max (s) :9 8</pre>
min()	<p>Fungsi ini menghasilkan nilai terkecil pada suatu senarai. Contoh:</p> <pre>>>> s = [4' 3, 7' 2, 5, 8' 1, 6] :9 >>> min (s) :9 8 >>></pre>

Operator pada Senarai

Sejumlah operator yang dapat dikenakan pada *list* dapat dilihat pada Tabel 3.3.

Tabel 3.3 Operator untuk senarai

Operator	Keterangan	Contoh
+	Operator ini digunakan untuk menggabungkan elemen-elemen kedua senarai.	<pre>>>> [1, 2, 3] + [3' 4' 5] :9 [1, 2, 3, 3, 4' 5] >>></pre>
*	Operator ini digunakan untuk mengulang elemen-elemen	<pre>>>> ['e # ' 's ' 'e v '] * 3 :9 ['e " ' 's ' 'e v ' 'e " ' 's ' 'e v ' 'e " ' 's ' 'e v ']</pre>

Operator	Keterangan	Contoh
	suatu senarai	<pre>1 2 3 4 5 1 2 3 4 5 [1]</pre> <pre>>>></pre>
<code>!=</code>	Operator ini digunakan untuk melakukan perbandingan ketidaksamaan elemen-elemen pada dua senarai. Hasil <code>True</code> jika perbandingan pada semua elemen pada posisi yang sama menghasilkan nilai <code>True</code> .	<pre>>>> x = (1, 2, 3) < 9 >>> y = (2, 1, 3) < 9 >>> x != y < 9 True >>></pre>
	Perbandingan sama dengan. Hasil <code>True</code> jika perbandingan pada semua elemen pada posisi yang sama menghasilkan nilai <code>True</code>	<pre>>>> (1,2) == (2,1) < 9 False >>> (1,2) == (1,2) < 9 True >>></pre>
<code>in</code>	Anggota ("terdapat dalam")	<pre>>>> 'a' in ['f', 'a', 'x'] < 1 True >>> 'A' in ['f', 'a', 'x'] < 9 False >>></pre>

3.5.2 Tupel

Tupel (*tuple*) sebenarnya menyerupai senarai. Perbedaannya seperti berikut:

- 1) tanda kurung siku digunakan pada tupel, sedangkan tanda kurung dipakai pada senarai;
- 2) tupel bersifat *immutable* (bagian isinya tidak dapat diubah), sedangkan senarai bersifat *mutable* (bagian isinya bisa diubah).

Bentuk pembuatan tupel adalah seperti berikut:

variabel = (*elemen1*, *elemen2*, ...)

Tabel 3.4 menunjukkan beberapa contoh pembentukan tupel.

Tabel S.4 Contoh pembentukan tupel

Penugasan	Keterangan
<code>t = ()</code>	Tupel kosong (belum memiliki elemen sama sekali)
<code>t = (1, 2)</code>	Tupel yang mengandung dua elemen
<code>t = 1, 2, 3</code>	Tupel dengan tiga elemen. Tanda kurung boleh tidak disertakan
<code>t = 1, atau t = (1,)</code>	Tupel dengan satu elemen
<code>t = (1, 2, (3, 4))</code>	Tupel dengan salah satu elemen berupa tupel
<code>t = ("Avanza ", 1300)</code>	Tupel dengan elemen berbeda tipe

Cara Mendapatkan Jumlah Elemen Tuple

Seperti halnya pada senarai, `len()` dapat digunakan untuk mendapatkan jumlah elemen pada tupel. Contoh:

```

>>> t = ['A', 1]{}
>>> t = [1, 2, 5, 7] {}
>>> len(t)      {}
4
>>> t = []{}
>>> len(t)      {}
0
>>>

```

Pengaksesan Elemen Tupel

Satu elemen pada tupel dapat diperoleh dengan menyebutkan indeks dengan bentuk seperti berikut:

namaTupel[indeks]

Adapun sejumlah elemen pada tupel dapat dilakukan melalui irisan dengan bentuk seperti berikut:

namaTupel[m:n]

Conteh:

```
.....
>>> t = ['P', 'Y', 't', 'h', 'o', 'n'] (/
>>> t[1] (/
'y'
>>> t[1:3] (/
['y', 't']
>>>
.....
```

Pada contoh ini, `t[1]` digunakan untuk mendapatkan elemen kedua pada `t`. Seperti halnya pada senarai, indeks pada tupel dimulai dari 0. Pada contoh kedua, `t[1:3]` berarti tupel dengan indeks 1 dan 2.

Operator pada Tupel

Operator pada Tabel 3.3 yang diterapkan pada senarai, juga bisa digunakan pada tupel. Conteh:

```
>>> ['y', 'a'] + [1, 2] (/
['y', 'a', 1, 2]
>>> ['y', 'a'] * 3 (/
['y', 'a', 'y', 'a', 'y', 'a']
>>> 'y' in ['y', 'a'] (/
False
>>>
```

Pada contoh pertama, tupel `['y', 'a']` dan `[1, 2]` digabungkan melalui `+`. Perhatikan bahwa operasi dengan elemen berbeda tipe bisa dikenakan pada tupel. Pada contoh kedua, tupel `['y', 'a']` digandakan tiga kali. Pada contoh ketiga, keberadaan 'Y' diperiksa di tupel `['y', 'a']`. Hasil `False` menyatakan bahwa 'Y' tidak terdapat pada `['y', 'a']`.

Konversi dari Senarai ke Tupel dan Sebaliknya

Suatu senarai dapat dikonversi menjadi tupel dengan cara seperti berikut:

`tuple(senarai)`

Contoh:

```
>>> s = [4, 5, 3, 1]
>>> t = tuple(s)
>>> t
(4, 5, 3, 1)
>>> type(s)
<class 'list'>
>>> type(t)
<class 'tuple'>
>>>
```

Pada contoh di atas, `s` adalah senarai dan `t` adalah tupel. Perintah `type(s)` dan `type(t)` memperlihatkan kelas masing-masing.

Untuk mengonversi dari tupel ke senarai, perintah yang digunakan berupa:

`list(tupe/)`

Contoh:

```
>>> t = ('a', 'b', 'c')
>>> s = list(t)
>>> s
['a', 'b', 'c']
>>>
```

Pengurutan pada Tupel

Tupel dapat diurutkan dengan menggunakan fungsi `sorted()`.

Namun, hasilnya selalu berupa senarai. Contoh:

```
>>> t = (5, 3, 4, 1) <Ji
>>> sorted(t) <Ji
[1, 3, 4, 5]
>>> sorted(t, reverse= True) <Ji
[5, 4, 3, 1]
>>>
```

Contoh di atas menunjukkan cara untuk mengurutkan secara urut naik maupun urut turun. Jika argumen `reverse = True` diberikan, hasilnya berupa senarai yang nilai elemen-elemennya diurutkan secara urut turun.

3.5.3 Kamus

Kamus (*dictionary*) berfungsi seperti kamus dalam kehidupan sehari-hari. Di dalam kamus, Pencarian suatu arti didasarkan pada kata yang menjadi kata kunci. Di dalam struktur data kamus, kunci digunakan untuk mendapatkan suatu nilai.

Penyusunan kamus dilakukan dengan bentuk seperti berikut:

variabel = {kunci: nilai, kunci: nilai, ...}

Perhatikan bahwa setiap elemen dalam kamus selalu berupa pasangan kunci dan nilai. Dalam hal ini:

- *kunci* dapat berupa tipe data *immutable* (misalnya string atau bilangan);
- *nilai* dapat berupa tipe apa saja.

Contoh berikut menunjukkan pembentukan kamus dengan nama hewan:

```
>>> hewan = {'kucing': 'cat', 'burung':
'bird', 'sapi': 'cow' }(/)
>>> hewan(/i
{'kucing': 'cat', 'burung': 'bird', 'sapi': 'cow'}
>>>
```

Cara Mendapatkan Nilai Melalui Kunci

Berdasarkan contoh di depan, dimungkinkan untuk mendapatkan kata dalam bahasa Inggris untuk "kucing", "burung", ataupun "sapi". Secara umum, nilai dalam kamus dapat diakses melalui kunci dengan bentuk seperti berikut:

variabel[kunci]

Contoh:

```
>>> hewan = { 'k u c i n g': 'cat', 'b u r u n g':
'bird', 'sapi': 'cow' }(/)
>>> hewan [ ' kucing' ] (/i
'cat'
>>> hewan [ 'sapi' ] (/i
'cow'
>>> hewan [ 'unta' ] (/i
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'unta'
>>>
```

Pada contoh ini, `hewan['kucing']` merupakan cara untuk mendapatkan nilai untuk kunci "kucing". Perlu diketahui, apabila kunci yang dimasukkan tidak terdapat pada kamus, akan muncul `KeyError`.

Cara Mendapatkan Jumlah Kunci

Jumlah kunci yang terdapat pada suatu kamus, dapat diperoleh melalui fungsi `len()`. Contoh:

```
>>> hewan = {'kucing' : 'cat', 'burung'
'bird', 'sapi' : 'cow'}
>>> len(hewan)
3
>>>
```

Hasil di atas menyatakan bahwa kamus `hewan` mempunyai tiga kunci.

Cara Mendapatkan Daftar Kunci

Metode pada kamus yang bernama `keys()` dapat dipakai untuk mendapatkan semua kunci pada kamus dengan hasil berupa senarai yang dinyatakan dengan `dict.keys()`. Contoh:

```
.....
>>> hewan = {'kucing' : 'cat', 'burung'
'bird', 'sapi' : 'cow'}
>>> hewan.keys()
dict_keys(['kucing', 'burung', 'sapi'])
>>>
.....
```

Senarai yang sesungguhnya dapat diperoleh dengan menuliskan:

```
list(hewan.keys())
```

Cara Mendapatkan Daftar Nilai

Untuk mendapatkan semua nilai dalam kamus, metode bernama `values()` bisa digunakan. Hasil berupa senarai yang dinyatakan dengan `dict.values()`. Contoh:

```
>>> hewan = {'kucing' : 'cat', 'burung'
'bird', 'sapi' : 'cow'}
>>> hewan.values()
dict_values(['cat', 'bird', 'cow'])
>>>
```

Senarai yang sesungguhnya dapat diperoleh dengan menuliskan:

```
list(hewan.values())
```

Cara Mendapatkan Pasangan Kunci dan Nilai

Untuk mendapatkan semua pasangan kunci dan nilai yang terdapat pada suatu kamus, metode `items()` bisa dipakai Hasilnya berupa senarai. Contoh:

```
.....  
>>> hewan = {'kucing' : 'cat', 'burung'  
'bird', 'sapi' : 'cow'} (/}  
>>> hewan.items() (/i  
dict_items([('kucing', 'cat'), ('burung',  
'bird'), ('sapi', 'cow')])  
>>>
```

Cara Memeriksa Keberadaan Suatu Kunci

Untuk memeriksa keberadaan suatu kunci, operator `in` bisa dipakai.
Contoh:

```
.....  
>>> hewan = {'kucing' : 'cat', 'burung'  
'bird', 'sapi' : 'cow'} (/}  
>>> 'kucing' in hewan (/i  
True  
>>> 'unta' in hewan (/i  
False  
>>>
```

Hasil `False` (salah) pada contoh di atas menyatakan bahwa kamus `hewan` tidak memiliki kunci bernama 'unta'.

Cara Mendapatkan Nilai Melalui Metode `get()`

Metode `get()` berguna untuk mendapatkan nilai berdasarkan suatu kunci. Contoh:


```
>>> hewan = {'kucing' : 'cat', 'burung'
'bird', 'sapi' : 'cow'} (/{
>>> hewan.get('kucing') (/{
'cat'
>>> hewan.get('unta') (/{
>>>
```

Hasil di atas menunjukkan bahwa metode `get()` tidak menghasilkan apa-apa kalau kunci yang dicari tidak ada pada kamus.

Metode Update()

Nilai yang terdapat pada kamus dapat diubah dengan menggunakan metode `update()`. Contoh:

```
>>> hewan = {'kucing' : 'cat', 'burung'
'bird', 'sapi' : 'cow'} (/{
>>> hewan.get('sapi') (/{
'cow'
>>> hewan.update({'sapi' : 'cattle'}) (/{
>>> hewan.get('sapi') (/{
'cattle'
>>>
```

Pemanggilan `get()` yang pertama menunjukkan bahwa nilai untuk 'sapi' adalah 'cow'. Kemudian, nilai 'cow' ini diubah melalui:

```
hewan.update({'sapi' : 'cattle'})
```

Setelah itu, pemanggilan `get()` menghasilkan 'cattle'.

Metode `update()` juga dapat digunakan untuk menambahkan pasangan kunci dan nilai ke kamus. Contoh:

```

>>> hewan = {'kucing' : 'cat', 'burung' :
'bird', 'sapi' : 'cow'} (/{
>>> hewan (/i
{'kucing': 'cat', 'burung':      'bird', 'sapi':
'cow'}
>>> hewan.update({'merpati' : 'pigeon'}) (/{
>>> hewan (/i
{'kucing': 'cat', 'burung': 'bird', 'sapi':
'cow', 'merpati': 'pigeon'}
>>>

```

Penambahan pasangan kunci dan nilai akan dilakukan secara otomatis begitu kunci tersebut tidak terdapat pada kamus.

Penghapusan Data dengan del()

Pasangan kunci dan nilai dalam suatu kamus dapat dihapus. Hal ini dilakukan dengan menggunakan fungsi del (). Contoh:

```

>>> hewan = {'kucing' : 'cat', 'burung' :
'bird', 'sapi' : 'cow'}
>>> hewan (/i
{'kucing': 'cat', 'burung': 'bird', 'sapi':
'cow'}
>>> del(hewan['burung'])
>>> hewan (/i
{'kucing': 'cat', 'sapi': 'cow'}
>>>

```

Tampak bahwa setelah pernyataan berikut dieksekusi

```
del(hewan['burung'])
```

Pasangan kunci dan nilai yang kuncinya berupa "burung" sudah tidak ada lagi.

Penghapusan Semua Data

Semua pasangan kunci dan nilai yang terdapat pada suatu kamus dapat dihapus dengan menggunakan metode clear (). Contoh:

```
>>> hewan = {'kucing'    'cat'    'burung'
'bird', 'sapi'    'cow•}
>>> hewan.clear()
>>> hewan
{}
>>>
```

Pemrosesan Item Melalui Perulangan

Conteh berikut menunjukkan cara mendapatkan kunci di kamus melalui

for:

```
.....
>>> hewan = {'kucing'    'cat'    'burung'
'bird', 'sapi'    'cow•}
>>> for nilai in hewan:
        print(nilai)
```

```
kucing
burung
sapi
>>>
```

Untuk mendapatkan nilai dari setiap kunci, notasi *namaKamus[kunci]* dapat digunakan. Conteh:

```
>>> hewan = {'kucing'    'cat'    'burung'
'bird', 'sapi'    'cow'}
>>> for nilai in hewan:
        print(nilai,      hewan[nilai])
```

```
kucing    cat
burung    bird
sapi      cow
>>>
```

Operator untuk Kamus

Operator yang biasa digunakan pada operasi kamus meliputi operator

!=, ==, dan in.

3.5.4 Himpunan

Himpunan (set) adalah suatu struktur data yang menyimpan sekumpulan nilai dengan nilai masing-masing tidak ada yang kembar. Selain itu, urutan data pada himpunan tidak penting. Jadi,

```
{1, 2, 3}
```

```
{3, 2, 1}
```

menyatakan himpunan yang sama. Berbeda pula dengan struktur-struktur data senarai dan tupel, elemen pada himpunan tidak dapat diakses karena tidak ada indeks.

Pembentukan Himpunan

Himpunan dibentuk dengan menggunakan {}. Contoh:

```
.....  
>>> warna = {'merah', 'biru', 'hijau'}  
>>> type(warna)  
<class 'set'>  
>>> warna  
{ 'hijau', ' bir u ', ' mera h ' }  
>>>
```

Perintah type() digunakan untuk memperoleh tipe warna. Tampak bahwa warna berkelas set.

Untuk membuat himpunan kosong, perintahnya berupa:

```
anggota = set ()
```

Himpunan bisa didapatkan melalui senarai. Contoh:

```
.....  
>>> senaraiBentuk = ['segitiga',  
'persegipanjang', 'bulat']  
>>> bentuk = set(senaraiBentuk)  
>>> bentuk  
{ 'segitiga', 'persegipanj ang', 'bulat' }  
>>>
```

Perubahan dari senarai ke himpunan dilakukan melalui:

```
bentuk = set(senaraiBentuk)
```

Penambahan Elemen ke Himpunan

Penambahan elemen ke suatu himpunan dilakukan menggunakan metode `add()`. Contoh:

```
>>> bentuk = {'segitiga', 'persegi panjang',  
'bulat'}  
>>> bentuk.add('oval')  
>>> bentuk  
{'segitiga', 'oval', 'persegi panjang', 'bulat'}  
>>>
```

Cara Mengetahui Jumlah Elemen di Himpunan

Jumlah elemen di himpunan diperoleh dengan menggunakan fungsi `len()`. Contoh:

```
>>> bentuk = {'segitiga', 'persegi panjang',  
'bulat'}  
>>> len(bentuk)  
3  
>>>
```

Penghapusan Elemen di Himpunan

Metode `discard()` dapat digunakan untuk menghapus elemen pada himpunan. Contoh:

```
>>> bentuk = {'segitiga', 'persegi panjang',  
'bulat'}  
>>> bentuk.discard('trapesium')  
>>> bentuk  
{'segitiga', 'persegi panjang', 'bulat'}  
>>> bentuk.discard('segitiga')  
>>> bentuk  
{'persegi panjang', 'bulat'}  
>>>
```

Pada contoh ini,

```
bentuk.discard('trapesium')
```

digunakan untuk menghapus "trapesium". Mengingat "trapesium" tidak ada di himpunan bentuk, bentuk tidak berubah. Hal ini menyatakan bahwa menghapus elemen yang tidak tersedia di himpunan tidak memberikan efek apa-apa. Pada contoh kedua,

```
bentuk.discard('segitiga')
```

membuat bentuk "segitiga" dihapus.

Metode bernama `remove()` juga dapat dipakai untuk menghapus elemen di himpunan. Berbeda dengan `discard()`, sekiranya elemen yang hendak dihapus ternyata tidak terdapat di himpunan, eksepsi akan ditimbulkan. Contoh:

```
>>> bentuk = {'segitiga', 'persegi panjang',  
              'bulat'}  
>>> bentuk.remove('segitiga')  
>>> bentuk  
{'persegi panjang', 'bulat'}  
>>> bentuk.remove('oval')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
KeyError: 'oval'  
>>>
```

Tampak bahwa

```
bentuk.remove('oval')
```

menimbulkan eksepsi `KeyError` mengingat bentuk "oval" tidak terdapat pada himpunan bentuk.

Untuk menghapus seluruh elemen di himpunan, metode `clear()` bisa digunakan. Contoh:

```
>>> bentuk = {'segitiga', 'persegi panjang',
'bulat'}<P
>>> bentuk.clear()<P
>>> bentuk<P
set()
>>>
```

Hasil berupa `set()` menyatakan bahwa `bentuk` tidak berisi elemen sama sekali.

Subhimpunan

Subhimpunan adalah suatu himpunan yang keseluruhan anggotanya terdapat pada himpunan lain. Sebagai contoh, terdapat dua himpunan seperti berikut:

- `paketA = { "apel", "anggur" }`
- `paketB = { "anggur", "kelengkeng", "anggur", "melon" }`

Maka, `paketA` adalah subhimpunan dari `paketB` karena kedua anggotanya terdapat pada `paketB`.

Suatu himpunan merupakan subhimpunan atau tidak bisa diperiksa dengan menggunakan metode bernama `is subset()`. Metode ini mengembalikan nilai `True` sekiranya suatu himpunan adalah subhimpunan. Contoh:

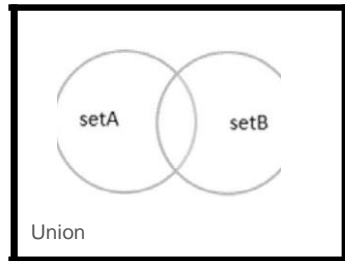
```
>>> paketA.issubset(paketB)<P
True
>>> paketB.issubset(paketA)<P
False
>>>
```

Hasil ini menyatakan bahwa `paketA` adalah subhimpunan `paketB`, tetapi tidak untuk sebaliknya.

Operasi Himpunan

Union, interseksi, dan selisih adalah operasi yang biasa dikenakan pada himpunan. Union adalah penggabungan elemen-elemen dari dua himpunan. Hasilnya, elemen yang kembar hanya akan disertakan satu kali sehingga elemen-elemen dalam himpunan tidak ada yang kembar.

Gambar 3.6 memberikan ilustrasi gabungan antara setA dan setB.



Gambar S.6 Union menggabungkan himpunan setA dan setB

Operasi union dapat dilakukan dengan menggunakan operator ataupun melalui metode `union()`. Contoh:

```
>>> setA = {'hijau', 'biru', 'ungu', 'merah'}
>>> setB = {'merah', 'kuning', 'putih', 'hijau',
            'hitam'}
>>> setC = setA | setB
>>> setC
{'kuning', 'putih', 'hitam', 'biru', 'hijau',
 'ungu', 'merah'}
>>> setC = setA.union(setB)
>>> setC
{'kuning', 'putih', 'hitam', 'biru', 'hijau',
 'ungu', 'merah'}
>>> setC = setB.union(setA)
>>> setC
{'kuning', 'putih', 'hitam', 'biru', 'hijau',
 'ungu', 'merah'}
>>>
```

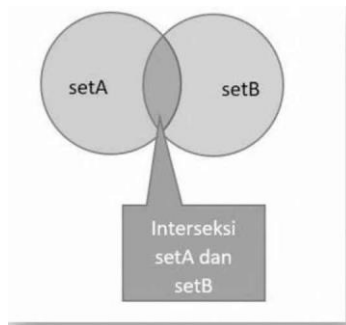

Tampak bahwa hasil `setA & setB`, `setA.union(setB)`, dan `setB.union (setA)` adalah gabungan elemen-elemen di `setA` dan `setB`.

Operasi interseksi digunakan untuk mendapatkan hanya elemen-elemen yang muncul di kedua himpunan. Hal ini dapat diperoleh dengan menggunakan metode `intersection ()` ataupun operator `&`.

Contoh:

```
>>> setA = { 'hijau', 'biru', 'ungu', 'merah' }
>>> setB = { 'merah', 'kuning', 'putih', 'hijau',
'hitam' }
>>> setC = setA & setB
>>> setC
{ 'h i j a u', 'mer a h' }
>>> setC = setB.intersection(setA)
>>> setC
{ 'h i j a u', 'mer a h' }
>>> setC = setA.intersection(setB)
>>> setC
{ 'h i j a u', 'mer a h' }
>>>
```

Tampak bahwa `setA & setB`, `setA.intersection (setB)`, dan `setB. intersection (setA)` memberikan hasil yang sama. Gambar 3.7 mempertegas hasil operasi interseksi.



Gambar :J.7 Interseksi himpunan setA dan setB

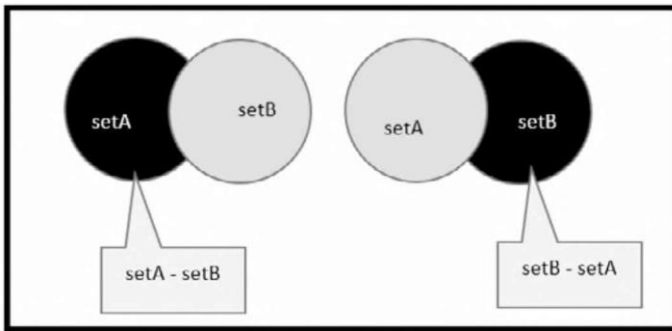
Operasi selisih digunakan untuk mendapatkan elemen-elemen yang terdapat di suatu himpunan, tetapi tidak berada di *set* lain. Hal ini dapat diperoleh dengan menggunakan metode `difference()`. Contoh:

```
>>> setA = {'hijau', 'biru', 'ungu', 'merah'}
>>> setB = {'merah', 'kuning', 'putih', 'hijau',
            'hitam'}
>>> setC = setA - setB
>>> setC
{'biru', 'ungu'}
>>> setC = setA.difference(setB)
>>> setC
{'hijau', 'merah'}
>>> setC = setB - setA
>>> setC
{'hitam', 'kuning', 'putih'}
>>> setC = setB.difference(setA)
>>> setC
{'hitam', 'kuning', 'putih'}
>>>
```

Hasil di atas menunjukkan bahwa `setA.difference(setB)` dan `setB.difference(setA)` memberikan hasil yang berbeda. Perbedaannya adalah seperti berikut.

- `setA.difference(setB)` berarti semua elemen di `setA` yang tidak ada di `setB`.
- `setB.difference(setA)` berarti semua elemen di `setB` yang tidak ada di `setA`.

Gambar 3.8 mempertegas operasi selisih pada dua himpunan.



Gambar 3.8 Selisih himpunan setA dan setB

Operator " \oplus " ("eksklusif atau") digunakan melaksanakan operasi "atau eksklusif" (xor). Hasil operasi ini adalah gabungan elemen di kedua *set* dikurangi dengan elemen-elemen yang sama pada kedua himpunan. Gambar 3.10 mempertegas hasil operasi ini.



Gambar 3.10 Operasi eksklusif atau pada himpunan setA dan setB

Operator " \oplus " dapat pula digantikan dengan metode bernama `symmetrical_difference()`. Contoh:

```
>>> setA = {'hijau', 'biru', 'ungu', 'merah'}
>>> setB = {'merah', 'kuning', 'putih', 'hijau',
'hitam'}
>>> setC = setA ^ setB
>>> setC
{'hitam', 'putih', 'kuning', 'ungu', 'biru'}
>>> setC = setA.symmetric_difference(setB)
>>> setC
-
{'hitam', 'putih', 'kuning', 'ungu', 'biru'}
>>> setC = setB.symmetric_difference(setA)
>>> setC
-
{'hitam', 'putih', 'kuning', 'ungu', 'biru'}
>>>
/,.....'.....'.....'.....'.....'.....'
```

Tampak bahwa dua perintah berikut memberikan hasil yang sama:

```
setA.symmetric_difference(setB)
setB.symmetric_difference(setA)
```

Kedua perintah tersebut bisa ditulis dengan salah satu perintah berikut:

```
setA ^ setB
setB ^ setA
```

Penentuan Keanggotaan Suatu Elemen dalam Himpunan

Operator `in` dapat dipakai untuk memeriksa suatu nilai merupakan anggota suatu himpunan atau tidak. Contoh:

```
.....
>>> vokal = {'a', 'e', 'i', 'o', 'u'}
>>> 'e' in vokal
True
>>> 'x' in vokal
False
>>>
```

Pada contoh ini, 'a' adalah anggota himpunan `vokal`, tetapi 'x' tidak.

Pembandingan dengan `==` dan `!=`

Operator perbandingan seperti ==, !=, dan < dapat diterapkan pada himpunan. Sebagai contoh, operator == dapat digunakan untuk mengetahui dua himpunan mempunyai elemen-elemen yang sama atau tidak. Contoh:

```
>>> vokal1 = { 'a' , 'e' , 'i' , 'o' , 'u' }
>>> setx = { 'o' , 'u' , 'i' , 'e' , 'a' }
>>> setY = { 'A' , 'E' , 'I' , 'U' , 'O' }
>>> vokal1 - setx
True
>>> vokal1 == setY
False
>>>
```

Hasil di atas menyatakan bahwa vokal dan setX memiliki elemen-elemen yang sama. Adapun vokal dan setY memiliki elemen-elemen yang sama.

Pada perbandingan dengan <, <=, >, atau >=, dasar yang digunakan adalah subhimpunan. Sebagai contoh, a < b bernilai True sekiranya a adalah subhimpunan dari himpunan b. Adapun a >= b bernilai True jika b berkedudukan sebagai subhimpunan dari himpunan a atau a dan b mempunyai elemen-elemen yang sama. Contoh:

```
.....
>>> vokal1 = { 'a' , 'e' , 'i' , 'o' , 'u' }
>>> setM = { 'a', 'i', 'u' }
>>> setM > vokal1
False
>>> setM <= vokal1
True
>>>
```

Pada contoh ini, setM > vokal bernilai False mengingat setM justru merupakan subhimpunan dari himpunan vokal.

3.6 Argumen Baris Perintah

Terkadang, kita menghendaki untuk melewati sejumlah argumen ketika mau menjalankan skrip Python. Sebagai contoh, dikehendaki untuk memasukkan perintah seperti berikut:

```
python namaskrip.py satu dua tiga
```

Pada contoh ini, "satu", "dua", dan "tiga" adalah argumen-argumen baris perintah.

Nah, persoalan sekarang adalah "Bagaimana mendapatkan argumen-argumen tersebut di skrip?" Jawabannya sederhana. Argumen-argumen tersebut dengan sendirinya diletakkan pada senarai `sys.argv`. Untuk keperluan ini, modul `sys` perlu dilibatkan.

Berikut adalah contoh untuk membaca argumen-argumen baris

```
m:
Berkas : argbaris.py
```

```
# Contoh pembacaan argumen baris perintah
```

```
import sys
```

```
print('Argumen:', sys.argv)
```

```
Akhir berkas
```

Contoh hasil pengujian skrip ini adalah seperti berikut:

```
.....
C:\LatOpenCV>python argbaris.py
Argumen: ['argbaris.py']

C:\LatOpenCV>python argbaris.py satu dua tiga
Argumen: ['argbaris.py', 'satu', 'dua', 'tiga']

C:\LatOpenCV>
```

Berdasarkan contoh di atas terlihat bahwa hasilnya adalah senarai dengan elemen pertama berisi nama skrip. Elemen kedua dan seterusnya berisi argumen-argumen baris perintah.

Oleh karena itu, kita bisa menuliskan skrip berikut untuk memperoleh argumen-argumen baris perintahnya secara individual:



Berkas : argbaris2.py

```
#Contoh pembacaan argumen baris perintah
#    Versi 2

import sys

if len(sys.argv) == 1:
    print('Tidak ada argumen baris perintah')
else:
    jumArgumen = len(sys.argv) - 1
    print('Argumen:')
    for indeks in range(1, jumArgumen):
        print(indeks, '. ', sys.argv[indeks], sep = '')
```

Akhir berkas

Pada skrip ini, pesan "Tidak ada argumen baris perintah" ditampilkan sekiranya jumlah elemen pada `sys. argv` hanya satu. Untuk keadaan lainnya, setiap argumen baris perintah ditampilkan pada baris sendiri. Dalam hal ini, `range(1, jumArgumen)` digunakan untuk mendapatkan senarai dengan elemen berupa 1, 2, 3 dan seterusnya hingga `jumArgumen - 1`.

Contoh hasil pengujian skrip ini adalah seperti berikut:

```
C:\LatOpenCV>python argbaris2.py  
Tidak ada argumen baris perintah
```

```
C:\LatOpenCV>python argbaris2.py mawar melati  
anggrek  
Argumen:  
1. mawar  
2. melati  
3. anggrek
```

```
C:\LatOpenCV>
```

Contoh berikut memperlihatkan skrip yang digunakan untuk menampilkan gambar dengan menyebutkan berkas gambar sebagai



n pertama dalam baris perintah:

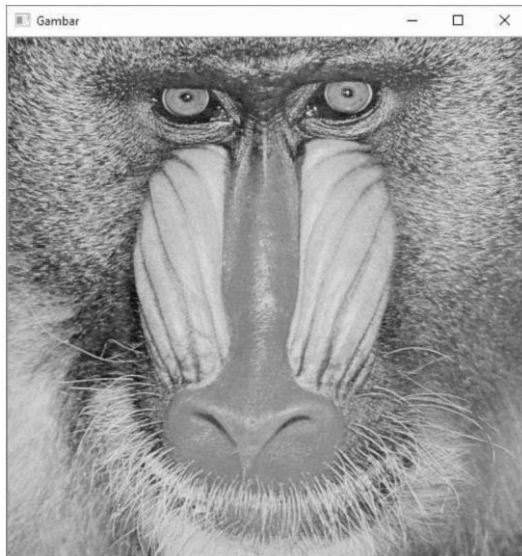
Berkas : gambar.py

```
import cv2  
import sys  
  
if len(sys.argv) == 1:  
    print('Masukkan nama berkas gambar')  
else:  
    berkas = sys.argv[1]  
    citra = cv2.imread(berkas)  
    if citra is None:  
        print('Tidak dapat membaca berkas', berkas)  
    else:  
        cv2.imshow('Gambar', citra)  
        cv2.waitKey(0)
```

Akhir berkas

Skrip ini melibatkan modul `cv2` untuk menangani gambar dan `sys` untuk menangani argumen baris perintah. Pada skrip ini, nama berkas yang diproses oleh `imread()` diperoleh melalui `sys.argv[1]`. Kondisi `citra is None` pada `if`, digunakan untuk memeriksa kalau `citra` tidak berisi data gambar setelah `imread()` dieksekusi.

Gambar 3.11 menunjukkan keadaan setelah jendela berisi gambar baboon.png ditampilkan.



Gambar S.11 Hasil pemanggilan python gambar.py baboon.png

Contoh berikut menunjukkan kalau berkas gambar tidak bisa diproses oleh `imshow()` :

```
-----  
C:\LatOpenCV>python gambar.py singa.png  
Tidak dapat membaca berkas singa.png
```

```
, C:\LatOpenCV>
```

3.7 Penanganan Eksepsi

Secara bawaan, Python akan menghentikan eksekusi ketika menjumpai kesalahan. Sebagai contoh, perhatikan berikut ini:

```

.....
>>> takada; print('Tes•)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'takada' is not defined
>>>

```

Pada contoh ini, perintah yang diberikan mengandung dua pernyataan yang ditulis dalam satu baris dengan pemisah berupa titik-koma. Pada saat `takada` dieksekusi, objek dengan nama seperti ini tidak ada. Maka, Python melontarkan eksepsi sehingga muncul pesan seperti di depan. Pada keadaan seperti ini, eksekusi segera dihentikan dan pernyataan berikutnya yaitu `print('Tes')` tidak dijalankan.

3.7.1 Pengertian Eksepsi

Eksepsi berarti keadaan yang tidak normal. Hal ini bisa terjadi kapan saja ketika suatu perintah dieksekusi oleh interpreter Python. Penyebabnya bisa bervariasi, seperti:

- objek yang dilibatkan dalam suatu perintah tidak ada;
- terjadi pembagian dengan bilangan nol;
- konversi ke bilangan bulat tidak dapat dilakukan karena string yang dikonversi tidak menyatakan bilangan.

Pada saat eksepsi terjadi, Python menampilkan nama eksepsi beserta pesan penjelasannya. `NameError` pada contoh di depan merupakan nama eksepsi. Adapun berikut adalah penjelasan eksepsi yang terjadi:

```

name 'takada' is not defined

```

Dalam praktik, eksepsi yang menghentikan eksekusi secara bawaan terkadang tidak dikehendaki. Oleh karena itu, diperlukan penanganan eksepsi yang dapat digunakan untuk mengontrol eksekusi perintah.

3.7.2 Jenis Eksepsi

Daftar eksepsi pada Python dapat diketahui dengan menuliskan perintah `dir(builtins)`. Contoh:

```
>>> dir(builtins)
['ArithmeticError', 'AssertionError',
'AttributeError', 'BaseException',
'BlockingIOError', 'BrokenPipeError',
'BufferError', 'BytesWarning',
'ChildProcessError', 'ConnectionAbortedError',
'ConnectionError', 'ConnectionRefusedError',

>>>
```

Perintah ini tidak hanya memuat nama-nama eksepsi, tetapi juga objek-objek lain. Nama-nama eksepsi umumnya ditandai dengan kata `Error`, misalnya `IOError` dan `KeyError`.

Tabel 3.5 memperlihatkan sejumlah nama-nama eksepsi.

Tabel 3.5 Nama-nama eksepsi

Eksepsi	Keterangan
AttributeError	Hal ini terjadi sekiranya terdapat penyebutan atribut yang sebenarnya tidak dimiliki oleh objek yang disebut. Contoh : <pre>>>> st ="abc" >>> st.length<:9 Traceback (most recent call last): File "<stdin>", line 1, in <module> AttributeError: 'str' object has no attribute 'length' >>></pre>
ModuleNotFoundError	Eksepsi ini timbul karena modul yang diimport tidak ada . Contoh: <pre>>>> import citra Traceback (most recent call</pre>

Eksepsi	Keterangan
	<pre>last): File "<stdin>", line 1, in <module> ModuleNotFoundError: No module named 'citra' >>></pre>
IndexError	<p>Eksepsi terjadi kalau indeks di luar jangkauan yang tersedia disebut. Contoh:</p> <pre>>>> st = " a b c " <9 >> st[6] <9 Traceback (most recent call last): File "<stdin>", line 1, in <module> IndexError : string index out of range >>></pre>
IOError	<p>Eksepsi ini terjadi kalau terdapat operasi masukan atau keluaran yang mengalami kegagalan. Contoh:</p> <pre>>>> open ("berkasku") <9 Traceback (most recent call last): File "<stdin>", line 1, in <module> FileNotFoundError: [Errno 2] No such file or directory: 'berkasku' >>></pre>
Keyboard Interrupt	<p>Eksepsi ini muncul kalau terdapat interupsi dari pemakai yang menekan tombol Ctrl+C.</p>
KeyError	<p>Eksepsi ini muncul kalau kunci yang digunakan tidak tersedia pada kamus. Contoh :</p> <pre>>>> mobil = {'jazz' : 'honda', 'avanza' : 'toyota' } <9 >>> mobil ['baleno'] <' Traceback (most recent call last): File "<stdin>", line 1, in <module> KeyError: 'baleno' >>></pre>

Ekseps	Keterangan
NameError	<p>Eksepsi ini muncul kalau terdapat penyebutan nama objek yang tidak dikenal. Contoh :</p> <pre>>>> print(takada) Traceback (most recent call last): File "<stdin>", line 1, in <module> NameError: name 'takada' is not defined >>></pre>
SyntaxError	<p>Eksepsi ini muncul kalau terdapat kesalahan secara sintaksis. Contoh print() ditulis dengan print[.]</p>
TypeError	<p>Eksepsi ini terjadi kalau terjadi suatu operasi terhadap objek yang sebenarnya tidak didukung. Contoh pengubah nilai elemen pada himpunan yang sebenarnya bersifat <i>immutable</i> atau tidak dapat diubah dilakukan. Contoh :</p> <pre>>>> warna = {' merah', 'kuning', 'hijau'} >>> warna[1] = 'or anye' Traceback (most recent call last): File "<stdin>", line 1,in <module> TypeError: 'set' object does not support item assignment >>></pre>
ValueError	<p>Eksepsi terjadi karena pemberian nilai argumen yang salah . Contoh :</p> <pre>>>> import math >>> math.sqrt('5') Traceback (most recent call last): File "<stdin>", line 1, in <module> TypeError: must be real number, not str >>></pre>
ZeroDivisionError	<p>Eksepsi ini terjadi karena terdapat pembagian</p>

Eksepsi	Keterangan
	<p>bilangan dengan nol. Contoh:</p> <pre>>>> x= Q<,P >>> print(S / x)<? Traceback (most recent call last): File "<stdin>", line 1, in <module> ZeroDivisionError: division by zero >>></pre>

3.7.3 Penanganan Eksepsi

Agar suatu eksepsi dapat dikontrol ketika terlontar, perlu penanganan eksepsi dengan menggunakan pernyataan `try .. catch`. Bentuk pernyataan ini adalah seperti berikut:

```
try:
    pernyataan_al

    pernyataan_an
except [nama_eksepsi]:
    pernyataan_bl

    pernyataan_bn
[else:
    pernyataan_cl

    pernyataan_cn
```

Bagian `[]` menyatakan opsional. Sekiranya terjadi eksepsi saat `pernyataan_al` hingga `pernyataan_an` dieksekusi, eksekusi akan dilanjutkan ke `except`. Jika eksepsi yang terjadi sama dengan `nama_eksepsi`, `pernyataan_bl` hingga `pernyataan_bn` dijalankan.

Adapun bagian `else` (*pernyataan_c1* hingga *pernyataan_c2*) dijalankan hanya kalau *pernyataan_a1* hingga *pernyataan_an* tidak mengalami eksepsi.

Skrip berikut memberikan gambaran tentang penanganan kesalahan

m masukan data yang dilakukan melalui `input()` :

Berkas : masukan.py

```
while True:
    try:
        masukan = input('Masukkan bilangan: ')
        bilangan = int(masukan)
        break
    except ValueError:
        print('Salah dalam memasukkan bilangan!')
```

```
    t('Bilangan =', bilangan)
    0.;
```

Akhir berkas

Pada skrip ini, eksepsi yang ditangani adalah `ValueError`. Dengan begitu, bagian `except` hanya akan dijalankan kalau eksepsi `ValueError` terjadi pada bagian `try`.

Berikut adalah contoh hasil pengujian skrip `masukan.py`:

```
C:\LatOpenCV>python masukan.py
Masukkan bilangan: dua puluh tiga
Salah dalam memasukkan bilangan!
Masukkan bilangan: 20tiga
Salah dalam memasukkan bilangan!
Masukkan bilangan: 23
Bilangan = 23
```

```
C:\LatOpenCV>
```

Pada saat dua puluh tiga dimasukkan, pernyataan `input()` tidak mengalami eksepsi. Namun, eksepsi timbul ketika pernyataan berikut dijalankan:

```
bilangan = int(masukan)
```

Eksepsi yang terjadi adalah `ValueError` mengingat `int()` tidak dapat mengonversi "dua puluh tiga" menjadi bilangan bulat. Eksepsi ini cocok dengan `ValueError` yang disebutkan dalam `except`. Itulah sebabnya, pernyataan berikut dijalankan:

```
print('Salah dalam memasukkan bilangan!')
```

Karena kondisi `while` selalu bernilai `True`, pernyataan yang mengandung `input()` dijalankan kembali.

Hal serupa yang dijelaskan di depan terjadi ketika 20 tiga dimasukkan. Namun, ketika 23 dimasukkan, `int()` tidak mengalami eksepsi. Akibatnya, pernyataan `break` dieksekusi. Sebagaimana diketahui, `break` di dalam `while` membuat eksekusi terhadap `while` berakhir. Dengan demikian, pernyataan selanjutnya, yaitu

```
print('Bilangan =', bilangan)
```

dijalankan.

3.7.4 Penggunaan `pass` pada `except`

Adakalanya, pernyataan `pass` perlu digunakan pada bagian `except`. Hal ini diperlukan kalau tidak dikehendaki untuk melakukan tindakan apa-apa sewaktu suatu eksepsi terjadi. Skrip berikut menunjukkan penggunaannya:

Contoh penggunaan pass

```
infoMobil = ["Avanza", 2018, "Biru", 1300, 5]
jumlah = 0
```

```
for nilai in infoMobil:
    try:
        bilangan = int(nilai)
        jumlah = jumlah + 1
    except ValueError:
        pass
```

```
print("Jumlah elemen berupa bilangan:", jumlah)
```

Akhir berkas

Hasil eksekusi skrip ini adalah seperti berikut:

```
C:\LatOpenCV>python jumbil.py
Jumlah elemen berupa bilangan: 3
```

```
C:\LatOpenCV>
```

Mekanisme penghitungan jumlah elemen pada tuple infoMobil dilakukan seperti berikut. Pernyataan try .. except digunakan untuk menangani eksepsi pada int(). Sekiranya tidak ada eksepsi, pernyataan jumlah = jumlah + 1 akan dijalankan. Hal ini akan membuat nilai jumlah naik sebesar 1. Sekiranya terjadi eksepsi, misalnya ketika mengonversi int ('Avanza'), pernyataan pass pada bagian except dijalankan, yang sebenarnya tidak menjalankan apa-apa. Hal tersebut akan diulang untuk semua elemen di tuple info, yang diatur melalui:

```
for nilai in infoMobil:
```

3.7.5 Pernyataan try..finally

Pernyataan try ...finally digunakan suatu perintah selalu dieksekusi dan tidak tergantung oleh suatu eksepsi terjadi atau tidak. Bentuk pemakaiannya adalah seperti berikut:

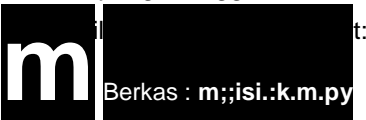
```
try:
    pernyataan_a1

    pernyataan_a1
finally:
    pernyataan_b1

    pernyataan_bn
```

Pada contoh ini, sekiranya eksepsi terjadi pada *pernyataan_a1* hingga *pernyataan_a2*, eksekusi tidak dihentikan. Adapun *pernyataan_b1* hingga *pernyataan_bn* selalu dieksekusi, tidak tergantung ada tidaknya eksepsi.

Contoh yang menggambarkan penggunaan pernyataan try ...finally



```
# Pembacaan data berkas teks
import sys

if len(sys.argv) == 1:
    print('Masukkan nama berkas yang ingin dibaca')
else:
    berkas = sys.argv[1]

    try:
        pegangan = open(berkas, 'r')
        konten = pegangan.read()
        print('Isi berkas:')
        print(konten)
    finally:
```

```
print('Operasi pembacaan berkas telah berakhir')
```

```
print('****')
```

```
Akhir berkas
```

Bagian `try` digunakan untuk membuka berkas yang dinyatakan dalam berkas dan menampilkan isinya. Pembukaan berkas ditangani oleh:

```
pegangan = open(berkas, 'r')
```

Objek `pegangan` berhubungan dengan berkas yang dibuka. Argumen `'r'` menyatakan bahwa berkas digunakan hanya untuk operasi pembacaan data. Isi seluruh berkas yang terbuka dibaca melalui:

```
konten = pegangan.read()
```

Dengan demikian, `konten` berisi seluruh data pada berkas yang terkait dengan objek `pegangan`.

Sekiranya pada bagian `catch` menimbulkan eksepsi, eksekusi akan langsung dilanjutkan ke bagian `finally`. Sekiranya sampai perintah terakhir pada `try` tidak ada yang menimbulkan eksepsi, bagian `finally` akan dijalankan. Dengan demikian, apapun yang terjadi bagian `finally` akhirnya dijalankan.

Contoh berikut memperlihatkan perintah untuk menampilkan isi berkas `argbaris.py`:

```

.....
C:\LatOpenCV>python bacadata.py argbaris.py
Isi berkas:
#Contoh pembacaan argumen baris perintah

import sys

print('Argumen:', sys.argv)

Operasi pembacaan berkas telah berakhir
****

C:\LatOpenCV>

```

Adapun contoh berikut memperlihatkan keadaan ketika pembacaan berkas tidak dapat dilakukan:

```

C:\LatOpenCV>python bacadata.py takada.txt
Operasi pembacaan berkas telah berakhir
Traceback (most recent call last):
  File "bacadata.txt", line 10, in <module>
    pegangan = open(berkas, 'r')
FileNotFoundError: [Errno 2] No such file or
directory: 'takada.txt'

C:\LatOpenCV>

```

Perintah ini digunakan untuk membaca isi berkas `takada.txt` yang memang tidak terdapat pada folder kerja. Perhatikan bahwa walau operasi pembacaan tidak berhasil dilakukan, perintah pada `finally` tetap dieksekusi. Namun, setelah bagian `finally` dijalankan, eksepsi tetap terlontar. Dalam hal ini, eksepsinya berupa `FileNotFoundError`. Itulah sebabnya, perintah berikut yang terletak di luar `try..finally` tidak dieksekusi:

```

print ( '****')

```

Untuk menangani eksepsi di atas, `except` dapat ditambahkan. Conteh:

```
# Pembacaan data berkas teks
# Versi 2

import sys

if len(sys.argv) == 1:
    print('Masukkan nama berkas yang ingin dibaca')
else:
    berkas = sys.argv[1]

    try:
        pegangan = open(berkas, 'r')
        konten = pegangan.read()
        print('Isi berkas:')
        print(konten)
    except:
        pass
    finally:
        print('Operasi pembacaan berkas telah berakhir')

print('****')
```

Akhir berkas

Tambahan

```
except:
    pass
```

digunakan untuk meniadakan eksepsi setelah `finally` dieksekusi.

Berikut adalah hasil pengujian ketika berkas tidak dapat dibuka:

```
C:\LatOpenCV>python bacadata2.py takada.py
Operasi pembacaan berkas telah berakhir
****
```

```
C:\LatOpenCV>
```

Tampak bahwa eksepsi yang terlontar pada contoh sebelum ini tidak terlihat lagi.

3.8 Pengenalan numpy

Numpy (Numerical Python) adalah modul dasar yang bermanfaat untuk melakukan komputasi dengan Python. Paket ini menyediakan larik (*array*) multidimensi yang sangat bermanfaat untuk kepentingan pemrosesan matriks maupun citra. Oleh karena itu, dasar mengenai paket ini perlu dibahas karena sering digunakan dalam pengolahan citra.

3.8.1 Penyertaan Modul numpy

Modul numpy dilibatkan dengan memberikan perintah seperti berikut sekali:

```
import numpy as np
```

Penggunaan `as np` dimaksudkan agar modul numpy dapat diakses melalui kata `np`.

```
.....  
:>>> import numpy as np  
:.....
```

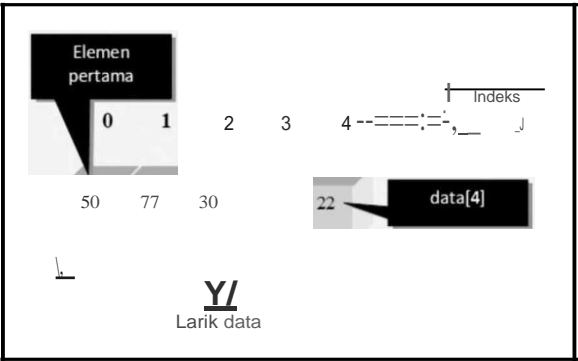
Conten berikut menunjukkan pemakaian `np` untuk mendapatkan versi numpy:

```
.....  
: >>> np.version.version :  
: '1.15.3'  
: >>> .....
```

3.8.2 Pengenalan Larik

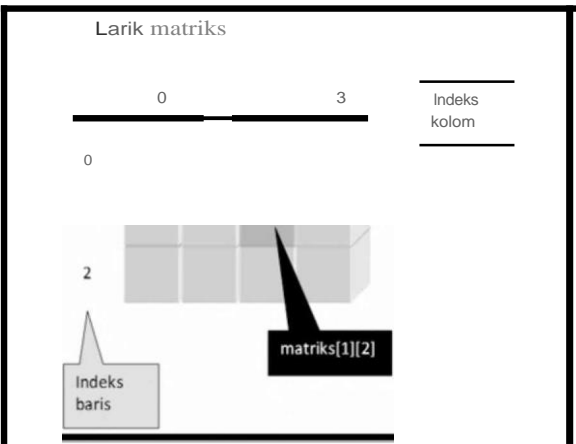
Larik adalah suatu wadah yang memungkinkan sejumlah nilai bertipe sama disimpan. Setiap elemen dalam larik dapat diakses dengan menggunakan indeks yang berupa bilangan bulat positif. Larik yang paling sederhana berdimensi satu. Dalam hal ini, satu indeks digunakan untuk mengakses setiap elemen. Gambar 3.12 menunjukkan contoh

larik berdimensi satu. Perlu diketahui, larik pada numpy hanya ditujukan untuk bilangan.



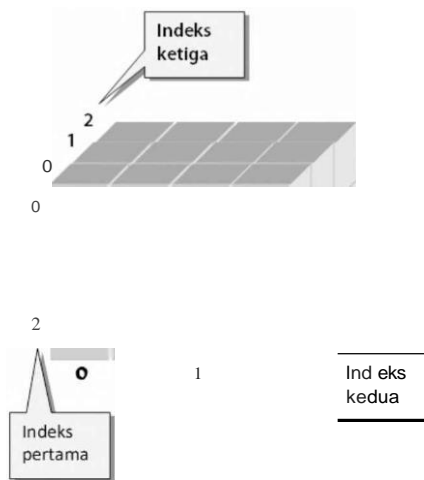
GambarS.lf/ Larik berdimensi dua

Larik berdimensi dua biasa digunakan pada matriks. Larik inilah yang juga biasa dipakai pada pengolahan citra berskala keabu-abuan. Gambar 3.13 menunjukkan contoh larik berdimensi dua. Pada larik ini, setiap elemen diakses melalui dua indeks. Indeks pertama digunakan untuk menyebutkan posisi baris dan indeks kedua untuk menyatakan posisi kolom.



Gambar S.JS Larik berdimensi dua

Larik berdimensi tiga melibatkan tiga indeks untuk menyatakan setiap kolom. Gambar 3.14 menunjukkan contoh larik berdimensi tiga. Larik seperti ini biasa digunakan pada citra berwarna, yang mengandung komponen B, G, dan R.



Gambar 3.14 Larik berdimensi tiga

3.8.3 Larik Berdimensi Satu

Larik berdimensi satu dapat diciptakan dan sekaligus diberi nilai awal melalui senarai. Contoh berikut digunakan untuk membentuk larik data seperti yang diperlihatkan pada Gambar 3.12 dan kemudian menampilkan isi larik beserta tipe data larik:

```
>>> data = np.array([50, 77, 30, 1, 22])
>>> print(data)
[50 77 30  1 22]
>>> type(data)
<class 'numpy.ndarray'>
>>>
```


Adapun berikut menunjukkan contoh untuk mengakses elemen larik:

```
>>> print(data[2])
30
>>> data[2] = 99
>>> print(data[2])
99
>>>
```

Pada contoh ini, data [2] menyatakan elemen dengan indeks 2 (elemen ketiga dari kiri mengingat indeks dimulai dari 0). Contoh di atas sekaligus menunjukkan cara mengambil nilai elemen larik dan mengubah nilai elemen larik.

Fungsi len () milik larik dapat digunakan untuk mengetahui jumlah elemen. Contoh:

```
: .....:
: >>> len(data)
: 5
: >>>
: .....:
```

Hasil di atas menyatakan bahwa jumlah elemen pada larik data sama dengan 5.

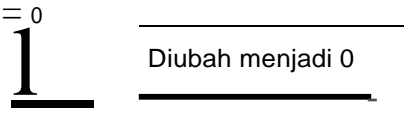
3.8.4 Larik Berdimensi Dua

Contoh berikut menunjukkan pembentukan larik berdimensi dua yang diberi nilai awal dan kemudian diikuti dengan cara untuk menampilkan isi larik dan tipe data larik:

```
>>> matriks = np.array([[1, 2, 3], [4, 5, 6],
[7, 8, 9]])
>>> print(matriks)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
>>> type(matriks)
<class 'numpy.ndarray'>
>>>
```

Adapun berikut menunjukkan contoh untuk mengakses elemen larik berdimensi dua:

```
.....
>>> matriks[2, 1]
8
>>> matriks[2, 1] = 0
>>> matriks[2, 1]
0
>>> matriks[2][1]
0
>>>
```



Pada matriks [2, 1], 2 menyatakan indeks baris dan 1 menyatakan indeks kolom. Notasi seperti matriks [2, 1] boleh juga ditulis mejadi matriks [2][1].

Metode shape () dapat dipakai untuk mendapatkan jumlah baris dan kolom pada larik berdimensi dua. Conteh:

```
.....
>>> matriks.shape
: (3, 3)
:
>>>
:.....
```

Pada contoh ini, angka pertama menyatakan jumlah baris dan angka kedua menyatakan jumlah kolom. Conteh berikut menunjukkan cara mendapatkan jumlah baris dan jumlah kolom secara individual:

```

>>> jumBaris = matriks.shape[0]
>>> jumKolom = matriks.shape[1]
>>> print(jumBaris)
3
>>> print(jumKolom)
3
>>>

```

3.8.5 Larik Berdimensi Tiga

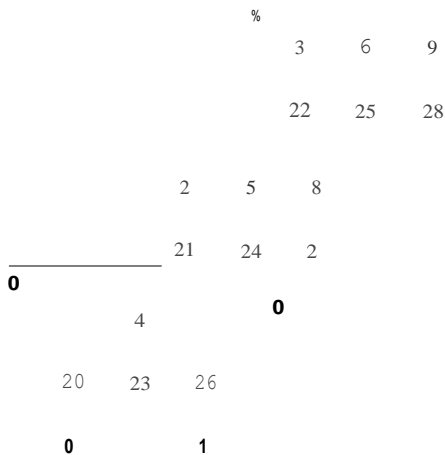
Conteh berikut menunjukkan pembentukan larik berdimensi tiga yang diberi nilai:

```

.....
>>> tigaDim=np.array([[[1, 2, 3], [4, 5, 6],
    [7, 8, 9]],
    [[20, 21, 22], [23,24,25], [26,27 ,28]]])
>>>

```

Gambar 3.15 menunjukkan larik yang terbentuk oleh perintah di atas.



Gambar S.15 Larik berdimensi tiga

Contoh berikut menunjukkan cara mengakses elemen larik berdimensi tiga:

```
.....
>>> tigaDim[0, 0, 1] {}
2
>>> tigaDim[0, 0, 2] {}
3
>>> tigaDim[0][0][2] {}
3
>>>
```

Tampak bahwa terdapat tiga indeks yang disebutkan dalam `tigaDim` untuk mengakses suatu elemen larik berdimensi tiga. Notasi seperti `tigaDim [0, 0, 2]` boleh ditulis menjadi `tigaDim [0][0][2]`.

Metode `shape ()` dapat dipakai untuk mendapatkan jumlah baris dan kolom pada larik berdimensi tiga. Contoh:

```
.....
: >>> tigaDim.shape (/}
:
: (2, 3, 3)
:
: »
: .....'
```

Pada contoh ini, angka pertama menyatakan jumlah baris, angka kedua menyatakan jumlah kolom, dan angka ketiga menyatakan jumlah kedalaman. Contoh berikut menunjukkan cara mendapatkan jumlah baris, jumlah kolom, dan jumlah kedalaman secara individual:

```
>>> jumBaris = tigaDim.shape[0] (/}
>>> jumKolom = tigaDim.shape[1] (/}
>>> jumKedalaman = tigaDim.shape[2] (/}
>>> print(jumBaris) (/}
2
>>> print(jumKolom) (/}
3
>>> print(jumKedalaman) (/}
3
>>>
```

3.8.6 Metode-Metode Khusus

Modul `numpy` menyediakan sejumlah metode untuk membentuk matriks tertentu.

Tabel S.6 Metode-metode untuk pembentukan matriks

Metode	Keterangan
<code>zeros()</code>	<p>Metode ini berguna untuk memperoleh matriks yang semua elemennya bernilai 0. Argumen berupa tupel yang mengandung dua elemen, dengan elemen pertama menyatakan jumlah baris dan elemen kedua berupa jumlah kolom. Contoh berikut menghasilkan matriks berukuran 3 x 2 dengan semua elemen bernilai 0:</p> <pre>>>> np.zeros((3,2))<9 array([[0., 0.], [0., 0.], [0., 0.]]) >>></pre>
<code>ones()</code>	<p>Metode ini berguna untuk memperoleh matriks yang semua elemennya bernilai 1. Argumen berupa tupel yang mengandung dua elemen, dengan elemen pertama menyatakan jumlah baris dan elemen kedua berupa jumlah kolom. Contoh berikut menghasilkan matriks berukuran 3 x 2 dengan semua elemen bernilai 1:</p> <pre>>>> np.ones((3,2))<9 array([[1., 1.], [1., 1.], [1., 1.]]) >>></pre>
<code>full()</code>	<p>Metode ini berguna untuk memperoleh matriks yang semua elemennya bernilai sama. Argumen pertama berupa tupel yang mengandung dua elemen, dengan elemen pertama menyatakan jumlah baris dan elemen kedua berupa jumlah kolom. Argumen kedua berupa nilai untuk elemen masing-masing. Contoh berikut menghasilkan matriks berukuran 3 x 2 dengan semua elemen bernilai 5:</p>

Metode	Keterangan
	<pre>>>> np.full((3, 2), 5) array([[5, 5], [5, 5], [5, 5]]) >>></pre>
eye()	<p>Metode ini berguna untuk memperoleh matriks identitas berukuran n x n. Matriks identitas adalah matriks yang semua elemen pada indeks j, j bernilai 1. Contoh:</p> <pre>>>> np.eye(4) array([[1., 0., 0., 0.], [0., 1., 0., 0.], [0., 0., 1., 0.], [0., 0., 0., 1.]]) >>></pre>
random.random()	<p>Metode ini berguna untuk memperoleh matriks yang semua elemennya bernilai acak. Argumen berupa tupel yang mengandung dua elemen, dengan elemen pertama menyatakan jumlah baris dan elemen kedua berupa jumlah kolom. Contoh berikut menghasilkan matriks berukuran 3 x 2 dengan semua elemen bernilai acak :</p> <pre>>>> np.random.random((3, 2)) array([[0.77925582, 0.36617836], [0.2119073 , 0.07329793], [0.89759981, 0.21463116]]) >>></pre>

3.8.7 Tipe Data Elemen Larik

Tipe elemen data larik dapat diketahui melalui:

namaLarik.dtype

Contoh:

```
.....
>>> x = np.array([5, 8, 4, 1, 2])
>>> print(x.dtype)
int32
>>>
-----
>>> y = np.array([5.0, 8.0, 4.0, 1.0, 2.0])
>>> print(y.dtype)
float64
>>>
-----
```

Pada contoh pertama, tipe data elemen larik berupa bilangan bulat 32 bit, sedangkan tipe elemen pada contoh kedua berupa bilangan pecahan 64 bit.

Pada saat membentuk larik, tipe data elemen larik juga bisa disebutkan.

Contoh:

```
.....
>>> z = np.array([5, 8, 4, 1, 2], dtype=np.int64)
>>> print(z.dtype)
int64
>>>
-----
```

Pada contoh ini, `dtype = np. uint64` menyatakan bahwa tipe data berupa `int64`.

3.8.8 Pengaksesan Larik

Pengaksesan elemen-elemen larik dapat dilakukan secara individual ataupun secara berkelompok. Irisan yang berlaku pada senarai juga berlaku pada larik untuk memperoleh sejumlah elemen. Sebagai contoh, `matriks[:2, 1:3]` berarti "2 baris pertama dan kolom 1 dan 2".

Hal ini diperlihatkan berikut ini:

```

>>> matriks = np.array([[1,2,3,4], [5,6,7,8], <Ji
... [9, 10, 11, 12], [13, 14, 15, 16]]) <Ji
>>> matriks</i
array ( [ [ 1, 2, 3, 4],
          [ 5, 6, 7, 8],
          [ 9, 10, 11, 12],
          [13, 14, 15, 16]])
>>> matriks [: 2, 1:3] <Ji
array([[2, 3],
       [6, 7]])
>>>

```

3.8.9 Operasi Matematika pada Larik

Operator +, -, dan * dapat digunakan pada dua larik yang berukuran sama. Hubungan yang berlaku pada ketiga operator ini adalah seperti berikut:

$$C_{ij} = A_{ij} + B_{ij}$$

$$C_{ij} = A_{ij} - B_{ij}$$

$$C_{ij} = A_{ij} * B_{ij}$$

Conteh:


```

>>> matA = np.array([[1, 2], [3, 4]])
>>> matB = np.array([[5, 8], [7, 9]])
>>> matA
array([[1, 2],
       [3, 4]])
>>> matB
array([[5, 8],
       [7, 9]])
>>> mate = matA + matB
>>> mate
array([[6, 10],
       [10, 13]])
>>> mate = matA - matB
>>> mate
array([[ -4,  -6],
       [ -4,  -5]])
>>> mate = matA * matB
>>> mate
array([[ 5, 16],
       [21, 36]])
>>>

```

Sebagai contoh, terdapat dua matriks seperti berikut:

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

Perkalian matriks pada $C = A \times B$ dinyatakan dengan rumus seperti berikut:

$$C = \begin{bmatrix} a_{11} * b_{11} + a_{12} * b_{21} & a_{11} * b_{12} + a_{12} * b_{22} \\ a_{21} * b_{11} + a_{22} * b_{21} & a_{21} * b_{12} + a_{22} * b_{22} \end{bmatrix}$$

Pada keadaan seperti ini, C diperoleh dengan menggunakan metode dot. Contoh:

```

>>> mate = matA.dot(matB)
>>> mate
array([[ 19,  26],
       [ 43,  60] ])
>>>

```

Contoh ini menggunakan `matA` dan `matB` yang dibuat pada contoh sebelumnya.

Perkalian `matA` dan `matB` bisa juga diperoleh dengan cara seperti berikut:

```

.....:
>>>mate= np.dot(matA, matB)
>>> mate
array([[ 19,  26],
       [ 43,  60] ])
>>>
.....:

```

Pada matriks, terdapat metode bernama `T` yang berguna untuk melakukan operasi transpose. Contoh:

```

>>> m = np.array([[1,2,3], [4,5,6], [7,8,9]])
>>> m
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> m.T
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
>>>

```