

KATA PENGANTAR

Buku ini merupakan pengenalan terhadap pemrograman komputer menggunakan Python yang memfokuskan pada dasar-dasar dan pembelajaran yang efektif. Buku ini dirancang untuk beragam siswa yang mempunyai minat dan kemampuan beragam. Buku ini cocok untuk mata kuliah pemrograman pertama bagi mahasiswa komputer dan rekayasa maupun disiplin lain. Tidak diperlukan pengalaman pemrograman, dan hanya memerlukan bekal aljabar sekolah menengah atas. Buku ini menggunakan pendekatan tradisional, menekankan struktur kendali, fungsi, dekomposisi procedural, dan struktur data. Python merupakan bahasa tingkat tinggi yang lebih lambat daripada C. Tetapi Python memberi banyak cara mengoptimalkan bagian-bagian penting dari kode. Tujuan buku ini adalah menghadirkan prinsip dasar pengembangan model komputasional untuk berbagai aplikasi

Penulis

DAFTAR ISI

KATA PENGANTAR	1
DAFTAR ISI.....	2
BAB I PENGENALAN PEMROGRAMAN VISUAL PYTHON	1
Tujuan	1
Sejarah Visual Python.....	1
Menggunakan VPython melalui GlowScript	2
Menggunakan VPython melalui Teks Editor.....	6
Memasang Anaconda Python	6
Memasang Pustaka VPython melalui Anaconda	7
Memuat Skrip dengan Spyder.....	8
Kesimpulan	10
BAB II MASUKAN / LUARAN.....	11
Tujuan	11
Masukan Pengguna	11
Masukan Bilangan	13
Luaran Terformat	14
Kesimpulan	19
BAB III PERCABANGAN	20
Tujuan	20
Percabangan Sederhana	21
Percabangan dengan 2 Keputusan	24
Percabangan dengan beberapa keputusan	27
Kesimpulan	30

BAB IV PERULANGAN	32
Tujuan	32
Perulangan Tentu	32
Perulangan Tak Tentu	34
Bentuk lain dari potongan program di atas dapat dilihat sebagai berikut:	38
Kesimpulan	38
BAB V LISTS, STRING, DAN FILES	40
Tujuan	41
Definisi <i>List</i>	41
Pembuatan <i>List</i>	42
Pengaksesan Elemen <i>List</i>	44
Kesalahan umum.....	45
Penggunaan tanda siku.....	46
Pemanfaatan perulangan	47
Penggandaan rujukan <i>list</i>	49
<i>Negative subscript</i>	51
Operasi <i>List</i>	52
Penambahan Elemen	53
Penyisipan Elemen.....	54
Pencarian Elemen.....	57
Penghapusan Elemen	59
Penggabungan <i>List</i>	63
Pengujian Kesamaan	64
Penjumlahan, Maksimal, Minimal, dan Penyortiran	64
Penggandaan <i>List</i>	65
Pembacaan Masukan.....	66

Pembuatan Tabel.....	68
Pengaksesan Elemen Tabel.....	71
Pengalokasian Elemen Tetangga	72
Penghitungan Jumlah Baris dan Kolom.....	73
String.....	76
<i>Files</i>	81
Pemrosesan File	82
Kotak Dialog.....	88
Kesimpulan	93
BAB VI FUNGSI.....	95
Tujuan	96
Fungsi dari Fungsi.....	96
Pendefinisian Fungsi.....	98
Pemanggilan Fungsi.....	100
Kategori Fungsi.....	102
Fungsi Sederhana	103
Pemanggilan Fungsi yang Mengembalikan Nilai	103
Pemanggilan Fungsi dengan Argumen	108
Fungsi Matematika <i>Built-In</i>	111
Fungsi Rekursif.....	118
Kesimpulan	124
BAB VII ANIMASI.....	126
Tujuan	126
Pendahuluan.....	127
Obyek 3D	128
<i>Variable Assignment</i>	135
Perulangan pada Animasi	142

<i>Lists</i> pada Animasi	149
Kesimpulan	154
DAFTAR PUSTAKA	156
INDEKS	A

BAB I

Pengenalan Pemrograman Visual Python

Tujuan

- Mampu menceritakan keterkaitan Visual Python dengan Python
- Mampu membuat skrip sederhana menggunakan Glowscript
- Mampu membuat skrip sederhana menggunakan Spyder

Sejarah Visual Python

Visual Python atau disingkat dengan VPython merupakan pustaka pemrograman yang awalnya diprakarsai oleh David Scherer di tahun 2000. Waktu itu pustaka pemrograman tersebut diberi nama Classic VPython, Setahun kemudian, dia bersama Bruce Sherwood memulai pengembangan GlowScript, sebuah lingkungan pemrograman yang mirip namun

dapat berjalan pada *web browser*. Di tahun 2014, sebuah bahasa pemrograman yang mirip dengan Python, RapydScript, memungkinkan VPython berjalan pada lingkungan GlowScript.

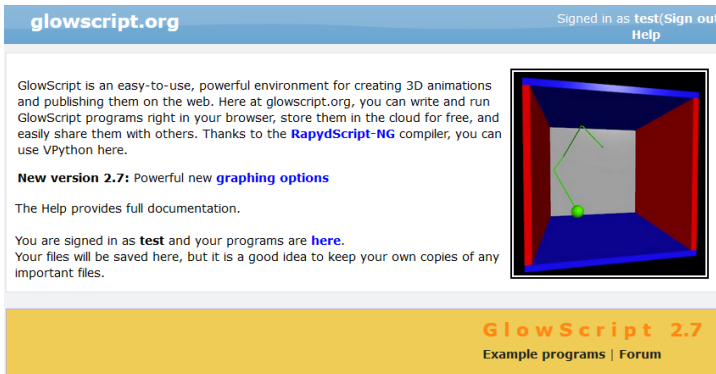
Pustaka VPython memungkinkan menulis program yang menghasilkan animasi 3 dimensi secara *real-time*. VPython dikembangkan berdasarkan bahasa pemrograman Python yang secara luas digunakan pada kelas pengenalan bahasa pemrograman.

Menggunakan VPython melalui GlowScript



Seperti penjelasan di atas, VPython dapat ditulis pada lingkungan *web browser* dengan bantuan GlowScript. Untuk melakukannya, seorang *programmer* melakukan langkah-langkah sebagai berikut:

- Mengakses laman glowsript.org.
- Melakukan pendaftaran pengguna baru pada laman [glowsript](https://glowsript.org) atau masuk dengan menggunakan informasi login Google (Gambar 1.1).



Gambar 1.1. Laman Dashboard Pengguna

- Memberikan nama pengguna yang ingin dipakai pada laman glowsript (Gambar 1.2)

Gambar1.2 Memberikan Informasi Nama Pengguna

Pengenalan

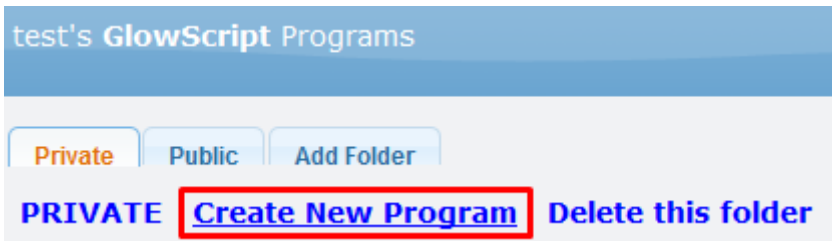
- Mengakses direktori kode sumber (Gambar 1.3)

You are signed in as **test** and your programs are [here](#).

Your files will be saved here, but it is a good idea to keep your own copies of any important files.

Gambar1.3 Akses Direktori Pengguna

- Membuat sebuah program baru (Gambar 1.4)



Gambar1.4 Membuat Program Baru

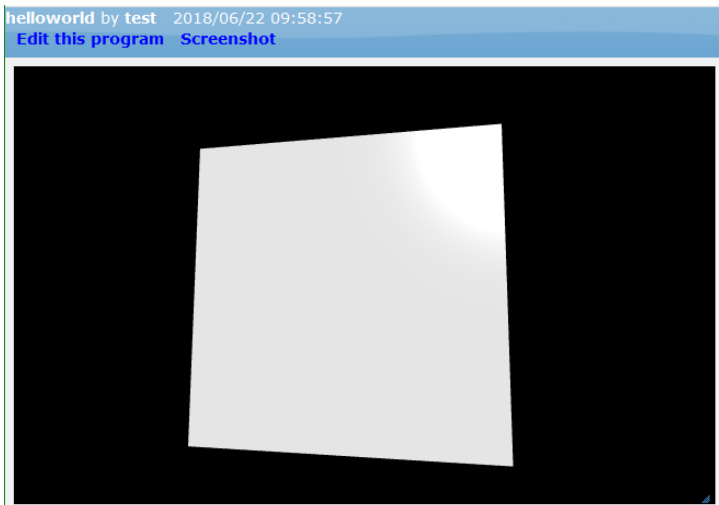
- Sebuah layar editor baru akan muncul dan *programmer* dapat menuliskan kode programnya.
- Di bawah baris bertuliskan “GlowScript 2.6 VPython” ketikkan `box()`. Lalu klik tombol “Run this program” untuk menjalankan program yang telah ditulis (Gambar 1.5)

```
helloworld by test 2018/06/22 09:58:57
Run this program Share or export this program Download

1 GlowScript 2.7 VPython
2 box()
```

Gambar1.5 Membuat Sebuah Kotak

- Gunakan klik kanan tetikus atau CTRL+geser klik kiri untuk mengubah tampilan kamera dari sudut yang berbeda (Gambar 1.6)



Gambar1.6 Tampilan box()

- Untuk melakukan perbesaran (*zoom in* atau *zoom out*), gunakan ALT+geser klik kiri atau roda tetikus (*mouse scrollwheel*).

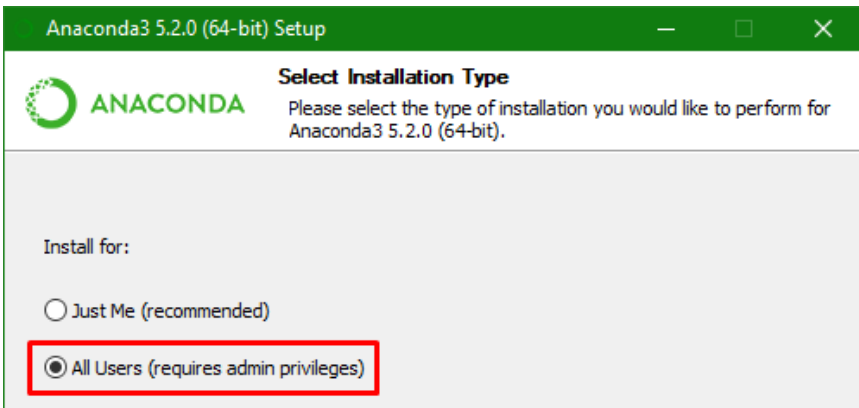
Menggunakan VPython melalui Teks Editor



Memasang Anaconda Python



- Unduh aplikasi pemasangan Anaconda dari situs resmi dengan tautan <https://www.anaconda.com/>. Pilihlah paket instalasi untuk python versi 3.6, dan sesuaikan dengan sistem operasi Windows yang dimiliki.
- Jalankan paket instalasi yang sudah diunduh dengan sempurna, lalu pilihlah instalasi untuk seluruh pengguna (Gambar 1.7).



Gambar1.7 Pemasangan Anaconda untuk Seluruh Pengguna

- Lanjutkan proses pemasangan hingga selesai.



Memasang Pustaka VPython melalui Anaconda

- Bukalah aplikasi *command prompt* yang ada di Microsoft Windows dengan hak akses *Administrator*.
- Pindahkan lokasi aktif direktori yang ada ke lokasi berikut:
`C:\Program Data\Anaconda3\Script`
- Ketikkan perintah berikut untuk memasang VPython:
`conda install -c vpython vpython atau pip install vpython`
- Proses pemasangan akan didahului dengan pengunduhan paket dari repositori, lalu dipasang pada sistem (Gambar 1.8)

```
Administrator: Command Prompt
C:\ProgramData\Anaconda3\Scripts>conda install -c vpython vpython
Solving environment: done

## Package Plan ##

  environment location: C:\ProgramData\Anaconda3

added / updated specs:
- vpython

The following NEW packages will be INSTALLED:

 autobahn: 18.3.1-py_0 vpython
  txai: 2.9.0-py_0 vpython
  ujson: 1.35-py36_0
  vpython: 0.1.3-py36_1 vpython
  vpython: 7.4.4-py36_0 vpython

Proceed ([y]/n)? y
Preparing transaction: done
Verifying transaction: done
Executing transaction: done

C:\ProgramData\Anaconda3\Scripts>
```

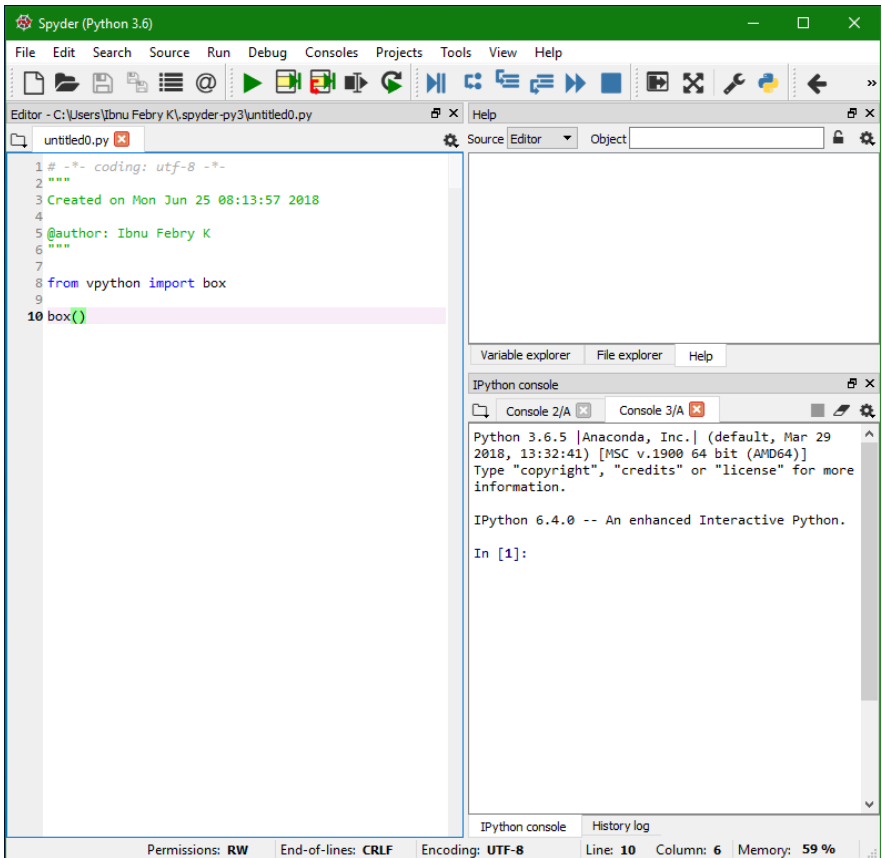
Gambar 1.8 Proses Pemasangan VPython melalui Conda

➡ Memuat Skrip dengan Spyder

Spyder merupakan editor teks yang terpasang otomatis bersamaan dengan pemasangan Anaconda. *Programmer* dapat mengakses aplikasi ini melalui Start Programs milik Microsoft Windows. Tampilan *splash screen* dapat dilihat pada Gambar 9, sedangkan tampilan antarmuka editor terlihat pada Gambar 10.



Gambar 1.9 Splashscreen Aplikasi Spyder



Gambar1.10 Tampilan Editor Spyder

Kesimpulan



Visual Python (VPython) merupakan pustaka pemrograman yang dikembangkan untuk keperluan visualisasi. Pustaka ini dikembangkan menggunakan bahasa pemrograman Python. Tak hanya dapat ditulis menggunakan editor teks, VPython dapat ditulis menggunakan IDE berbasis web, yakni Glowscript.

Pemasangan pustaka ini dapat menggunakan bantuan Anaconda. Anaconda akan memberikan informasi paket dependensi, lalu sekaligus memasangnya. Selain itu, pemasangan Anaconda juga sekaligus memasang teks editor VPython. Editor ini berguna untuk keperluan perhitungan saintifik

BAB II**MASUKAN / LUARAN**

Program yang menarik biasanya meminta para pengguna untuk memberikan nilai masukan (*input*), lalu masukan tersebut akan dikomputasi dan hasilnya akan ditampilkan ke layar. Selain itu, program juga dapat bertindak laku sesuai dengan masukan yang diberikan. Program yang seperti ini disebut dengan program yang interaktif.

Tujuan

- Mampu menerima masukan dari pengguna;
- Mampu mencetak masukan dari pengguna ke layar;
- Mampu mencetak keluaran dari hasil komputasi;

Masukan Pengguna

Program menjadi lebih fleksibel ketika dapat bertindak laku sesuai dengan masukan yang diberikan oleh pengguna daripada memberikan nilai balikan yang tetap. Untuk memulai proses penerimaan

masuk dari pengguna, program mencetak pesan instruksi untuk pengguna. Pesan tersebut diberi istilah “prompt” dan biasanya berisi jenis masukan yang dibutuhkan. Berikut contoh pesan prompt:

```
namadepan = input("Masukkan nama depan  
Anda: ")
```

Fungsi input pada baris perintah di atas menampilkan argumen kalimat pada layar konsol dan menempatkan kursor pada baris yang sama di belakang pesan *prompt*. Jika dijalankan, maka pada layar konsol akan tampil seperti ini:

```
Masukkan nama depan Anda: █
```

Perhatikan jeda antara “:” dengan kursor. Ilustrasi ini memperlihatkan perbedaan visual Antara prompt dengan masukan dari pengguna. Jika prompt sudah tampil, maka program akan menunggu masukan pengguna ke dalam sistem, dan menekan tombol ENTER.

```
Masukkan nama depan Anda: Ahmad█
```

Ilustrasi di atas memperlihatkan kondisi masukan yang telah diberikan ke dalam sistem. Pengguna menuliskan “Ahmad” dan program akan masukan ini ke dalam suatu variabel yang telah dideklarasikan, yakni *namadepan*.

Perhatikan, dan jalankan program berikut pada editor teks Anda:

```
nama1 = input("Masukkan nama depan penulis pertama: ")
nama2 = input("Masukkan nama depan penulis kedua: ")
inisial = nama1[0] + "&" + nama2[0]
print(inisial)
```

Hasil Eksekusi

```
Masukkan nama depan penulis pertama: Ahmad
Masukkan nama depan penulis kedua: Faris
A&F
```

Masukan Bilangan

Perintah fungsi input hanya dapat menerima masukan berupa kalimat (string) dari pengguna. Namun, ada kalanya seorang pengguna ingin memasukkan bilangan sesuai dengan permintaan prompt. Untuk itu, sebuah fungsi konversi bawaan Python dibutuhkan dalam proses penerimaan masukan bilangan. Fungsi tersebut ialah int. Sehingga program Python akan menjadi seperti berikut:

```
masukan = input("Masukkan jumlah barang yang akan dibeli: ")
jumlah_barang = int(masukan)
```

Pada potongan program di atas, masukan bilangan dari pengguna akan dikonversi menjadi bilangan

bulat. Hal ini dikarenakan fungsi `int` bertugas mengkonversi kalimat menjadi bilangan bulat (integer). Jika seorang pengguna ingin memasukkan bilangan pecahan, fungsi `float` digunakan. Perhatikan potongan program berikut:

```
masukan = input("Masukkan jumlah harga  
barang: ")  
harga_barang = float(masukan)
```

Luaran Terformat

Python menyediakan fasilitas untuk mencetak variabel dengan format-format tertentu. Format yang dimaksud dapat berupa presisi bilangan, maupun spasi. Sebagai contoh seorang *programmer* ingin mencetak bilangan pecahan dengan presisi 2 digit di belakang koma seperti berikut

```
Nilai PI: 3.14
```

Namun, luaran seperti berikut ingin dihindari oleh *programmer*

```
Nilai PI: 3.14159265358979323846
```

Untuk memfasilitasi keperluan tersebut, sintaks berikut dapat digunakan:

```
formatstring % (nilai1, nilai2, ..., nilain)
```

dimana `formatstring` merupakan sintaks format untuk keperluan pencetakan bilangan sesuai dengan tipe data yang digunakan.

Sehingga, perintah berikut dapat digunakan dalam Python:

```
print("%.2f" % PI)
```

Selain itu, *programmer* juga dapat menentukan lebar *field* (jumlah karakter berikut dengan spasi) yang ingin dicetak. Perhatikan potongan program berikut

```
print("%10.2f" % PI)
```

Hasil dari potongan program di atas akan tercetak rata kanan menggunakan 10 karakter, yakni 6 spasi diikuti dengan 4 karakter 3.14. Berikut ilustrasi yang dimaksud:

						3	.	1	4
--	--	--	--	--	--	---	---	---	---

Eksprei `%10.2f` disebut sebagai *format specifier* yang mendeskripsikan bagaimana suatu nilai harus diformat. Huruf `f` di akhir *format specifier* menunjukkan bahwa bilangan yang diformat merupakan bilangan pecahan (*floating-point*). Sedangkan, `d` digunakan untuk nilai bilangan bulat, serta `s` untuk string.

Seorang *programmer* juga dapat mencetak beberapa nilai dengan sebuah operasi format string, namun

harus dalam lingkup tanda kurung dan dipisahkan dengan tanda koma. Perhatikan contoh berikut:

```
print("Jumlah: %d Total: %10.2f" % (jumlah, total))
```

Saat lebar *field* telah ditentukan, nilai-nilai yang akan dicetak akan dalam kondisi rata kanan sesuai dengan nilai yang ditentukan. Rata kanan merupakan format umum untuk bilangan, namun tidak untuk string. Perhatikan contoh berikut:

```
teks1 = "jumlah:"  
teks2 = "harga:"  
print("%10s %10d" % (teks1, 15))  
print("%10s %10.2f" % (teks2, 100))
```

contoh tersebut akan menghasilkan teks berikut:

jumlah:	15
harga:	100

Hasilnya akan jauh lebih baik jika teks rata kiri, dan angka rata kanan. Oleh karena itu untuk membuat suatu format rata kiri, tanda minus “-“ digunakan. Sehingga potongan program di atas akan menjadi berikut:

```
teks1 = "jumlah:"  
teks2 = "harga:"  
print("%-10s %10d" % (teks1, 15))  
print("%-10s %10.2f" % (teks2, 100))
```

dan memiliki luaran sebagai berikut:

jumlah:	15
harga:	100

Contoh *format specifier*:

Format string	Contoh luaran	Deskripsi					
“%d”	<table><tr><td>1</td><td>7</td></tr></table>	1	7	Gunakan ‘d’ untuk bilangan bulat			
1	7						
“%5d”	<table><tr><td></td><td></td><td></td><td>1</td><td>7</td></tr></table>				1	7	Spasi ditambahkan sehingga lebar <i>field</i> adalah 5
			1	7			
“%05d”	<table><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>7</td></tr></table>	0	0	0	1	7	Karakter “0” ditambahkan sebelum nilai untuk memenuhi <i>field</i> yang panjangnya 5
0	0	0	1	7			
“%f”	<table><tr><td>3</td><td>.</td><td>1</td><td>4</td><td>1</td></tr></table>	3	.	1	4	1	Gunakan ‘f’ untuk bilangan pecahan
3	.	1	4	1			

“.2f”	<table><tr><td>3.</td><td>1</td><td>4</td></tr></table>	3.	1	4	Cetak 2 digit di belakang koma				
3.	1	4							
“7.2f”	<table><tr><td></td><td></td><td></td><td>3</td><td>.</td><td>1</td><td>4</td></tr></table>				3	.	1	4	Spasi ditambahkan sehingga panjang <i>field</i> adalah 7
			3	.	1	4			
“%s”	<table><tr><td>b</td><td>a</td><td>g</td><td>u</td><td>s</td></tr></table>	b	a	g	u	s	Gunakan “s” untuk jenis teks		
b	a	g	u	s					
“%7s”	<table><tr><td></td><td></td><td>b</td><td>a</td><td>g</td><td>u</td><td>s</td></tr></table>			b	a	g	u	s	Spasi ditambahkan sehingga panjang <i>field</i> adalah 7. Hasilnya teks berada dalam kondisi rata kanan
		b	a	g	u	s			
“-7%s”	<table><tr><td>b</td><td>a</td><td>g</td><td>u</td><td>s</td><td></td><td></td></tr></table>	b	a	g	u	s			Spasi ditambahkan sehingga panjang <i>field</i> adalah 7. Hasilnya teks berada
b	a	g	u	s					

		dalam kondisi rata kiri.
--	--	--------------------------------

Kesimpulan

Python memberikan fleksibilitas pada program yang dibuat dengan adanya layanan masukan-luaran. Layanan ini memungkinkan adanya perbedaan perilaku program sesuai dengan masukan yang diberikan oleh pengguna. Untuk mencakup masukan dengan beberapa format, fungsi konversi juga disediakan oleh python.

Python menyediakan berbagai macam *format specifier* untuk keperluan luaran terformat baik untuk variabel teks maupun bilangan. Luaran terformat bilangan juga dibedakan untuk bilangan bulat dan pecahan.

BAB III**PERCABANGAN**

Dalam pemrograman terdapat tatanan baris instruksi yang ditulis oleh *programmer* untuk menyelesaikan permasalahan besar yang dihadapi. Tatanan ini juga disebut sebagai *sequencing*. Urutan instruksi-instruksi ini berkaitan satu dengan lainnya. Namun, teknik ini saja seringkali tidak cukup untuk menyelesaikan permasalahan yang dihadapi. Adakalanya beberapa aksi harus dilakukan terhadap beberapa kondisi yang dihadapi. Hal ini disebut sebagai percabangan.

Percabangan ini akan membuat program berjalan seauai dengan konteks permasalahan yang disesuaikan dengan kondisi yang diterima oleh sistem. Kondisi satu dengan lainnya bisa saja membutuhkan aksi yang berbeda. Sehingga, bisa saja

Tujuan

- Mampu membuat percabangan sederhana
- Mampu membuat percabangan dengan 2 keputusan

- Mampu membuat percabangan dengan beberapa keputusan

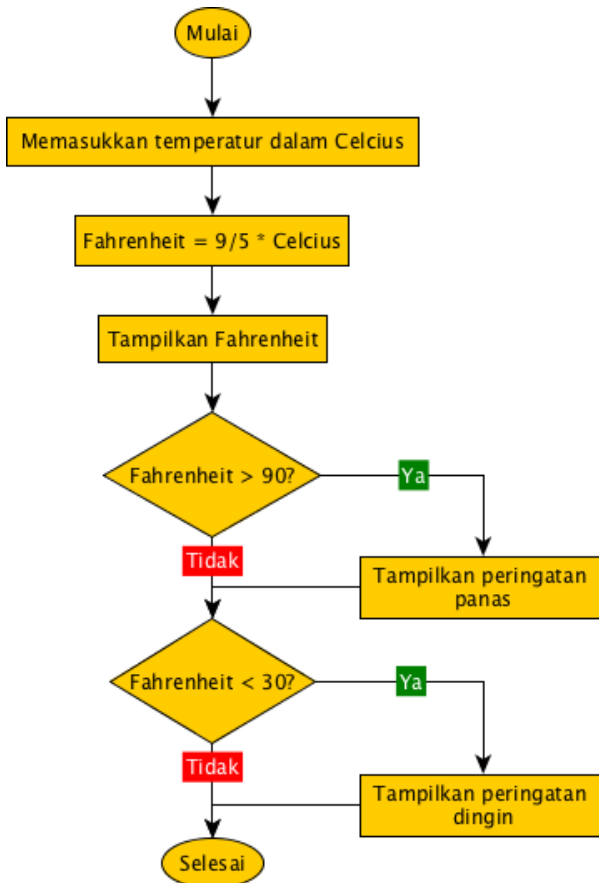
Percabangan Sederhana

Salah satu bentuk percabangan ialah percabangan sederhana dengan 2 keputusan sederhana di akhir. Dalam Python indentasi mengindikasikan suatu langkah harus dilakukan jika hanya kondisi di dalam baris sebelumnya bernilai benar.

Dalam percabangan sederhana, terdapat pengujian kondisi yang terletak di dalam sintaks if. Jika kondisi yang diuji bernilai benar, maka baris instruksi yang terdapat di badan percabangan if akan dieksekusi. Sebagai contoh aplikasi konsep ini, Anda dapat membayangkan sebuah program pemeriksa suhu ruangan dalam Fahrenheit. Program tersebut akan memberikan pesan peringatan jika temperatur ruangan terlalu dingin atau panas. Alur berpikir dari program ini dapat dilihat berikut:

```
input temperatur dalam celcius, celcius
hitung fahrenheit = 9/5 celcius + 32
output fahrenheit
jika fahrenheit > 90
    output peringatan suhu panas
jika fahrenheit < 30
    output peringatan suhu dingin
```

Berdasarkan rancangan alur berpikir di atas, *flowchart* program dapat dilihat pada Gambar 3.1. Bab 3



Gambar 3.1. Flowchart Percabangan Sederhana

Akhirnya, program konversi dan peringatan temperatur ruangan dapat disusun seperti berikut:

```
def main():
```

```

Percabangan
print("Program konversi Celcius-
Fahrenheit\n")
celcius = float(input("Masukkan suhu
(celcius): "))
fahrenheit = 9 / 5 * celcius + 32
if fahrenheit > 90:
    print ("Temperatur terlalu panas")
if fahrenheit < 32:
    print ("Temperatur terlalu dingin")

main()

```

Kondisi di dalam instruksi percabangan Python dapat diisi dengan pengecekan suatu variabel dengan menggunakan operator matematis. Adapun operator matematis yang dimaksud dapat dilihat pada Tabel 3.1.

Tabel 3.1. Tabel operasional matematika di dalam Python

Python	Matematika	Arti
<	<	kurang dari
<=	≤	kurang dari atau sama dengan
==	=	sama dengan
>=	≥	lebih besar atau sama dengan
>	>	lebih besar
!=	≠	tidak sama dengan

Percabangan dengan 2 Keputusan

Setelah mempelajari cara untuk mengeksekusi instruksi-instruksi secara selektif, percabangan juga dapat digunakan untuk menangkap suatu kondisi yang tidak diinginkan dan menghentikan eksekusi instruksi berikutnya. Dengan kata lain, percabangan ini digunakan untuk *error checking*.

Perhatikan rumus perhitungan akar-akar persamaan kuadrat berikut:

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

Secara matematis, nilai persamaan tersebut tidak dapat diperoleh jika nilai di dalam akar ($\sqrt{b^2 - 4ac}$) bernilai negatif. Jika bernilai negatif, maka bilangan tersebut adalah bilangan imajiner.

Perhatikan potongan program berikut:

```
import math

def main():
    print("Program menghitung akar-akar persamaan
    kuadrat\n")
```

Percabangan

```
a = float(input("Masukkan nilai koefisien a:"))
b = float(input("Masukkan nilai koefisien b:"))
c = float(input("Masukkan nilai koefisien c:"))

d = math.sqrt(b * b - 4 * a * c)
x1 = (-b + d) / (2 * a)
x2 = (-b - d) / (2 * a)

print ("akar-akar persamaannya adalah: ", x1, x2)

main()
```

jika program di atas dieksekusi, maka pesan *error* berikut ditampilkan:

```
ValueError: math domain error
```

Lalu bandingkan program di atas dengan program berikut:

```
import math

def main():
    print("Program menghitung akar-akar persamaan kuadrat\n")
    a = float(input("Masukkan nilai koefisien a:"))
    b = float(input("Masukkan nilai koefisien b:"))
    c = float(input("Masukkan nilai koefisien c:"))
```

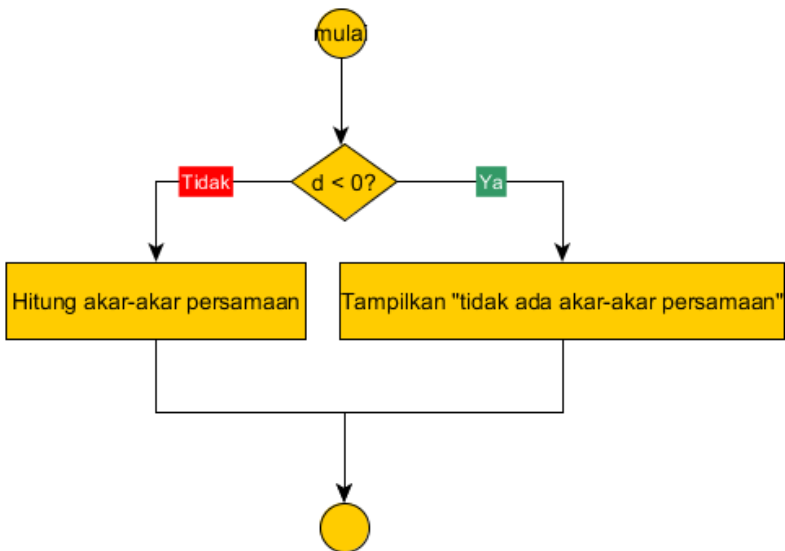
```
d = math.sqrt(b * b - 4 * a * c)

if d < 0:
    print("Persamaan tersebut tidak memiliki
akar-akar persamaan")
else:
    x1 = (-b + d) / (2 * a)
    x2 = (-b - d) / (2 * a)
    print ("akar-akar persamaannya adalah: ", x1,
x2)

main()
```

Program di atas akan menyeleksi apakah hasil perhitungan akar bernilai negatif atau tidak. Sehingga pesan *error* kalkulasi tidak ditemui. Ilustrasi percabangan dalam program di atas dapat dilihat pada Gambar 3.2.

Percabangan



Gambar 3.2. Percabangan dengan 2 Keputusan

Percabangan dengan beberapa keputusan

Percabangan dengan beberapa keputusan akhir disebut juga *nested branch*. Pada percabangan ini pernyataan dari suatu percabangan dapat berisi percabangan lain. Hal ini dimungkinkan karena *programmer* menghendaki ada pemeriksaan lebih lanjut mengenai detail kondisi yang dihadapi.

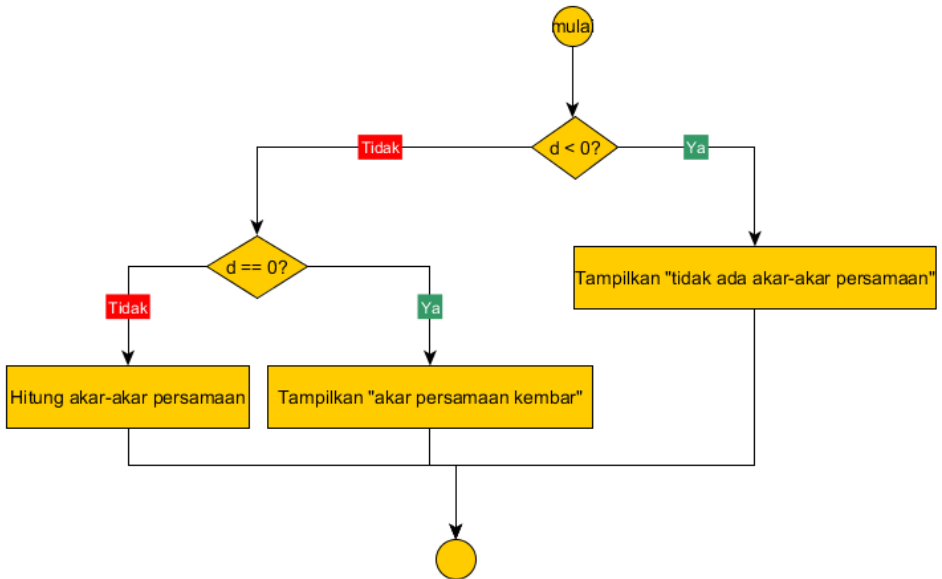
Program perhitungan akar-akar persamaan yang telah dibuat di awal bab ini telah mampu mencegah suatu kondisi kesalahan fatal dalam perhitungan. Namun, dalam perhitungan persamaan kuadrat nilai diskriminan (d) memberikan petunjuk tentang akar-akar penyelesaian. Jika nilai $d < 0$, maka persamaan kuadrat tersebut tidak memiliki penyelesaian. Lalu, jika nilai $d = 0$, maka persamaan kuadrat tersebut memiliki akar kembar. Sebaliknya, jika nilai $d > 0$, maka akar penyelesaian dari persamaan kuadrat tersebut berbeda satu dengan lain. Desain percabangan baru yang mengikuti ketentuan ini adalah sebagai berikut:

```
if d < 0:
    print("Persamaan tersebut tidak memiliki
    akar-akar persamaan")
else:
    if d == 0:
        x = -b/(2*a)
        print("Akar persamaan kembar dengan nilai
        ", x)
    else:
        x1 = (-b + d) / (2 * a)
        x2 = (-b - d) / (2 * a)
        print ("akar-akar persamaannya adalah: ",
        x1, x2)

main()
```

Alur program di atas jika diilustrasikan dalam *flowchart* dapat dilihat pada Gambar 3.3.

Percabangan



Gambar 3.3. Flowchart Percabangan Bersarang

Percabangan bersarang atau *nested branch* dalam Python memiliki batas hingga 4 kedalaman. Jika seorang *programmer* ingin melakukan pengujian lebih lanjut, Python menyediakan sintaks percabangan lain yang memiliki tampilan semantik yang lebih baik. Sintaks `elif` dapat digunakan untuk kebutuhan ini. `elif` merupakan kependekan dari kata “else if”. Sintaks ini memungkinkan *programmer* menguji kebenaran suatu kondisi dengan memberikan argumen. Bentuk program dalam Python akan menjadi seperti berikut:

```
if <kondisi 1>:
```

```

    <pernyataan 1>
elif <kondisi 2>:
    <pernyataan 2>
else
    <pernyataan 3>

```

Dengan menggunakan sintaks ini, program perhitungan akar-akar persamaan yang baru akan menjadi seperti berikut:

```

if d < 0:
    print("Persamaan tersebut tidak memiliki
akar-akar persamaan")
elif d == 0:
    x = -b/(2*a)
    print("Akar persamaan kembar dengan nilai ",
x)
else:
    x1 = (-b + d) / (2 * a)
    x2 = (-b - d) / (2 * a)
    print ("akar-akar persamaannya adalah: ", x1,
x2)

main()

```

Kesimpulan

Terdapat 3 jenis percabangan yang dapat digunakan oleh *programmer*, yakni percabangan sederhana, percabangan dengan 2 keputusan, serta percabangan dengan beberapa keputusan. Penggunaan konsep-

Percabangan

konsep percabangan ini didasarkan pada kebutuhan pengujian kondisi saat program berjalan. Detail pengujian kondisi juga bervariasi bergantung pada tingkat pengujian dan keputusan yang akan dibuat.

Percabangan sederhana hanya menguji suatu kondisi. Jika kondisi yang dikehendaki didapat, maka badan percabangan akan dieksekusi. Selanjutnya, percabangan dengan 2 keputusan melakukan pengujian terhadap validitas kondisi dan memberikan instruksi bila kondisi tersebut baik ditemui maupun tidak. Percabangan dengan beberapa keputusan disebut juga percabangan bersarang. Dalam konsep ini, di dalam badan percabangan akan terdapat percabangan lain. Sehingga konsep ini memiliki detail pemeriksaan kondisi terbaik.

BAB IV

PERULANGAN

Instruksi perintah lain yang penting dalam pemrograman ialah perulangan. Perulangan memberikan fleksibilitas kepada *programmer* untuk melakukan suatu instruksi secara iteratif baik secara definitif (sudah ditentukan) maupun kondisional. Dengan adanya konsep perulangan ini, *programmer* tidak perlu menuliskan baris instruksi yang sama secara berulang kali.

Tujuan

- Mampu memahami perulangan tentu / definitif
- Mampu memahami perulangan tak tentu / kondisional

Perulangan Tentu

Salah satu bentuk perulangan dalam pemrograman ialah perulangan tentu atau definitif. Perulangan ini memiliki awalan, batas variabel kontrol, hingga cara perbaruan variabel kontrol. Bentuk perulangan

Perulangan

definitif ini ialah sintaks `for`, yang memiliki struktur seperti berikut:

```
for <var> in <sequence>
    <body>
```

Variabel index perulangan `var` akan diperbarui nilainya tiap satu babak perulangan selesai, dan pernyataan di dalam badan perulangan akan dieksekusi sekali dalam tiap babak perulangan.

Sebagai contoh seorang *programmer* ingin membuat program yang dapat menghitung rata-rata deret bilangan yang dituliskan oleh pengguna. Untuk membuat program tersebut umum, program harus menerima berapapun jumlah bilangan yang diberikan pengguna. Untuk mencari rerata dari suatu kumpulan bilangan, bilangan-bilangan tersebut dijumlah lalu dibagi dengan banyaknya kumpulan bilangan. *Programmer* tidak perlu tahu seluruh bilangan masukan dari pengguna. *Programmer* cukup menjumlah bilangan setiap kali dimasukkan oleh pengguna lalu menghitung reratanya di akhir program. Formulasi penyelesaian tersebut dapat dilihat sebagai berikut

```
input banyaknya bilangan yang ingin dihitung, n
inisialisasi variabel total ke 0
ulangi sebanyak n
    input sebuah angka, x
    jumlahkan x ke variabel total
```

output rerata sama dengan total / n

Jika formulasi penyelesaian di atas dituliskan dalam program Python, maka akan terlihat seperti berikut:

```
def main():
    n = int(input("Masukkan jumlah bilangan yang
dihitung: "))
    total = 0.0
    for i in range(n):
        x = float(input("Masukkan sebuah bilangan:
"))
        total = total + x
    print ("\nRata-rata bilangan adalah: ",
total/n))

main()
```

Perulangan Tak Tentu

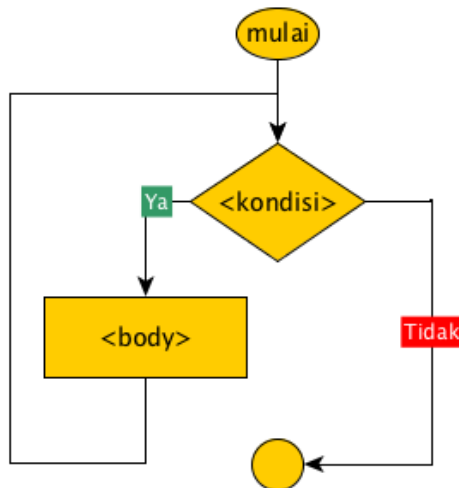
Berbeda dengan perulangan definitif, perulangan tak tentu atau kondisional tidak punya batas yang jelas di awal badan perulangan. Perulangan ini tetap beriterasi hingga suatu kondisi tertentu. Sehingga, tidak ada jaminan kapan iterasi tersebut berlangsung.

Dalam Python, sebuah perulangan tak tentu diimplementasikan dengan sintaks `while`. Bentuk sintaks dari perintah ini sebagai berikut:

Perulangan

```
while <kondisi>  
    <body>
```

Pada ungkapan di atas, kondisi merupakan pernyataan Boolean, seperti pernyataan `if`. Bagian `body` merupakan baris-baris instruksi. Ungkapan instruksi `while` ini cukup jelas (Gambar 4.1). Badan perulangan ini mengeksekusi secara berkali-kali selama kondisi bernilai `true` (benar). Saat kondisi bernilai `false` (salah), perulangan ini akan berhenti. Perhatikan pada sintaks di atas, terlihat bahwa kondisi akan diuji di awal perulangan. Konfigurasi ini disebut sebagai *pre-test loop*. Jika kondisi awalnya bernilai `false`, maka badan perulangan tidak akan dieksekusi sama sekali.



Gambar 4.1. Flowchart Perulangan Tak Tentu

Program perhitungan rerata bilangan dengan sintaks for di atas, dapat dituliskan seperti berikut

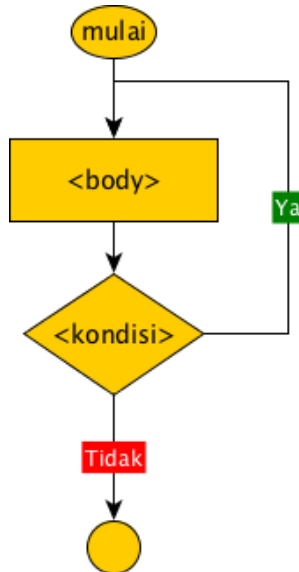
```
def main():
    n = int(input("Masukkan jumlah bilangan yang
dihitung: "))
    total = 0.0
    i = 0
    while i < n:
        x = float(input("Masukkan sebuah bilangan:
"))
        total = total + x
        i++
    print ("\nRata-rata  bilangan  adalah: ",
total/n))

main()
```

Perhatikan program di atas memiliki inisialisasi variabel kontrol *i* yang diawali dengan nilai 0 (*i*=0) dan diperbarui nilainya di akhir perulangan (*i*++). Perulangan tersebut akan berhenti hingga nilai *i* sama dengan *n* (*i* < *n*).

Selain perulangan tak tentu dengan pengecekan kondisi di awal (*pre-test loop*), terdapat konsep perulangan lain dengan pengujian kondisi di akhir iterasi. Konsep ini disebut dengan *post-test loop*. Gambaran diagram alir untuk konsep ini dapat dilihat pada Gambar 4.2.

Perulangan



Gambar 4.2. Flowchart Post-test Loop

Salah satu contoh implementasi *post-test loop* ini adalah sebuah program yang menerima input berupa bilangan positif. Jika pengguna memasukkan bilangan lain, maka program akan meminta input bilangan lain. Proses ini juga disebut sebagai validasi input (*input validation*). Program yang baik hendaknya dapat memeriksa input jika memungkinkan. Contoh program berikut memperlihatkan penggunaan *post-test loop* untuk keperluan validasi input.

```
def main():  
    inp = -1  
    while inp < 0:
```

```
inp = int(input("Masukkan sebuah bilangan
positif: "))

print ("\nBilangan Anda adalah: ", inp))
main()
```

Bentuk lain dari potongan program di atas dapat dilihat sebagai berikut:

```
def main():

    while True:
        inp = int(input("Masukkan sebuah bilangan
positif: "))
        if inp < 0:
            print("Anda memasukkan bukan bilangan
positif")
        else:
            break
    main()
```

Pada potongan program di atas, terlihat pengujian kondisi bilangan yang dimasukkan pengguna. Jika bilangan bernilai negatif, maka muncul pesan peringatan. Sebaliknya, jika input bilangan valid, maka perulangan akan dihentikan (sintaks break).

Kesimpulan

Terdapat 2 jenis perulangan di dalam Python, yakni perulangan definitif dan perulangan tak tentu.

Perulangan

Perulangan definitif memiliki batas jumlah perulangan yang ditentukan di awal, sedangkan perulangan tak tentu tidak memiliki jumlah iterasi yang pasti. Banyaknya iterasi dalam perulangan tak tentu didapatkan berdasarkan pengujian kondisi saat program dijalankan.

Berdasarkan mekanisme pengujian kondisi perulangan, perulangan tak tentu / kondisional dibagi menjadi 2 jenis yaitu *pre-test loop* dan *post-test loop*. *Pre-test loop* mengharuskan pengujian kondisi terletak di awal perulangan, sebaliknya *post-test loop* melakukan pengujian kondisi di akhir perulangan.

BAB V

LISTS, STRING, DAN FILES

Dalam beberapa program, kita perlu mengumpulkan nilai-nilai dalam jumlah yang besar. Dalam bahasa pemrograman, sering kali kita menjumpai array sebagai media penyimpanan nilai-nilai tersebut. Setiap array memiliki nama dan indeks yang merujuk pada alamat atau urutan penyimpanan nilai pada array tersebut. Setiap nilai yang tersimpan di dalam array tersebut disebut elemen array.

Python menggunakan *list* dalam merepresentasikan fungsi dari array. *List* dapat berkembang secara otomatis ketika adanya perubahan jumlah banyaknya nilai di dalamnya. Perbedaan berikutnya dengan array adalah kemampuan *list* yang mampu menyimpan nilai-nilai dalam berbagai jenis tipe data (contoh: bilangan bulat atau bilangan decimal/pecahan).

Dalam pemrograman, string merupakan salah satu bagian yang tidak terhindarkan ketika kita membahas tentang array. Melalui pemrosesan string, kita akan mempelajari operasi-operasi string dan

beberapa fungsi yang digunakan ketika kita mengolah *list*, *String*, dan *Files* string tersebut.

Dalam bab ini, kita akan mempelajari tentang *list* dan beberapa algoritma untuk memprosesnya. Selain itu, kita juga mempelajari tentang pemrosesan string dan operasi-operasi *file* yang ada di dalam pemrograman Python.



Tujuan

- Mampu menjelaskan definisi *list*;
- Mampu membuat *list*;
- Mampu membuat *string*;
- Mampu membuat *files*.

Definisi *List*



Kita memulai bab ini dengan mengenalkan tipe data dalam *list*. *List* merupakan mekanisme dasar dalam Python untuk mengumpulkan beberapa nilai. Dalam bagian selanjutnya, kita akan belajar mengenai cara membuat *list* dan mengakses elemen *list*.

Pembuatan *List*

Kita dapat membuat sebuah program untuk membaca beberapa nilai dan menampilkannya dalam sebuah susunan serta menandai nilai yang terbesarnya. Kumpulan nilai tersebut antara lain:

```
10
35
23.5
56  => nilai yang terbesar
26
```

Kita tidak dapat mengetahui nilai mana yang ditandai sebagai nilai terbesar sampai kita melihat keseluruhan dari nilai-nilai tersebut. Hal ini dilakukan karena kemungkinan nilai yang terbesar merupakan nilai yang terakhir. Jadi, program harus menyimpan nilai-nilai yang dimasukkan dari awal sampai akhir program dijalankan (nilai-nilai tersebut tercetak ke layar).

Pada dasarnya, kita dapat menyimpan nilai-nilai tersebut ke dalam lima variabel yang berbeda. Namun, penggunaan lima variabel tersebut tidaklah praktis. Kita harus menuliskan kode pembuatan dan pengisian variabel tersebut sebanyak 5 kali. Pada bahasa pemrograman Python, penggunaan *list*

List, String, dan Files

merupakan opsi terbaik dalam menyimpan kumpulan dari nilai tersebut.

Kita dapat membuat sebuah *list* dan memberikan nilai awal untuk disimpan pada *list* tersebut. Untuk mendefinisikan sebuah *list* pada Python, setiap elemen *list* akan dipisahkan dengan koma (,) dan akan berada di antara tanda kurung siku. Sebagai contoh, ada sebuah *list* yang dirujuk oleh sebuah variabel `nilai`, menampung 5 nilai yang telah disebutkan sebelumnya.

```
nilai = [10, 35, 23.5, 56, 26]
```

Tanda kurung siku mengindikasikan adanya sebuah *list* yang mencakup beberapa nilai di dalamnya. Kita dapat menyimpan *list* dalam sebuah variabel sehingga kita dapat mengaksesnya nanti. Data nilai tersebut akan tersusun dalam sebuah *list* sebagai berikut

indeks	→	0	1	2	3	4
nilai	→	10	35	23.5	56	26

Gambar 5.1. Sebuah list dengan ukuran 5

Pengaksesan Elemen List

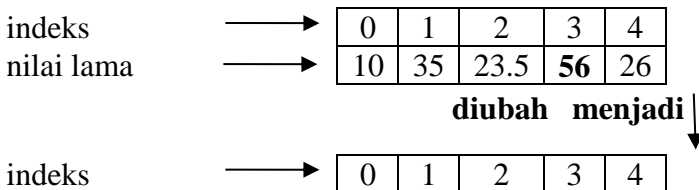
Sebuah *list* berisi dari sekumpulan elemen yang mana masing-masing elemen memiliki posisi yang bulat. Sebuah *list* yang memiliki panjang n , memiliki indeks ke-0 sampai indeks $n-1$ (seperti yang terlihat pada Gambar 5.1). Dimana dalam mengakses sebuah elemen *list*, kita perlu mengetahui indeks mana yang akan digunakan, sebagai contoh:

```
print(nilai[3]) #mencetak elemen pada
               indeks ke-3
```

Pada contoh tersebut, angka yang ditunjuk pada indeks ke-3 pada *list* (56) yang tercetak pada layar. Kita juga dapat mengganti isi dari setiap elemen *list* dengan sebuah nilai yang lain, contohnya:

```
nilai[3] = 87
```

Dalam perintah tersebut, elemen *list* yang dirujuk oleh variabel `nilai` pada indeks ke-3 yang semula terisi dengan 56 akan diubah menjadi 87 (terlihat pada Gambar 5.2).



nilai baru 

10	35	23.5	87	26
----	----	------	----	----

Gambar 5.2. Pengaksesan elemen list

Pengaksesan elemen *list* hanya diperbolehkan dalam jangkauan indeks yang ada di dalam *list* tersebut. Sebagai contoh sebuah *list* dengan ukuran 5 memiliki indeks 0 sampai 4 dan dirujuk oleh sebuah variabel `nilai`. Jadi pengaksesan elemen *list* pada kasus tersebut hanya `nilai[0]`, `nilai[1]`, `nilai[2]`, `nilai[3]`, dan `nilai[4]`. Selain indeks tersebut, kita tidak dapat mengakses elemen di dalam *list* tersebut.

Kesalahan umum

Bentuk kesalahan yang umum dalam pengaksesan elemen *list* adalah ketika kita mengakses suatu indeks yang merupakan jumlah elemen di dalam suatu *list*. Kita sering melupakan bahwa elemen di dalam *list* dimulai dari 0 sampai dengan $n-1$ dengan n adalah jumlah elemen atau ukuran (panjang maksimum) dari sebuah *list*.

Sebuah pengaksesan elemen *list* yang dirujuk oleh variabel `nilai` adalah `nilai[5] = 46`. Pengaksesan terhadap *list* pada indeks ke-5 akan mengalami kesalahan (*error*). Hal ini dikarenakan tidak adanya indeks ke-5 pada *list* tersebut. Meskipun jumlah elemen *list* adalah 5, indeks terakhir yang dapat diakses pada *list* tersebut adalah indeks ke-4.

Jadi kita tidak dapat mengakses *list* tersebut dengan indeks 5 atau selebihnya.

Jika program kita mengakses sebuah *list* dengan menggunakan indeks berada di luar jangkauan *list* tersebut, program akan menghasilkan sebuah pesan eksepsi kesalahan pada saat waktu eksekusi. Salah satu pesan kesalahan dapat dilihat sebagai berikut ini.

```
nilai[5] = 46
IndexError: list assignment index out
of range
```

Dalam mengatasi kesalahan tersebut, kita perlu mengetahui terlebih dahulu jumlah elemen *list* yang dapat diakses dan indeks terakhir yang ada di dalam *list* tersebut.

Penggunaan tanda siku

Sebuah *list* tidak terlepas dengan adanya tanda kurung siku. Ada dua kegunaan yang berbeda dari tanda kurung siku. Ketika tanda kurung siku mengikuti sebuah nama variabel, tanda kurung siku tersebut akan berperan sebagai operator *subscript* sebagai penunjuk indeks dari sebuah elemen *list*, seperti pada perintah `siku[8]`. Pengaksesan elemen (pada indeks ke-8) *list* dilakukan pada perintah `siku[8]` tersebut.

Penggunaan tanda kurung siku lain yang tidak mengikuti sebuah nama variabel, tanda kurung siku tersebut akan berperan dalam pembuatan sebuah *list*. Sebagai contoh `siku = [8]`. *List* tersebut akan berisi sebuah elemen tunggal 8 dan dirujuk oleh sebuah variabel `siku`. Jadi tanda siku dalam *list* dapat berperan sebagai operator penunjuk indeks dari sebuah elemen pada suatu *list* seperti pada contoh yang pertama. Selain itu, tanda siku tersebut juga dapat berguna pada proses pengisian (inisialisasi) awal dari sebuah *list* seperti pada contoh yang kedua.

Pemanfaatan perulangan

Anda dapat menggunakan perulangan dalam mengakses elemen dari *list*. Batas dari perulangan tersebut adalah jumlah indeks terakhir di dalam *list* tersebut. Perintah yang dapat digunakan adalah sebagai berikut.

```
nilai = [10, 35, 23.5, 87, 26]
for i in range(5) :
    print(i, nilai[i])
```

Kode program itu akan menampilkan semua nilai indeks dan elemen yang sesuai dalam sebuah daftar nilai, terdiri dari dua kolom yang berisi nomor indeks dan isi dari *list*. Hasil luaran dari kode program tersebut adalah sebagai berikut.

```

0 10
1 35
2 23.5
3 87
4 26

```

Variabel `i` akan berulang dari 0 sampai 4 karena tidak ada elemen yang sesuai dengan isi dari variabel `nilai[5]`. Jika kita tidak memerlukan nilai indeks. Kita dapat membuat perulangan setiap elemen individu dengan menggunakan perulangan *for loop* dalam bentuk sebagai berikut.

```

nilai = [10, 35, 23.5, 87, 26]
for element in nilai :
    print(element)

```

Kode program itu akan mencetak elemen-elemen yang ada di dalam setiap indeks pada *list*. Hasil luaran dari kode program tersebut adalah sebagai berikut.

```

10
35
23.5
87
26

```

Tampilan luaran tersebut sedikit berbeda dengan tampilan luaran pada kode program pengaksesan *list* dengan menggunakan perulangan yang sebelumnya. Isi setiap elemen *list* ditampilkan tanpa diawali dengan nomor indeksinya.

Penggandaan rujukan *list*

Pada dasarnya variabel nilai tidak menyimpan angka apapun sedangkan *list* akan disimpan di tempat lain dalam sebuah lokasi memori. Variabel `nilai` hanya menyimpan alamat rujukan kepada *list* tersebut. Sebuah variabel *list* juga dapat dirujuk oleh variabel yang lain. Kedua variabel ini pada dasarnya akan mengakses *list* yang sama. Jadi isi dari masing-masing variabel tersebut hanyalah alamat rujukan yang sama-sama menuju ke lokasi penyimpanan *list* tersebut berada. Contoh kode program penggandaan variabel rujukan adalah sebagai berikut.

```
nilai = [10, 35, 23.5, 87, 26]
angka = nilai #penggandaan rujukan
list
angka[2] = 75
for i in range(5) :
    print(i, nilai[i])
```

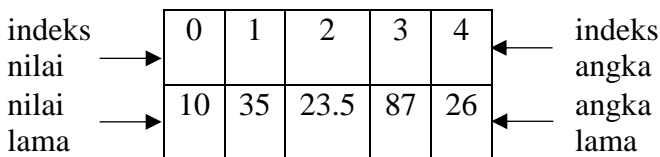
Ada dua buah variabel (`nilai` dan `angka`) di dalam kode program tersebut yang mengakses *list* yang sama. Jadi `nilai` dan `angka` memiliki isi yang sama, yakni alamat rujukan terhadap lokasi *list* berada. Meskipun kita menginisialisasi sebuah *list* menggunakan variabel `nilai`, kita juga dapat mengubah elemen pada indeks ke-2 dari *list* menggunakan variabel `angka`. Hal ini dapat dilakukan ketika variabel `angka` diisi dengan rujukan yang sama dengan variabel `nilai` (melalui perintah `angka =`

nilai). Lalu, melalui variabel `angka`, kita dapat mengubah elemen *list* pada indeks ke-2 (melalui perintah `angka[2] = 75`).

Luaran dari program tersebut adalah isi dari setiap elemen *list* yang ditunjuk oleh variabel `nilai`. Namun, isi dari *list* tersebut mengalami perubahan pada indeks ke-2 sesuai dengan perubahan isi dari *list* yang telah dilakukan melalui variabel `angka`. Jadi, tampilan luaran dari program tersebut adalah sebagai berikut.

```
0 10
1 35
2 75
3 87
4 26
```

Ilustrasi lebih lanjut mengenai konsep *list* dan variabel rujukan yang terkait dapat dilihat pada Gambar 5.3. Dari gambar 5.3, kita dapat melihat ada sebuah *list* yang dapat diakses oleh dua variabel yang berbeda (variabel `nilai` dan `angka`).



diubah menjadi

indeks	→	0	1	2	3	4	←	indeks
nilai	→	10	35	75	87	26	←	angka
nilai baru	→						←	angka baru

Gambar 5.3. List dengan dua variabel rujukan

➡ Negative subscript

Python tidak seperti beberapa bahasa pemrograman yang lain. Python memungkinkan kita untuk menggunakan *negative subscript* ketika mengakses sebuah elemen. *Negative subscript* memungkinkan kita untuk mengakses indeks dari sebuah *list* dalam urutan yang terbalik, seperti yang terlihat pada Gambar 5.4.

indeks	→	0	1	2	3	4
nilai	→	10	35	23.5	56	26
<i>negative subscript</i>	→	-5	-4	-3	-2	-1

Gambar 5.4. Pengaksesan list dengan *negative subscript*

Sebagai contoh, sebuah *subscript* -1 memungkinkan kita mengakses elemen pada indeks terakhir dari sebuah *list*. Kode program ini merupakan contoh pengaksesan elemen indeks terakhir dari sebuah list menggunakan *subscript* -1.

```
nilai = [10, 35, 75, 87, 26]
a = nilai[-1]
b = nilai[-5]
```

```
print("elemen terakhir dari list
adalah ", a)
print("elemen pertama dari list
adalah ", b)
```

Luaran dari program tersebut adalah sebuah tulisan sebagai berikut.

```
elemen terakhir dari list adalah 26
elemen terakhir dari list adalah 10
```

Contoh selanjutnya adalah *subscript -2*. *Subscript* ini merujuk pada indeks ke-2 dari akhir dari *list* sedangkan *subscript -5* merujuk pada elemen pertama dari sebuah *list* (sesuai dengan gambar 5.4). Secara umum, *negative subscript* yang dapat digunakan dalam sebuah variabel penunjuk *list* adalah dari -1 sampai panjang atau ukuran dari *list* tersebut (dalam contoh sebelumnya adalah 5).

Operasi *List*

Bahasa pemrograman Python memiliki kemudahan-kemudahan yang ditawarkan dalam melakukan sesuatu. Kelebihan yang dimiliki oleh Python adalah sekumpulan operasi yang membuat pemrosesan *list* menjadi lebih mudah. Operasi-operasi tersebut akan dibahas di subbab-subbab berikut ini.

➔ Penambahan Elemen

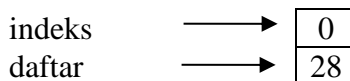
Bagian sebelumnya telah menjelaskan bahwa kita dapat membuat sebuah *list* dengan memberi beberapa angka dalam sebuah urutan sebagai inisialisasi awalnya. Namun, kita juga sering kali tidak mengetahui angka-angka yang akan dimasukkan dalam *list* ketika membuat sebuah *list*. Oleh karena itu, kita dapat membuat sebuah *list* kosong dan menambahkan elemen-elemen yang diperlukan setelahnya. Pertama, kita dapat membuat *list* kosong dengan menggunakan perintah sebagai berikut ini.

```
daftar = []
```

Sebuah elemen dapat ditambahkan pada akhir dari sebuah *list* dengan metode *append* seperti berikut ini.

```
daftar.append(28)
```

Setelah proses penambahan maka *list* tersebut menjadi seperti yang terlihat pada Gambar 5.5.



Gambar 5.5. Penambahan satu elemen list

Ukuran dan panjang dari *list* akan bertambah ketika kita menggunakan metode *append*. Beberapa angka dapat ditambahkan pada *list* tersebut:

```
daftar.append(28)
```

```
daftar.append(7)
daftar.append(14)
```

Setelah proses penambahan-penambahan tersebut maka *list* tersebut menjadi seperti yang terlihat pada Gambar 5.6.

indeks	→	0	1	2
daftar	→	28	7	14

Gambar 5.6. Penambahan 2 elemen berikutnya pada *list*

➡ Penyisipan Elemen

Kita telah mempelajari penambahan sebuah elemen ke dalam *list* menggunakan metode `append`. Jika urutan elemen bukan merupakan masalah yang berarti, penambahan elemen menggunakan `append` sudah cukup dilakukan. Namun, urutan elemen baru yang dimasukkan terkadang dapat menjadi penting. Elemen baru tersebut dapat dimasukkan ke dalam suatu urutan tertentu di dalam *list*. Sebagai contoh, ada *list* sebagai berikut.

```
daftar = [1, 2, 3, 4, 5]
```

Setelah *list* tersebut terisi oleh lima elemen itu (1, 2, 3, 4, 5) maka susunan dari *list* tersebut dapat terlihat pada Gambar 5.7.

indeks	→	0	1	2	3	4
daftar	→	1	2	3	4	5

Gambar 5.7. Kondisi list awal sebelum memasukkan elemen

Anggap kita ingin menyisipkan angka 2.5 kepada *list* pada urutan ke-2 setelah angka 2. Metode yang diberikan untuk memasukkan angka tersebut adalah sebagai berikut. Ilustrasi lokasi penyisipan angka pada *list* tersebut dapat dilihat pada Gambar 5.8.

```
daftar.insert(2, 2.5)
```

Lokasi penyisipan angka

indeks	→	0	1	2	3	4
daftar	→	1	2	3	4	5

Gambar 5.8. Lokasi penyisipan angka pada list

Semua elemen yang berada pada posisi ke-2 dan seterusnya akan bergeser posisi 1 langkah ke belakang. Setelah setiap panggilan ke perintah *insert*. Susunan setiap elemen di dalam *list* tersebut akan menjadi seperti pada Gambar 5.9.

indeks	→	0	1	2	3	4	5
daftar	→	1	2	2.5	3	4	5

Gambar 5.9. Kondisi list setelah penyisipan angka

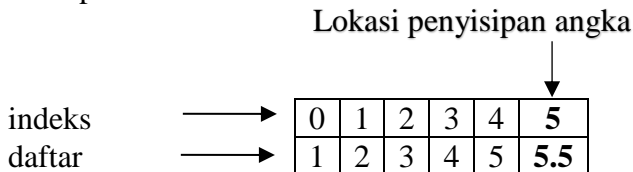
Indeks dari elemen baru yang akan disisipkan haruslah berada di antara 0 sampai jumlah elemen yang ada pada *list*. Sebagai contoh, *list* yang memiliki panjang 5, indeks yang valid untuk proses penyisipan adalah 0, 1, 2, 3, 4, dan 5. Elemen akan disisipkan sebelum elemen pada indeks yang diberikan, kecuali jika indeks elemen sama dengan jumlah elemen yang ada pada *list* tersebut sehingga proses tersebut merupakan proses penambahan elemen. Contoh perintah penambahan elemen menggunakan metode *insert* adalah sebagai berikut.

```
daftar.insert(5, 5.5)
```

Perintah tersebut pada dasarnya adalah sama dengan metode berikut ini.

```
daftar.append(5.5)
```

Pada dasarnya, dua metode tersebut adalah sama, yakni memasukkan sebuah angka pada bagian akhir dari *list*. Gambar 5.10 menunjukkan lokasi penyisipan angka 5.5 pada *list* tersebut



Gambar 5.10. Lokasi penyisipan angka pada bagian akhir list



Pencarian Elemen

Anda dapat mencari keberadaan dari elemen pada sebuah *list* menggunakan operator `in`. Contoh penggunaan operator tersebut adalah sebagai berikut ini.

```
genap = [2, 4, 6, 4, 10]
if 4 in genap :
    print("angka genap")
```

Dari program tersebut, kita mencari angka 4 di dalam *list* yang dirujuk oleh variabel `genap`. Jika angka tersebut berada di dalam *list* maka program akan mencetak tulisan “angka genap”. Oleh karena angka 4 tersebut termasuk salah satu angka di dalam *list* maka program tersebut akan mencetak “angka genap” pada layar.

Sering kali, kita ingin mengetahui posisi dari elemen yang dicari. Kita dapat mencari elemen yang diinginkan menggunakan metode `index`. Indeks tersebut akan dimunculkan berdasarkan elemen yang pertama kali ditemukan pada suatu *list*. Pencarian elemen dalam sebuah *list* dapat dilihat pada contoh berikut ini.

```
genap = [2, 4, 6, 4, 10]
n = genap.index(4) #n diisi dengan 1
print(n)
```

Program tersebut akan menampilkan isi dari variabel `n`. Variabel `n` berisi nomor indeks dari angka 4 di dalam *list*. Luaran dari program tersebut adalah 1 karena angka 4 yang pertama ditemukan berada pada indeks yang ke-1.

Jika angka yang dicari ada dua atau lebih, kita dapat menemukan posisi dari angka yang ke-2 atau lebih dengan menggunakan metode `index` juga. Kita dapat mencari posisi dari angka 4 yang berikutnya dengan menggunakan contoh sebagai berikut.

```
genap = [2, 4, 6, 4, 10]
n = genap.index(4)
m = genap.index(4,n+1)
print(l)
```

Kita dapat menggunakan lokasi hasil pencarian sebelumnya (`n`) untuk mencari lokasi pencarian angka setelahnya. Dalam hal ini variabel `n` akan diisi dengan angka `m` dan variabel `m` akan diisi dengan indeks lokasi angka 4 yang berikutnya, yakni indeks ke-3.

Program tersebut akan menghasilkan nilai kesalahan ketika nilai yang dicari tidak ditemukan di dalam *list*. Oleh karena itu, kita dapat menggunakan proses pengujian menggunakan operator `in` sebelum memanggil metode `index`. Pengujian tersebut dapat dilihat pada contoh program berikut ini.

```
genap = [2, 4, 6, 4, 10]
```


List, String, dan Files

```
if 4 in genap :  
    n = genap.index(4)  
else :  
    n = -1  
print(n)
```

Program itu akan menghasilkan indeks letak angka 4 pertama kali ditemukan dan akan mengembalikan nilai -1 ketika angka 4 tidak ditemukan di dalam *list*.

Penghapusan Elemen

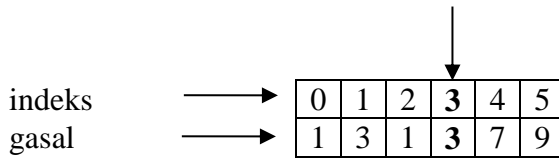
Metode `pop` digunakan untuk menghapus elemen pada suatu indeks lokasi yang diberikan. Sebagai contoh, ada sebuah *list* yang berisi angka-angka berikut ini.

```
gasal = [1, 3, 5, 3, 7, 9]
```

Penghapusan elemen pada posisi indeks ke-3 dapat dilakukan dengan menggunakan metode `gasal.pop(3)`.

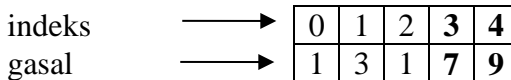
Semua elemen yang mengikuti elemen yang dihapus akan bergeser maju ke depan dan ukuran *list* akan berkurang 1 (seperti yang terlihat pada Gambar 5.11). Indeks yang dipakai dalam metode `pop` harus berada pada nilai yang valid.

Lokasi penghapusan elemen



indeks	→	0	1	2	3	4	5
gasal	→	1	3	1	3	7	9

List tersebut menjadi



indeks	→	0	1	2	3	4
gasal	→	1	3	1	7	9

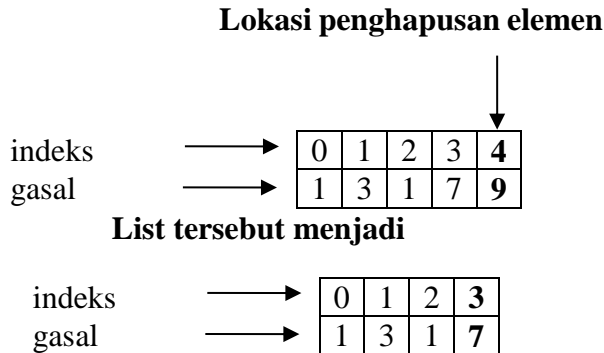
Gambar 5.11. Contoh penghapusan elemen pada bagian tengah list

Pada Gambar 5.11, kita dapat melihat jumlah elemen pada *list* menjadi 5 dengan perubahan indeks elemen 7 dari 4 ke 3 serta indeks elemen 9 (elemen yang terakhir) dari 5 ke 4. Kita juga dapat memanggil elemen yang akan dihapus kemudian menghapusnya dengan metode berikut ini.

```
print("elemen yang dihapus adalah ",
      gasal.pop(3))
```

Kita juga dapat memanggil metode `pop` tanpa menggunakan sebuah argumen. Hal ini akan berarti kita ingin menghapus elemen terakhir dari sebuah *list*. Sebagai contoh, kita memberikan metode `gasal.pop()` pada kondisi *list* terakhir yang dirujuk oleh variabel `gasal`. Pengurangan jumlah elemen pada *list* tersebut akan mempengaruhi indeks maksimum dari *list* tersebut. Indeks terakhir atau

maksimum dari *list* yang baru akan berkurang 1 dari *list* sebelumnya. Dalam hal ini, jika metode `gasal.pop()` tersebut dijalankan maka isi dari *list* akan menjadi seperti yang terlihat pada Gambar 5.12.

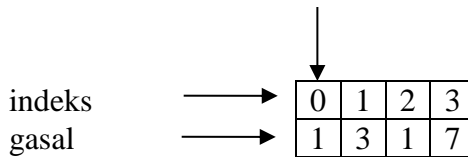


Gambar 5.12. Contoh penghapusan elemen pada bagian tengah list

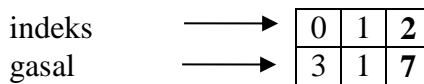
Selain metode `pop()`, python juga menyediakan metode `remove()` untuk menghapus elemen dari sebuah *list*. Perbedaannya adalah metode `pop()` menghapus elemen dari sebuah *list* menggunakan argumen posisi dari elemen tersebut sedangkan metode `remove()` menggunakan argumen nilai dari *list* yang ingin dihapus. Sebagai contoh, kita ingin menghapus angka 5 dari *list* yang dirujuk oleh variabel `gasal` tersebut. Kita dapat menggunakan metode `remove(1)` untuk menghapusnya. Kita tanpa perlu mengetahui lokasi dari angka 1 tersebut berada. Namun, ketika suatu *list* memiliki dua atau lebih nilai

atau angka yang ingin dihapus. Angka 1 terdapat pada dua indeks dalam *list* tersebut (indeks ke-0 dan indeks ke-1). Ketika metode `remove(1)` dieksekusi, angka 1 yang terhapus adalah angka 1 yang berada pada indeks yang lebih awal. Dalam kasus ini, angka 1 pada indeks ke-0 yang akan dihapus. Ilustrasi lebih detail dari penghapusan nilai menggunakan metode `remove()` dapat dilihat pada Gambar 5.13.

Lokasi penghapusan elemen



List tersebut menjadi



Gambar 5.13. Contoh penghapusan elemen pada bagian awal list

Metode `remove()` berbeda dengan metode `pop()`. Dalam hal ini, kita tidak dapat mengakses elemen yang akan dihapus kemudian menghapusnya seperti yang dapat dilakukan menggunakan metode `pop()`. Kita juga dapat melakukan pengujian terlebih dahulu mengenai keberadaan sebuah elemen yang ingin dihapus. Jika elemen yang ingin dihapus berada pada *list*, kita dapat menghapusnya. Agar dapat

melakukannya, kita dapat menggunakan cara berikut ini.

```
elemen = 1
if elemen in gasal :
    gasal.remove(elemen)
```



Penggabungan *List*

Kita dapat menggabungkan dua buah *list* menjadi sebuah *list* yang baru. *List* yang baru tersebut merupakan gabungan dari elemen-elemen pada *list* pertama dan diikuti oleh elemen-elemen dari *list* yang kedua. Sebagai contoh, kita memiliki dua *list* sebagai berikut ini.

```
gasal = [1, 3, 5, 7, 9]
genap = [2, 4, 6, 8, 10]
```

Kita dapat menggabungkan dua *list* tersebut ke dalam sebuah *list* yang baru dengan menggunakan operator plus (+).

```
bilangan = gasal + genap
#bilangan berisi [1, 3, 5, 7, 9, 2, 4,
6, 8, 10]
```

Jika kita ingin menggandakan *list* yang sama secara beberapa kali, kita dapat menggunakan operator kali (*). Sebagai contoh,

```
bilanganAsli = [1, 2, 3] * 2
```

```
#list-nya berisi dengan [1, 2, 3, 1,
2, 3]
```

➡ Pengujian Kesamaan

Kita dapat menggunakan operator ‘=’ untuk membandingkan setiap elemen dari dua buah *list* dalam setiap urutan indeks. Hasil dari perbandingan tersebut akan menghasilkan `True` jika setiap indeks dari kedua *list* tersebut berisi elemen-elemen yang sama. Jika tidak, hasil yang dikeluarkan adalah `False`. Sebagai contoh, `[2, 5, 8] = [2, 5, 8]` adalah `True` sedangkan `[2, 5, 8] = [8, 4, 2]` adalah `False`

Kebalikan dari operator ‘=’ adalah ‘!=’. Arti dari operator tersebut tidak lagi sama dengan melainkan “tidak sama dengan”. Sebagai contoh, `[2, 5, 8] != [8, 4, 2]` adalah `True`

➡ Penjumlahan, Maksimal, Minimal, dan Penyortiran

Jika kita memiliki sebuah *list* yang berisi beberapa angka, fungsi `sum` akan menjumlahkan semua nilai yang ada di dalam *list* seperti pada contoh berikut ini.

```
sum(4, 8, 2, 6) #hasilnya 20
```

List, String, dan Files

Dalam sebuah *list*, kita juga dapat mencari nilai maksimum dan minimum-nya dengan menggunakan perintah `max` dan `min`. Berikut ini merupakan contoh penggunaan kedua perintah tersebut.

```
max(4, 8, 2, 6) #hasilnya 8
min(4, 8, 2, 6) #hasilnya 2
```

Metode `sort` digunakan untuk menyortir nilai-nilai yang ada dalam *list*.

```
angka = [4, 8, 2, 6]
angka.sort() #nilainya menjadi [2, 4,
6, 8]
```

Penggandaan List

Seperti yang telah dibahas pada subbab penggandaan rujukan *list* sebelumnya, sebuah *list* dapat dirujuk oleh satu atau lebih variabel. Variabel *list* tersebut hanya berisikan alamat dari *list* tersebut sehingga ketika kita menggandakan alamat tersebut maka kita akan memperoleh variabel *list* lain yang tetap merujuk ke *list* yang sama. Kita dapat memodifikasi *list* tersebut menggunakan dua alamat yang telah dibentuk.

Sering kali, kita menginginkan untuk membuat duplikat dari sebuah *list*. Sebuah *list* yang baru memiliki elemen-elemen yang sama dalam

urutan yang sama dengan *list* sebelumnya. Kita dapat menggunakan perintah *list* untuk melakukannya.

```
angka = [3, 2.5, 1, 4, 2, 3.5]
nilai = list(angka)
```

Sekarang, variabel `angka` dan `nilai` merujuk pada *list* yang berbeda (seperti yang terlihat pada Gambar 5.14). Setelah proses penggandaan, kedua *list* memiliki isi yang sama. Namun, kita dapat memodifikasi salah satu *list* tanpa mempengaruhi *list* yang lain.

indeks	→	0	1	2	3	4	5
angka	→	3	2.5	1	4	2	3.5
dan							
indeks	→	0	1	2	3	4	5
Nilai	→	3	2.5	1	4	2	3.5

Gambar 5.14. Contoh penggandaan *list*

Pembacaan Masukan

Hal ini merupakan paling umum dilakukan ketika kita bekerja menggunakan *list*. Kita dapat membaca masukan dari pengguna dan menyimpannya untuk proses selanjutnya. Kita memulai dengan sebuah *list* kosong, lalu setiap masukan pengguna dibaca untuk

List, String, dan Files

ditambahkan nilainya ke dalam *list*. Contoh program untuk hal tersebut adalah sebagai berikut.

```
daftar = []
print("silakan masukkan nilai dan tekan X
untuk keluar!")
masukan = input("")
while masukan.upper() != 'X' :
    daftar.append(float(masukan))
    masukan = input("")
print(daftar)
```

Program akan selalu meminta masukan pengguna dan menambahkannya ke dalam *list* sampai pengguna memasukkan atau menekan tombol 'X'. Hasil dari eksekusi program tersebut adalah sebagai berikut ini.

```
silakan masukkan nilai dan tekan X untuk
keluar!
3.5
2.6
2
6.4
5
X
[3.5, 2.6, 2.0, 6.4, 5.0]
```

Pembuatan Tabel

Kumpulan nilai tidak jarang juga disimpan dalam bentuk dua dimensi data. Hal ini sering terlihat pada aplikasi saintifik atau finansial. Sebuah susunan yang berisi beberapa baris dan kolom disebut tabel atau matriks. Pada bagian ini, kita akan mengeksplorasi cara menyimpan contoh data dalam sebuah tabel. Sebagai contoh berikut ini, kita dapat menyimpan komponen-komponen nilai dari seorang murid dalam sebuah tabel.

Nama Murid	Nilai Presensi	Nilai Tugas	Nilai USS	Nilai US
Ibnu	90	85	75	80
Adi	75	80	85	80
Febry	80	90	85	70
Eko	85	70	90	90
Kurniawan	85	75	90	85
Hari	70	85	80	75

Python tidak memiliki sebuah tipe data untuk membuat tabel. Namun, struktur tabel dua dimensi dapat dibentuk dengan menggunakan *list*. Program berikut ini akan menunjukkan kita cara membuat tabel dengan 6 baris dan 4 kolom, yang sesuai dengan daftar nilai dari 6 murid.

```
murid = 6
nilai = 4
daftar = [
```

List, String, dan Files

```
[90, 85, 75, 80],  
[75, 80, 85, 80],  
[80, 90, 85, 70],  
[85, 70, 90, 90],  
[85, 75, 90, 85],  
[70, 85, 80, 75]  
]
```

Contoh program berikut akan membuat sebuah *list* yang dimana masing-masing elemennya merupakan sebuah *list* juga (terlihat di Gambar 5.15).

Kita dapat memanfaatkan perulangan untuk mengisi nilai-nilai yang ada dalam tabel sesuai dengan aturan atau kaidah tertentu. Hal ini akan lebih efisien daripada kita memasukkan satu per satu elemen ke dalam *list*.

List Baris		List Kolom			
Indeks Baris	Isi Baris	Indeks Kolom			
		0	1	2	3
0		90	85	75	80
1		75	80	85	80
2		80	90	85	70
3		85	70	90	90
4		85	75	90	85
5		70	85	80	75

Gambar 5.15. Contoh pembuatan tabel menggunakan *list*

Berikut ini merupakan contoh program untuk melakukan hal tersebut.

```
tabel = []
nbaris = 5
nkolom = 20
for i in range(nbaris) :
    baris = [i+1] * nkolom
    tabel.append(baris)
print(tabel)
```

Pertama, kita menyiapkan *list* kosong yang dirujuk oleh variabel *tabel*. Lalu, kita membuat *list* baru menggunakan perulangan (menggunakan jumlah kolom sebagai ukuran dari *list* tersebut). Kemudian, *list* tersebut akan ditambahkan ke dalam setiap baris di dalam tabel. Hasil dari program tersebut adalah sebagai berikut ini.

[illegible]

Pengaksesan Elemen Tabel

Dalam pengaksesan setiap elemen di dalam tabel, kita memerlukan dua nilai indeks yang dipisahkan dengan kurung siku '[' dan ']'. Nilai indeks yang pertama menandakan baris dari tabel sedangkan nilai indeks yang kedua merupakan kolom dari tabel tersebut (seperti yang terlihat pada Gambar 5.16). Sebagai contoh, ada perintah `bilangan = tabel[4][2]`.

Pengaksesan semua elemen dalam tabel dapat memanfaatkan dua perulangan bersarang. Sebagai contoh, kita ingin menampilkan isi dari setiap elemen yang ada dalam tabel yang dirujuk oleh variabel `daftar`, kita dapat menggunakan contoh program berikut ini.

```
for i in range(baris) :
    #Proses baris ke-i
    for j in range(kolom) :
        #Proses baris ke-j
        print("%8d" % daftar[i][j],
end="")
print() #Memulai baris yang baru
```

		Indeks Kolom			
		0	1	2	3
Indeks Baris	0	90	85	75	80
	1	75	80	85	80
	2	80	90	85	70

3	85	70	90	90
4	85	75	90	85
5	70	85	80	75

↑

bilangan = 90

Gambar 5.16. Contoh pengaksesan elemen pada tabel

Pengalokasian Elemen Tetangga

Sering kali, beberapa program yang bekerja menggunakan tabel memerlukan lokasi tetangga dari beberapa elemen di dalamnya. Pada sebuah permainan atau *game*, sering kali lokasi ketetanggaan dari sebuah elemen diperlukan. Gambar 5.17 akan menunjukkan cara menghitung nilai indeks dari tetangga sebuah elemen.

$[i-1][j-1]$	$[i-1][j]$	$[i-1][j+1]$
$[i][j-1]$	$[i][j]$	$[i][j+1]$
$[i+1][j-1]$	$[i+1][j]$	$[i+1][j+1]$

Gambar 5.17. Lokasi ketetanggaan pada sebuah tabel

Sebagai contoh, tetangga kiri dari sebuah elemen `tabel[4][2]` adalah `tabel[4][1]` sedangkan tetangga kanannya adalah `tabel[4][3]`. Tetangga

atas dari elemen tersebut adalah `tabel[3][2]` sedangkan tetangga bawahnya adalah `tabel[5][2]`.

Kita perlu berhati-hati dalam menghitung tetangga dari batas sebuah *list*. Sebagai contoh, `tabel[2][0]` tidak memiliki tetangga kiri karena elemen tersebut telah berada pada batas kiri dari tabel. Dalam menghitung jumlah tetangga kiri dan kanan elemen `tabel[i][j]`. Kita perlu memeriksa lokasi keberadaan elemen tersebut pada batas kiri atau kanan tabel. Kita dapat menggunakan program berikut ini untuk melakukannya.

```
total = 0
if j > 0 :
    total = total + tabel[i][j-1]
if j < kolom-1 :
    total = total + tabel[i][j+1]
```

Penghitungan Jumlah Baris dan Kolom

Hal yang umum dalam pemanfaatan tabel adalah penghitungan jumlah baris dan kolom di dalam tabel tersebut. Dalam contoh daftar nilai, jumlah baris berarti merepresentasikan jumlah nilai yang diperoleh oleh seorang anak. Dalam penghitungan total baris, kita perlu mengunjungi setiap elemen yang ada di

dalam baris tersebut (seperti yang terlihat pada Gambar 5.18).

Jumlah Kolom – 1

↓

		Indeks Kolom			
		0	1	2	3
Indeks Baris	0	[0][0]]	[0][1]]	[0][2]]	[0][3]]
	Penjumlahan baris ke-1	[1][0]]	[1][1]]	[1][2]]	[1][3]]
	2	[2][0]]	[2][1]]	[2][2]]	[2][3]]
	3	[3][0]]	[3][1]]	[3][2]]	[3][3]]
	4	[4][0]]	[4][1]]	[4][2]]	[4][3]]
	5	[5][0]]	[5][1]]	[5][2]]	[5][3]]

Gambar 5.18. Penjumlahan nilai elemen-elemen pada sebuah baris

Seperti yang dapat kita lihat, kita perlu menghitung jumlah dari `tabel[i][j]`, dimana `j` berkisar antara 0 sampai kolom-1 (dalam kasus ini adalah 3). Kode program perulangan berikut ini diperlukan untuk menghitung penjumlahan baris tersebut tersebut.

List, String, dan Files

```
total = 0
for j in range(kolom) :
    total = total + tabel[i][j]
```

Penjumlahan kolom juga mirip dengan penjumlahan baris. Ilustrasi penjumlahan kolom juga dapat dilihat pada Gambar 5.19. Dalam kasus ini, penghitungan kolom dimaksudkan untuk mencari nilai total dari setiap elemen nilai untuk semua murid. Penghitungan tersebut dapat mengikuti contoh program berikut ini.

```
total = 0
for i in range(baris) :
    total = total + tabel[i][j]
```

		Indeks Kolom			
		0	1	2	3
Indeks Baris	0	[0][0]	[0][1]	[0][2]	[0][3]
	1	[1][0]	[1][1]	[1][2]	[1][3]
	2	[2][0]	[2][1]	[2][2]	[2][3]
	3	[3][0]	[3][1]	[3][2]	[3][3]
	4	[4][0]	[4][1]	[4][2]	[4][3]
	5	[5][0]	[5][1]	[5][2]	[5][3]

Jumlah Baris – 1

Gambar 5.19. Penjumlahan nilai elemen-elemen pada sebuah kolom

String

String merupakan kumpulan dari dua atau lebih karakter. Kumpulan karakter tersebut mencakup huruf, angka, dan tanda-tanda khusus yang ada pada *keyboard*, termasuk juga karakter-karakter khusus yang digunakan dalam berbagai bahasa lain di dunia (bahasa Arab, Cina, Jepang, Rusia, dan Yunani). Dalam Python, *string* direpresentasikan sebagai urutan karakter yang berada di antara tanda kutip tunggal atau ganda (' dan ' atau " dan ") dan dapat dirujuk oleh sebuah variabel. String tidak dapat diubah, sekali didefinisikan, string tidak dapat dimodifikasi. Perintah berikut ini mendefinisikan sebuah string yang dirujuk oleh sebuah variabel `strku`.

```
strku = "Universitas Negeri"
```

Salah satu kegiatan yang paling umum digunakan adalah penggabungan string. Kegiatan tersebut menggabungkan dua atau lebih string menjadi sebuah string yang lebih panjang. Penggabungan string tersebut dapat menggunakan sebuah operator plus (+). Perintah berikut ini mendefinisikan cara penggabungan dua string menjadi sebuah string baru yang dirujuk oleh variabel `strgab`. Pemanggilan

terhadap variabel `strgab` akan menghasilkan sebuah string baru “Universitas Negeri Surabaya”.

```
strgab = strku + " Surabaya"
```

Fungsi `len` digunakan untuk mengukur panjang dari sebuah string, yang berarti menghitung jumlah karakter yang ada pada sebuah string. Perintah berikut ini merupakan contoh penggunaan fungsi `len` untuk menghitung jumlah karakter dalam sebuah string yang dirujuk oleh variabel `strgab`. Jumlah karakter tersebut akan disimpan oleh variabel `pjgstr`. Isi dari variabel `pjgstr` adalah 27. Hal ini menandakan panjang karakter dari string `strgab` adalah 27 karakter (termasuk 2 karakter spasi di dalamnya).

```
pjgstr = len(strgab)
```

Proses *indexing* diperlukan untuk mengakses sebagian karakter dari sebuah string. Nilai indeks dimulai pada nol dan diapit oleh tanda kurung siku '[' dan ']'. Perintah berikut ini akan menampilkan karakter dengan indeks ke-3 dari sebuah `strgab`. Karakter yang dimaksud adalah 'v',

```
print(strgab[3])
```

Pemotongan string merupakan salah satu kegiatan yang juga sering dilakukan terhadap sebuah string. Dalam melakukan kegiatan tersebut, kita akan membutuhkan sebuah operator pemisah (:) pada

jangkauan indeks dimana beberapa karakter tersebut dipotong. Dalam contoh berikut ini, perintah pemotongan beberapa karakter atau substring dari sebuah string `strgab` dilakukan. Karakter pada indeks ke-3 sampai indeks ke-8 akan dipisahkan dan dirujuk oleh variabel `strpsh`. Isi dari variabel tersebut adalah sebuah substring “versi”.

```
strpsh = strgab[3:8]
```

Contoh perintah berikut ini akan menjelaskan cara penggabungan sebuah substring pada indeks ke-0 sampai indeks ke-19 dengan sebuah string baru “Jakarta”. Substring yang dihasilkan dari proses pemotongan tersebut adalah “Universitas Negeri”. Substring tersebut akan digabung dengan string “Jakarta” sehingga membentuk substring baru “Universitas Negeri Jakarta” yang dirujuk oleh variabel `strbaru`.

```
strbaru = strgab[0:19] + "Jakarta"
```

Operator keanggotaan `in` digunakan untuk memeriksa keberadaan sebuah karakter atau substring di dalam sebuah string lain serta akan mengembalikan hasil `True` atau `False`. Hasil `True` akan diperoleh ketika sebuah karakter atau substring tersebut berada di dalam sebuah string yang diperiksa sedangkan hasil `False` akan dikembalikan ketika karakter atau substring tersebut tidak ada di dalam sebuah string

yang diperiksa. Variabel `cekstr` akan merujuk hasil pengembalian dari pengujian keberadaan substring “Jak” di dalam string yang dirujuk oleh variabel `strbaru`. Setelah pemeriksaan dilakukan, ternyata substring “Jak” memang berada di dalam string “Universitas Negeri Jakarta” yang dirujuk oleh variabel `strbaru` sehingga hasil dari pemeriksaan tersebut adalah `True`. Nilai `True` inilah yang disimpan pada variabel `cekstr`.

```
cekstr = "Jak" in strbaru
```

Beberapa karakter dalam sebuah string merupakan sebuah *escape character* yang ditandai dengan sebuah *backslash* (`\`). Contohnya adalah sebuah karakter *newline* merupakan bagian dari sebuah string. Hal ini menandakan adanya perpindahan baris dari sebuah substring ke substring yang lain pada string tersebut. Perintah berikut ini akan memperjelas penjelasan mengenai hal tersebut. Dalam perintah ini, penulisan pesan akan terdiri dari dua baris. Baris pertama berisi substring “Silakan memulai program sekarang” sedang baris kedua berisi substring “Klik pada ikon”.

```
print("Silakan      memulai      program  
sekarang\nKlik pada ikon")
```

Sebuah string ditandai dengan adanya kata atau kalimat yang diapit oleh tanda kutip ganda (`"`).

Namun, satu atau lebih tanda kutip ganda tersebut juga dapat menjadi bagian dari string. Kita dapat menggunakan *backslash* seperti pada contoh berikut ini.

```
print("Kami mengucapkan \"selamat
datang\" dan \"terima kasih\" pada semua
pihak")
```

Sesuai contoh, program akan menampilkan tulisan “Kami mengucapkan "selamat datang" dan "terima kasih" pada semua pihak” Python menyediakan beberapa metode string dan bentuk umum untuk memanggil metode-metode tersebut antara lain:

```
nama_string.nama_metode(argument)
```

Sebagai contoh, metode `find` digunakan untuk mencari nilai indeks pada string `strbaru` ketika substring “Negeri” ada di dalamnya. Hasil yang dikembalikan merupakan nilai indeks dari substring “Negeri” ditemukan, yakni indeks ke-12.

```
strbaru.find("Negeri")
```

Metode string lain yang sangat berguna adalah `isdigit`. Metode ini digunakan untuk menguji apakah semua karakter dalam sebuah string merupakan kumpulan angka. Luaran dari metode ini ada dua, yakni `True` dan `False` bergantung

komponen-komponen angka penyusun suatu string yang diperiksa tersebut. Contoh program menggunakan metode `isdigit` adalah sebagai berikut ini.

```
strnomor = '12345'  
  
strnomor.isdigit()
```

Hasil dari program tersebut adalah `True` karena seluruh komponen penyusun string `strnomor` adalah angka (1, 2, 3, 4, dan 5). Hal ini akan berbeda (menghasilkan nilai `False`) ketika ada salah satu atau semua komponen penyusun dari sebuah string yang diperiksa merupakan bukan angka.

Files



Program pengolah kata menginspirasi dari operasi-operasi yang ada dalam subbab *files* berikut ini. Fitur-fitur kritis dari program pengolah kata adalah menyimpan dan mengembalikan dokumen *files* pada sebuah *storage*. Dalam sesi ini, kita akan membahas file masukan dan luaran, yang juga berkaitan dengan pengolahan string.

Pemrosesan File

Pemrosesan *file* memiliki cara yang berbeda untuk setiap bahasa pemrograman. Namun, hampir semua bahasa pemrograman memiliki dasar konsep manipulasi *file* yang sama. Pertama, kita membutuhkan beberapa cara untuk menghubungkan suatu *file* pada *storage* dengan obyek pada suatu program. Proses ini yang disebut dengan membuka *file*. Setelah *file* dibuka, isi dari *file* tersebut dapat diakses oleh obyek *file* yang terkait. Kedua, kita memerlukan suatu kumpulan operasi yang dapat memanipulasi obyek *file* tersebut. Minimal, kita dapat membaca informasi dari sebuah *file* dan menuliskan informasi baru ke dalamnya. Proses pembacaan dan penulisan untuk *file* teks ini mirip dengan operasi masukan dan keluaran berbasis teks, masukan dan keluaran yang interaktif.

Akhirnya, ketika kita selesai membaca dan menulis pada sebuah *file*, *file* tersebut akan ditutup. Penutupan sebuah *file* memastikan bahwa proses operasi pada *file* dan keterkaitan obyek *file* dengan *file* yang ada di *storage* telah usai. Hal ini ditandai dengan adanya perubahan informasi pada *file* yang ada pada *storage*. Sebagai contoh, jika kita menuliskan informasi pada sebuah obyek *file* maka perubahan tersebut tidak akan sampai pada versi *storage* sampai

file tersebut ditutup. Perubahan yang dilakukan hanya pada data di memori bukan pada *file* di *storage*.

Bekerja dengan *file* pada Python cukup mudah. Langkah pertama adalah membuat obyek *file* yang berhubungan dengan *file* pada *disk* dengan menggunakan fungsi `open`. Pada umumnya, obyek *file* akan dirujuk oleh sebuah variabel.

```
<variabel> = open(<nama>, <mode>)
```

Nama di sini merupakan nama *file* yang ada pada *storage*. Parameter `mode` adalah karakter ‘r’ atau ‘w’ tergantung operasi *file* yang kita gunakan (membaca atau menulis). Sebagai contoh, kita ingin membuka *file* bernama “angka.txt” untuk membaca konten di dalam *file* tersebut. Kita dapat menggunakan perintah sebagai berikut ini.

```
baca = open("tulisan.txt", 'r')
```

Dalam hal ini, kita dapat menggunakan obyek *file* `baca` untuk membaca konten dari `tulisan.txt` dari *storage*.

Python menyediakan tiga operasi yang terhubung dengan pembacaan informasi dari sebuah *file*, yakni:

1. `<file>.read()` mengembalikan semua konten yang ada pada *file* sebagai sebuah

string tunggal (dapat juga terdiri dari beberapa baris).

2. `<file>.readline()` mengembalikan baris berikutnya pada sebuah *file*. Semua teks akan dikembalikan sampai karakter pindah baris berikutnya.
3. `<file>.readlines()` mengembalikan sebuah daftar baris pada sebuah *file*. Setiap *item* di daftar merupakan isi dari satu baris termasuk karakter pindah baris pada bagian akhir baris.

Berikut ini merupakan contoh program yang menampilkan isi dari sebuah *file* ke layar menggunakan operasi `read`.

```
nama_file = input("Nama file = ")
file_masukan = open(nama_file, 'r')
data = file_masukan.read()
print(data)
```

Pertama, program meminta pengguna untuk memasukkan sebuah `nama_file` dan membuka isi dari *file* tersebut melalui variabel `file_masukan`. Kita dapat menggunakan sembarang nama untuk variabel tersebut. Kita menggunakan `file_masukan` tersebut untuk menekankan kegunaan dari *file* tersebut adalah sebagai masukan. Semua isi dari *file* dibaca sebagai satu string besar dan disimpan di dalam variabel `data`.

List, String, dan Files

Perintah `print` memungkinkan isi dari variabel data untuk ditampilkan.

Operasi `readline` dapat digunakan untuk membaca baris berikutnya dari sebuah *file*. Beberapa panggilan berurutan ke `readline` mendapatkan akan mendapatkan beberapa baris berurutan dari *file*. Secara analogi, pembacaan masukan pengguna secara berulang per karakter sampai pengguna menekan tombol <Enter>. Hal yang perlu diingat ketika menggunakan metode `readline` adalah string yang dikembalikan oleh `readline` akan selalu berakhir dengan karakter pindah baris sedangkan pembacaan masukan menggunakan metode `input` akan mengabaikan karakter tersebut.

Contoh berikut ini merupakan kumpulan kode program yang mengampilkan tiga baris pertama dari sebuah *file*:

```
file_masukan = open(banyak_file, 'r')
for i in range(3)
baris =file_masukan.readline()
print(baris[:-1])
```

Perhatikan kumpulan kode program untuk memotong karakter pindah baris pada akhir dari baris tersebut. Hal ini dikarenakan perintah `print` yang secara otomatis melompat ke baris berikutnya (untuk menghasilkan baris yang baru) dan mencetak baris

baru secara eksplisit di akhir dan menempatkan baris kosong tambahan dari luaran yang berada di antara setiap baris pada *file* tersebut. Sebagai alternatif, kita juga dapat mencetak seluruh baris, tetapi tidak menambahkan karakter pindah baris itu secara otomatis.

```
print(line, end = "")
```

Satu cara untuk mengulang pembacaan seluruh isi *file* adalah membaca semua *file* menggunakan `readlines` dan mengulangnya melalui daftar yang dihasilkan.

```
file_masukan = open(banyak_file, 'r')
for baris in file_masukan.readlines():
    #proses terhadap baris
file_masukan.close()
```

Namun, pendekatan tersebut memiliki kelemahan ketika *file* yang dibaca sangat besar, proses pembacaan yang ke dalam sebuah daftar dapat memakan memori pada RAM yang terlalu besar. Python menyediakan alternatif yang sederhana dalam mengalami hal tersebut. Python memperlakukan *file* tersebut sebagai urutan baris. Jadi, perulangan melalui baris *file* dapat dilakukan seperti berikut ini.

```
file_masukan = open(banyak_file, 'r')
for baris in file_masukan:
    #proses terhadap baris
file_masukan.close()
```

List, String, dan Files

Langkah ini merupakan cara yang sangat berguna untuk memproses baris *file* satu per satu.

Pembukaan sebuah *file* untuk proses penulisan dilakukan untuk mempersiapkan *file* untuk menerima data. Jika tidak ada *file* yang tersedia, sebuah *file* yang baru akan dibuat. Berikut ini merupakan contoh membuka *file* dalam proses penulisan data.

```
file_keluaran = open("dataku.txt",  
'w')
```

Cara termudah untuk menulis informasi ke dalam sebuah *file* teks adalah cukup dengan menggunakan fungsi `print`. Dalam mencetak sebuah *file*, kita hanya perlu menambahkan sebuah parameter untuk menspesifikasikan *file* tersebut.

```
print(..., file = <file_keluaran>
```

Perintah ini selayaknya perintah `print` biasa kecuali hasil yang dikeluarkan dikirimkan ke `file_keluaran` selain ditampilkan ke layar. Hal yang perlu kita ingat, operasi-operasi pada *files* ini dapat dilakukan ketika *file* dan program *python* berada pada lokasi yang sama (satu folder).

Kotak Dialog

Salah satu masalah yang sering muncul dengan program manipulasi *file* adalah dalam mencari tahu cara menspesifikasikan *file* yang akan digunakan. Jika sebuah data *file* berada pada lokasi yang sama (folder atau direktori yang sama) dengan program Python kita maka kita tinggal menuliskan nama dari *file* tersebut dengan tepat dengan tanpa adanya informasi lain. Lalu, Python akan mencari *file* tersebut sesuai dengan lokasi yang ada. Sering kali, dalam mengetahui nama lengkap beserta lokasi file merupakan hal yang susah. Sistem operasi yang paling modern menggunakan nama *file* sesuai dengan bentuk *form* <nama>.<tipe> dimana tipe merepresentasikan ekstensi dari sebuah *file*. Sebagai contoh, data pengguna disimpan pada *file* “pengguna.txt” dimana ekstensi “.txt” mengindikasikan sebuah *file* teks.

Situasi akan menjadi lebih susah ketika mencari *file* yang berada pada suatu tempat. Program pemrosesan *file* dapat digunakan pada *file* yang disimpan di suatu lokasi memori. Kita harus menspesifikasikan *path* yang lengkap sebagai lokasi *file* dalam sistem komputer pengguna. Bentuk yang pasti dari sebuah *path* dapat membedakan sistem operasi yang digunakan. Dalam sebuah sistem Windows, nama *file* lengkap dengan *path*-nya dapat dilihat sebagai berikut.

List, String, dan Files

```
C:/Users/Ibnu/Documents/Python_Programs/pengguna.txt
```

Namun, banyak pengguna yang masih belum mengetahui *path* lengkap beserta nama *file* untuk setiap *file* yang digunakan.

Solusi untuk permasalahan ini adalah memungkinkan pengguna untuk mencari *path* dari sebuah *file* secara visual. Oleh karena itu, kotak dialog (sebuah jendela khusus untuk interaksi pengguna) dapat mengarahkan pengguna ke direktori atau *file* tertentu. Melalui kotak dialog ini, pengguna dapat diarahkan untuk mengklik di sekitar *file system* menggunakan *mouse* dan memilih atau mengetikkan nama *file*. Untungnya bagi kita, sebuah pustaka GUI `tkinter` yang disertakan di sebagian besar pada proses instalasi Python standar telah menyediakan beberapa fungsi yang mudah digunakan dalam membuat kotak dialog untuk mendapatkan nama *file* tersebut.

Dalam meminta pengguna agar nama *file* tersebut dibuka, kita dapat menggunakan fungsi `askopenfilename`. Fungsi itu dapat ditemukan di modul `tkinter.filedialog`. Pada bagian atas program, kita perlu menambahkan proses impor terhadap fungsi tersebut.

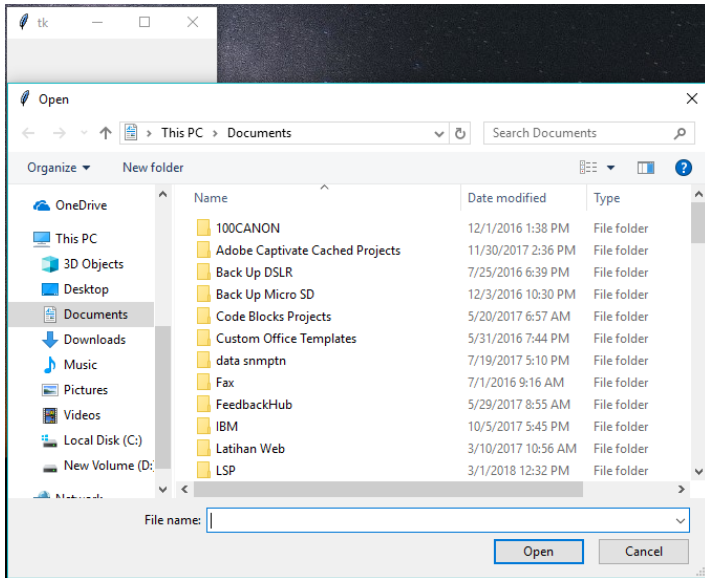
```
from tkinter.filedialog import
askopenfilename
```

Notasi titik pada proses impor tersebut menandakan bahwa paket `tkinter` terdiri dari beberapa modul. Dalam hal ini, kami memilih modul `filedialog` dari `tkinter` dan menggunakan sebuah fungsi `askopenfilename`. Pemanggilan `askopenfilename` akan memunculkan kotak dialog *file* sesuai sistem. Sebagai contoh, untuk mendapatkan nama dari *file* pengguna, kita dapat menggunakan kode sebagai berikut ini.

```
file_masukan = askopenfilename()
```

Hasil dari proses eksekusi kode program tersebut pada sistem operasi Windows adalah seperti yang terlihat pada Gambar 5.20. Kotak dialog memungkinkan tersebut memungkinkan pengguna untuk mengetikkan nama *file* atau hanya memilihnya dengan menggunakan *mouse*. Ketika pengguna mengeklik tombol “Open”, nama *path* lengkap dari *file* dikembalikan sebagai string dan disimpan ke variabel `file_masukan`. Jika pengguna mengklik tombol “Cancel”, fungsi akan mengembalikan sebuah string kosong.

List, String, dan Files



Gambar 5.20. Kotak dialog dari askopenfilename

Paket tkinter pada Python juga menyediakan fungsi analog, `asksaveasfilename`, untuk menyimpan file (seperti pada Gambar 5.21). Penggunaannya sangat mirip dengan pembuatan kotak dialog yang sebelumnya (menggunakan fungsi `askopenfilename`).

```
from tkinter.filedialog import  
asksaveasfilename
```

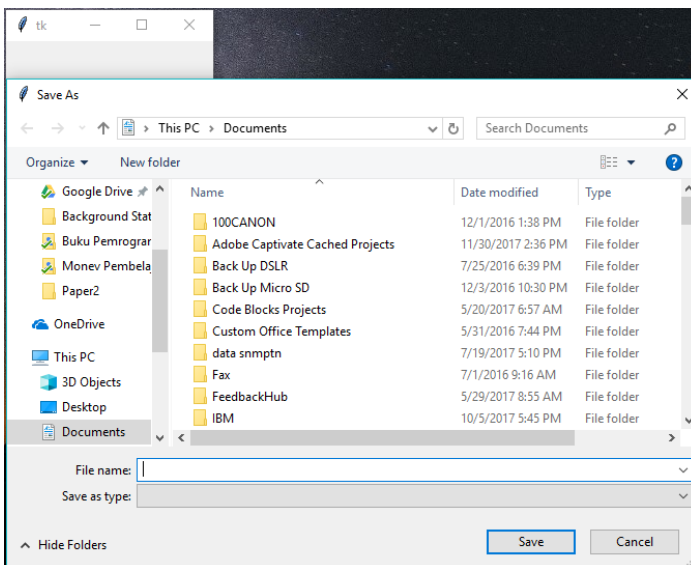
```
...
```

```
file_keluaran = asksaveasfilename()
```

Kita juga dapat mengimpor kedua fungsi tersebut secara sekaligus seperti pada kode program berikut ini.

```
from tkinter.filedialog import
askopenfilename, asksaveasfilename
```

Kedua fungsi ini memiliki parameter ini juga memiliki beberapa parameter opsional sehingga program dapat menyesuaikan kotak dialog yang dihasilkan. Misalnya, kita dapat mengubah judul atau menyarankan sebuah nama *file default*.



Gambar 5.21. Kotak dialog dari *asksaveasfilename*

Kesimpulan

Bab ini membahas mengenai operasi-operasi yang dapat dilakukan pada *list*, *string* dan *file* di pemrograman Python. Pada operasi *list*, kita mempelajari proses penambahan, penyisipan, dan pencarian elemen pada *list*, penggabungan *list*, pengujian kesamaan pada *list*. Pada *list* juga, kita dapat melakukan penjumlahan, pencarian nilai maksimal, nilai minimal, penyortiran serta penggandaan pada *list* tersebut.

Operasi *string* dapat dilakukan pada data yang berisi kumpulan dari beberapa karakter. Kita dapat belajar menggabungkan *string* yang satu dengan yang lain, menghitung panjang karakter dari sebuah *string*, mencari suatu *substring* dari suatu *string* yang lebih besar. Beberapa operasi *string* tersebut dapat menjadi dasar kita dalam belajar mengolah *string* dalam pemrograman Python.

Pemrosesan *file* dan pembuatan kotak dialog dapat memudahkan kita dalam membuat program Python yang memerlukan pengolahan terhadap *file* di dalamnya. Melalui pembelajaran operasi *file* di buku ini, kita dapat membaca dan menulis ke dalam suatu *file* serta kita dapat membuat kotak dialog untuk membuka *file* dan menyimpan data ke dalam suatu *file*

tertentu. Kotak dialog tersebut akan memudahkan pengguna dalam memilih *file* yang akan dibuka dan memilih *file* sebagai tempat penyimpanan data.

BAB VI

FUNGSI

Sebuah program umumnya terbagi menjadi beberapa bagian. Pada Python, bagian-bagian itu berupa modul, *class*, atau fungsi. Sebuah fungsi merupakan modul atau unit dekomposisi yang paling dasar di dalam pemrograman Python. Ketika dipanggil, sebuah fungsi melakukan tugas tertentu dalam suatu program dan dapat menerima data masukan dari fungsi lain; data masukan ini dikenal sebagai argumen. Fungsi ini juga dapat mengembalikan data luaran ketika fungsi tersebut dieksekusi.

Dalam bab ini, kita akan belajar cara merancang dan mengimplementasikan fungsi buatan kita sendiri sehingga kita dapat memecah tugas-tugas yang rumit menjadi beberapa fungsi yang terkait. Bab ini memberikan rincian mengenai pendefinisian dan pemanggilan serta pendekomposisian fungsi. Bab ini juga membahas mekanisme dasar untuk pengiriman data antara dua fungsi atau lebih. Beberapa contoh pemanggilan fungsi matematika *built-in* juga dibahas di dalam bab ini.

Tujuan

- Menjelaskan alasan *programmer* membagi program ke dalam beberapa fungsi yang terkait;
- Mendefinisikan fungsi baru menggunakan Python;
- Menjelaskan rincian pemanggilan fungsi dan *passing* parameter dalam Python;
- Membuat program yang menggunakan fungsi untuk mengurangi duplikasi kode dan meningkatkan modularitas program.

Fungsi dari Fungsi

Sebuah masalah sering kali terasa terlalu besar dan rumit untuk diselesaikan dengan satu unit program. Dalam pemecahan masalah dan desain algoritmik, satu masalah yang besar dapat dipartisi menjadi beberapa masalah kecil agar lebih mudah diselesaikan. Partisi masalah menjadi beberapa bagian yang lebih kecil dikenal dengan proses dekomposisi. Bagian-bagian kecil ini sering dikenal sebagai unit modular, yang mana unit ini jauh lebih mudah untuk

dikembangkan dan dikelola daripada satu unit besar dan utuh semula. Unit-unit modular ini dianggap sebagai blok bangunan untuk membangun algoritma yang lebih besar dan lebih kompleks.

Solusi dari masalah yang besar diperoleh dari perakitan beberapa solusi dari masalah-masalah yang kecil tersebut. Proses pencarian kesalahan atau pengujian terhadap solusi tersebut akan lebih mudah karena ruang lingkup dari solusi tersebut menjadi lebih sempit sehingga bagian *tester* dapat menguji program yang dibuat untuk setiap solusi yang kecil dengan lebih efektif.

Pendefinisian terhadap suatu fungsi perlu dilakukan sebelum fungsi tersebut dipanggil oleh instruksi dalam program Python. Selain memanggil fungsi yang dibuat *programmer*, instruksi dalam program dapat memanggil fungsi *built-in* yang disediakan oleh pustaka *interpreter* Python standar atau modul Python lainnya. Sebuah pustaka merupakan kumpulan definisi fungsi yang terkait dan / atau definisi *class* yang mungkin juga termasuk data.

Pada dasarnya, program yang kita buat hanya terdapat sebuah fungsi, biasanya disebut fungsi utama (*main function*). Kita juga sering menggunakan metode-metode yang termasuk fungsi-fungsi *built-in* di dalam Python (misalnya `print`, `abs`), fungsi dan

Fungsi

metode dari pustaka standar Python (misalnya `math.sqrt`), dan metode dari modul grafis (misalnya `myPoint.getX()`).

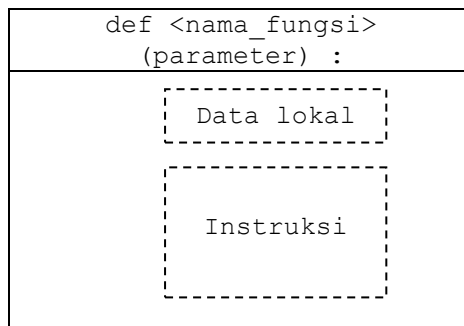
Fungsi merupakan hal yang penting dalam membuat sebuah program yang baik. Fungsi dapat digunakan untuk mereduksi penduplikasian kode dan membuat program lebih mudah dipahami dan dipelihara. Penggunaan fungsi ini juga dapat mendukung proses pengerjaan program secara tim. Setiap orang atau bagian dari tim dapat mengerjakan satu atau dua buah fungsi. Selanjutnya, beberapa fungsi yang telah dikerjakan dapat dikompilasi kembali sebuah program yang utuh.

Pendefinisian Fungsi

Sebuah program Python sering didekomposisi menjadi beberapa modul, *class* atau fungsi. Sebuah fungsi membawa tugas yang spesifik dalam sebuah program. Data dalam sebuah fungsi hanya dapat dikenali oleh fungsi tersebut (tergantung ruang lingkup / *modifier* dari data tersebut). Data lokal dalam sebuah fungsi memiliki waktu aktif yang terbatas. Data tersebut hanya dapat digunakan selama fungsi tersebut dijalankan. Data local tersebut dapat

berupa variabel-variabel yang ada di dalam fungsi tersebut dan bersifat lokal.

Sebuah program Python biasanya terdiri dari fungsi dan instruksi yang memanggil fungsi tersebut. Gambar 6.1 mengilustrasikan struktur umum dari sebuah fungsi dalam bahasa Python.



Gambar 6.1. Struktur umum fungsi dalam Python

Dalam kode program, bentuk sintaksis umum dari pendefinisian fungsi di dalam bahasa pemrograman Python, seperti pada contoh berikut:

```

def    nama_fungsi    (    [parameter-
parameter] ) :
    [ deklarasi variabel lokal ]
    [ instruksi-instruksi ]
  
```

Pendefinisian data lokal dalam fungsi merupakan langkah opsional untuk dilakukan. Instruksi-instruksi di dalam fungsi merupakan tubuh

Fungsi

dari fungsi itu sendiri. Perintah atau instruksi tersebut yang merupakan kegiatan utama di dalam fungsi. Pemanggilan sebuah fungsi pada dasarnya adalah pengeksekusian terhadap instruksi-instruksi atau perintah-perintah yang merupakan isi atau tubuh dari fungsi tersebut.

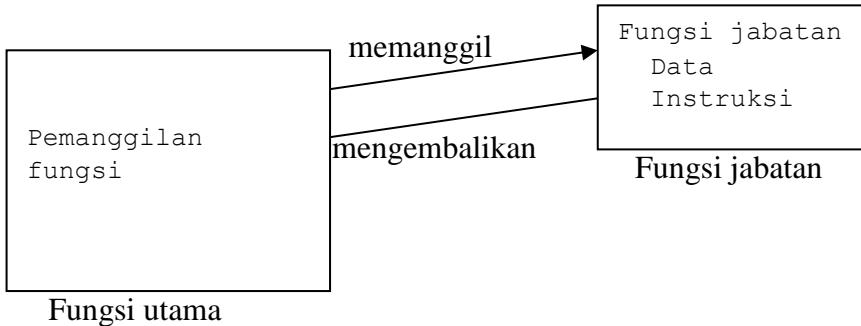
Dokumentasi internal yang relevan dari pendefinisian fungsi dideskripsikan dalam satu atau lebih baris komentar, yang dimulai dengan karakter ("") dan diakhiri dengan (""). Pendefinisian data lokal dalam sebuah fungsi adalah opsional. Instruksi dalam program merupakan tubuh dari fungsi.

Pemanggilan Fungsi



Setelah fungsi didenifisikan, fungsi tersebut dapat dipanggil dengan menuliskan nama fungsi yang dipanggil di dalam instruksi program. Ketika sebuah fungsi dipanggil, terjadi perubahan alur urutan normal pada pengeksekusian program. Pada pemanggilan fungsi tersebut, program akan menjalankan isi atau tubuh dari fungsi tersebut disertai dengan pengiriman parameter yang dibutuhkan dan hasil yang dikeluarkan (jika ada) kemudian program akan mengeksekusi perintah-perintah di bawah

pemanggilan fungsi tersebut. Gambar 6.2 merupakan ilustrasi pemanggilan sebuah fungsi jabatan. Setelah proses eksekusi selesai, fungsi jabatan akan menjalankan perintah-perintah di dalamnya, mengembalikan hasil (jika ada) dan mengeksekusi kembali perintah-perintah yang ada setelah pemanggilan fungsi tersebut. Dalam Python, pemanggilan fungsi dilakukan dengan menuliskan nama fungsi dan diikuti sepasang tanda kurung kosong “()”. Sebagai contoh memanggil fungsi `tampil_pesan` dapat dilakukan dengan menuliskan nama fungsi tersebut sebagai perintahnya, yakni menuliskan perintah `tampil_pesan()`.



Gambar 6.2. Hubungan dua buah fungsi

Kategori Fungsi



Pengiriman data dapat terjadi di antara instruksi pemanggilan fungsi dan fungsi yang dipanggil. Pengiriman data ini dapat terjadi baik dari instruksi yang memanggil fungsi ke fungsi yang dipanggil dan atau fungsi yang dipanggil ke instruksi yang memanggil. Sehubungan dengan pengiriman data tersebut, ada empat kategori dari fungsi tersebut:

1. Fungsi sederhana, yang tidak memungkinkan adanya pengiriman data ketika fungsi tersebut dipanggil. Contohnya adalah fungsi `tampil_pesan`, fungsi ini hanya untuk menampilkan pesan. Jadi tidak ada pengiriman data yang terjadi di dalamnya. Seringkali, fungsi ini dinamakan prosedur.
2. Fungsi yang mengembalikan sebuah nilai ketika fungsi tersebut dieksekusi.
3. Fungsi yang memerlukan satu atau lebih parameter, yang mana parameter-parameter tersebut merupakan data yang ditranfer ke fungsi tersebut.
4. Fungsi yang memungkinkan proses pengiriman data terjadi di dua arah. Fungsi ini memerlukan satu atau lebih parameter dan mengembalikan sebuah nilai ke instruksi yang memanggil fungsi tersebut.

➡ Fungsi Sederhana

Fungsi sederhana ini tidak mengembalikan nilai ke instruksi yang memanggil fungsi. Dalam fungsi ini tidak terjadi pengiriman data dari atau ke fungsi tersebut. Gambar 6.2 menunjukkan sebuah instruksi yang memanggil fungsi jabatan. Setelah proses pemanggilan dilakukan, isi dari fungsi yang dipanggil (fungsi jabatan) dieksekusi lalu kembali menuju ke instruksi yang memanggilnya. Salah satu contoh dari fungsi ini adalah fungsi `tampil_pesan` sebagai berikut ini.

```
def tampil_pesan() :
    """
        Fungsi ini akan menampilkan sebuah
        pesan pada layar
    """
    print("sebuah pesan")
```

Fungsi tersebut merupakan fungsi sederhana yang bertujuan untuk menampilkan sebuah pesan teks pada layar. Fungsi `tampil_pesan` itu tidak memerlukan parameter masukan dan luaran sebagai hasil dari fungsi tersebut.

➡ Pemanggilan Fungsi yang Mengembalikan Nilai

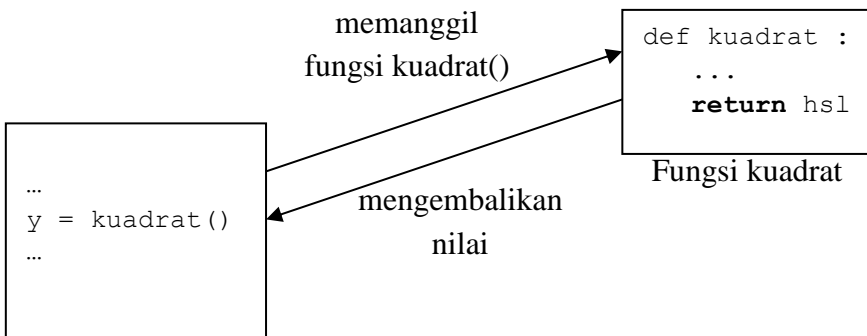
Fungsi yang mengembalikan nilai mengirimkan hasil pengolahan kembali instruksi yang memanggil fungsi

Fungsi

tersebut. Pada umumnya, nilai yang merupakan hasil dari pengolahan tersebut dikembalikan ke instruksi yang memanggil fungsi. Fungsi-fungsi ini dapat dipanggil dengan salah satu dari dua cara berikut ini.

1. Pemanggilan fungsi dalam perintah sederhana;
2. Pemanggilan fungsi dalam perintah sederhana dengan sebuah ekspresi.

Pendefinisian fungsi-fungsi ini harus menyertakan minimal satu pernyataan untuk pengembalian, yang ditulis dengan menggunakan kata kunci **return** yang diikuti oleh sebuah ekspresi. Nilai dalam pernyataan pengembalian dapat berupa nilai-nilai ekspresi yang valid dan yang mengikuti kata kunci **return**. Ekspresi tersebut dapat mencakup konstanta, variabel, atau kombinasi dari kedua hal tersebut.



Gambar 6.3. Pemanggilan fungsi kuadrat

Gambar 6.3 mengilustrasikan proses pemanggilan fungsi kuadrat melalui sebuah instruksi. Nilai pengembalian dari fungsi kuadrat ditujukan kepada instruksi yang memanggil fungsi tersebut. Nilai tersebut akan dimasukkan ke variabel `y`.

Kode program berikut ini menunjukkan penggunaan sebuah fungsi kuadrat yang mengembalikan nilai. Fungsi kuadrat tersebut didefinisikan terlebih dahulu sebelum dipanggil. Fungsi ini mengembalikan nilai kuadrat dari variabel `x` yang diberi nilai 3.14. Instruksi pemanggilan fungsi ini juga terdapat di dalam kode program berikut ini. Setelah proses pemanggilan fungsi, nilai dari variabel `y` juga ditampilkan melalui program ini.

```
def kuadrat() :
    """
    Fungsi ini mengembalikan kuadrat dari
    variabel x dengan nilai 3.14
    """
    x = 3.14
    hsl = x**2
    return hsl

y = kuadrat()
print("Nilai dari y adalah ", y)
```

Ketika program tersebut dieksekusi, tampilan yang dikeluarkan oleh program tersebut adalah “Nilai dari y adalah 9.8596”. Nilai 9.8596 diperoleh dari

3.14^2 atau $3.14 * 3.14$. Hasil dari kuadrat dari nilai 3.14 tersebut akan dirujuk oleh variabel `y`. Sehingga ketika isi dari variabel `y` tersebut dikeluarkan maka nilai 9.8596 yang akan muncul sebagai hasil kuadratnya.

Memasukkan Pendefinisian Fungsi dari Modul yang Lain

Sering kali, kita merasa nyaman untuk memasukkan satu atau lebih pendefinisian fungsi dari modul yang lain. Sebelum fungsi dipanggil, modul yang berisi pendefinisian fungsi tersebut dimasukkan / diimpor.

Dalam contoh ini, pendefinisian fungsi dari fungsi `kuadrat` berada pada modul `pangkat.py` dan instruksi pemanggil fungsi tersebut berada pada `pemanggil.py`. Sebelumnya, pada `pemanggil.py`, kita mengimpor modul `pangkat.py` (letak fungsi tersebut berada) terlebih dahulu. Perintah untuk mengimpor dapat dilakukan dengan menuliskan `import` diikuti dengan nama modul letak fungsi berada(`pangkat`). Kita dapat memanggil fungsi `kuadrat` melalui instruksi di `pemanggil.py` dengan menuliskan nama modul tempat fungsi tersebut berada diikuti dengan tanda titik/ dot (`.`) dan dilanjutkan dengan nama fungsinya (dalam kasus ini, `pangkat.kuadrat()`)

akan dituliskan sebagai instruksi pemanggil fungsi tersebut). Selanjutnya, hasil pemanggilan fungsi tersebut dirujuk oleh variabel `y` dan isi dari variabel `y` tersebut akan ditampilkan ke layar, seperti yang telah dilakukan pada subbab sebelumnya. Dalam hal ini, Contoh program berikut ini akan memperjelas penjelasan mengenai pemanggilan fungsi yang berada di lain modul.

pangkat.py

```
def kuadrat() :
    x = 3.14
    hsl = x**2
    return hsl
```

pemanggil.py

```
import pangkat
y = pangkat.kuadrat()
print("Nilai dari y adalah ", y)
```

Setelah kedua modul program tersebut disimpan dan dieksekusi, *pemanggil.py* akan mengimpor modul *pangkat.py* dan memanggil fungsi `kuadrat` di dalam *pangkat.py*. Nilai 9.8596 yang merupakan hasil dari fungsi `kuadrat` pada modul *pangkat.py* akan dikeluarkan. Nilai ini diperoleh dari $3.14 * 3.14$ (sesuai dengan isi dari fungsi `kuadrat`), seperti yang telah dibahas pada subbab sebelumnya. Nilai tersebut akan dirujuk oleh variabel `y` dan akan dicetak ke

Fungsi

dalam sebuah kalimat “Nilai dari variabel y adalah 9.8596”.

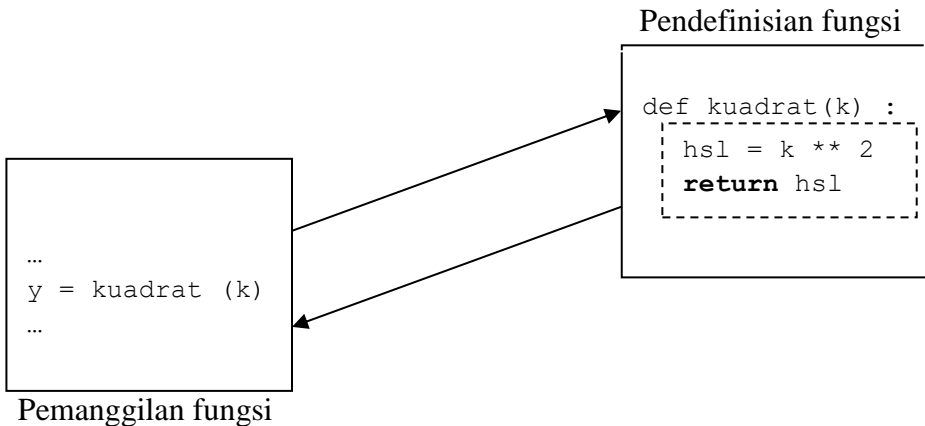
Pemanggilan Fungsi dengan Argumen

Suatu fungsi dapat didefinisikan dengan menggunakan satu atau lebih parameter. Parameter ini yang nantinya sebagai data dalam proses pengiriman data pada saat fungsi tersebut dipanggil. Instruksi pemanggil fungsi akan mengirimkan data pada saat proses pemanggilan dan data tersebut akan diolah oleh fungsi sebagai parameter masukan dari fungsi tersebut.

Parameter masukan yang diterima dari instruksi pemanggil data akan diperlakukan sebagai data (variabel) lokal oleh fungsi. Data (variabel) lokal tersebut tentunya memiliki ruang lingkup lokal (hanya pada tubuh atau isi fungsi itu saja). Nilai data yang digunakan pada saat memanggil fungsi ini disebut argumen. Setiap argumen dalam proses pemanggilan fungsi harus sesuai dengan spesifikasi parameter dalam fungsi yang telah didefinisikan. Sebagai contoh, sebuah fungsi yang dipanggil dengan dua argumen harus telah didefinisikan dengan menggunakan dua parameter di dalamnya.

Salah satu contoh pemanggilan fungsi dengan argument ini dapat dilihat pada Gambar 6.4. Gambar

tersebut menunjukkan contoh dimana fungsi kuadrat memerlukan satu parameter di dalamnya. Pemanggilan fungsi ini juga berbeda dengan pemanggilan fungsi yang sebelumnya. Pemanggilan fungsi ini membutuhkan argumen agar fungsi dapat menghitung nilai kuadrat dari nilai di dalam argumen tersebut sehingga fungsi yang dipanggil tersebut dapat mengembalikan hasil nilai yang telah dihitung.



Gambar 6.4. Pemanggilan fungsi kuadrat dengan argumen

Program Python berikut ini menunjukkan pendefinisian fungsi `kuadrat` yang menggunakan satu parameter `k` dan mengembalikan hasil berupa nilai variabel lokal `hsl`.

Fungsi

Program ini menghitung nilai kuadrat dari sebuah variabel

```
def kuadrat(k) :  
    """  
        Fungsi ini mengembalikan hasil  
        kuadrat dari k  
    """  
    hsl = k ** 2  
    return hsl  
  
x = 4  
y = kuadrat(x)  
print("Nilai dari argumen x : ", x)  
print("Nilai dari y : ", y)
```

Ketika program tersebut dieksekusi, akan tampil keluaran program sebagai berikut ini.

```
Nilai dari argument x : 4  
Nilai dari y : 16
```

Pemanggilan Fungsi yang berada di Modul yang Lain

Kita dapat menggunakan sebuah fungsi meskipun fungsi tersebut berada di modul yang berbeda dengan program tersebut berada. Jika fungsi kuadrat tersimpan di modul `fku.py` sedangkan program kita tidak berada pada modul tersebut, kita dapat memanggilmnya seperti pada contoh program berikut ini.

```
y = fku.kuadrat(3.5)
```

Kita dapat memanggil fungsi `kuadrat` dengan menuliskan nama modul (`fku`) lalu diikuti tanda dot (`.`) dan nama fungsi tersebut (`kuadrat`). Selanjutnya hasil dari proses kuadrat (12.25) akan dimunculkan sebagai hasil luaran dari pemanggilan fungsi `kuadrat` tersebut.

Fungsi Matematika *Built-In*

Python menyediakan berbagai fungsi yang dikelola dan disimpan di berbagai pustaka modul-modul standar serta disertai dengan banyak fungsi yang telah didefinisikan. Salah satu pustak standar tersebut adalah pustaka `math`. Dalam program Python, akses ke pustaka ini dapat dilakukan dengan mengimpornya menggunakan perintah `import math` pada bagian atas program Python.

Program berikut ini berisikan pemanggilan fungsi `cos` yang diterapkan pada variabel `x`. Variabel `x` ini merupakan argumen yang ditulis di dalam tanda kurung dan nilainya diasumsikan dalam skala radian. Nilai yang dikembalikan oleh fungsi tersebut adalah hasil dari cosinus `x`.

Fungsi

```
from import math *
# pencetakan variabel pi = 3.14 yang ada di
pustaka math
print("Nilai dari pi : ", pi)
x = 1/3*pi
print("Nilai x : ", x)
# penggunaan pi sebagai pi radian dalam
argumen fungsi cos
print("Nilai cos dari x : ", cos(x))
```

Program di atas memungkinkan untuk memanggil variabel `pi` yang tersedia pada pustaka `math`. Isi dari variabel `pi` (3.14) akan dikalikan $1/3$ dan dirujuk oleh variabel `x`. Hasil yang ditampilkan dari program di atas adalah sebagai berikut:

```
Nilai pi : 3.141592653589793
Nilai x : 1.0471975511965976
Nilai cos dari x : 0.5000000000000001
```

Akar kuadrat dari suatu variabel z atau secara matematis dinyatakan \sqrt{z} dapat dihitung menggunakan fungsi `sqrt()`. Fungsi ini juga berada dalam modul matematika. Oleh karena itu kita tetap harus mengimpor pustaka `math` terlebih dahulu untuk dapat menggunakan fungsi tersebut. Sebagai contoh, kita ingin menghitung suatu variabel v dengan ekspresi matematika berikut ini.

$$v = \sqrt{\sin^2 x + \cos^2 y}$$

Perintah-perintah yang diperlukan untuk menghitung isi dari variabel `v` terlihat pada program berikut ini. Program ini memerlukan tiga fungsi `sin()`, `cos()`, dan `sqrt()` untuk menghitung nilai sinus, cosinus dan akar kuadrat dari masing-masing argument atau variabel yang di dalamnya.

```
from math import *
x = 1/6*pi
y = 1/3*pi
v = sqrt(sin(x) ** 2 + cos(y) ** 2)
print("Nilai dari v adalah ", v)
```

Pertama, program akan menghitung nilai `x` dan menghasilkan nilai 0.5235987755982988, sedangkan nilai `y` adalah 1.0471975511965976. Setelah itu, program tersebut akan menghitung nilai dari `v`. Nilai `v` diperoleh dari persamaan atau ekspresi matematika di atas. Hasil dari variabel `v` adalah 0.7071067811865476.

Fungsi eksponensial `exp()` digunakan untuk menghitung nilai dari e yang dipangkatkan oleh suatu angka dalam argumen fungsi tersebut. Sebagai contoh, kita ingin menghitung nilai e^3 , kita dapat memanggil fungsi tersebut dengan memasukkan argument 3 di dalamnya. Pemanggilan fungsi tersebut adalah `exp(3)`. Hasil dari pemanggilan fungsi tersebut adalah 20.085536923187668.

Fungsi

Kita juga dapat menghitung logaritma berbasis e dari suatu variabel x atau secara matematis, kita mengenalnya dengan nama $\ln x$ atau $\log_e x$. Fungsi yang diperlukan untuk menghitung logaritma natural tersebut adalah `log()`. Sebagai contoh, kita ingin menghitung nilai logaritma natural dari e . Kita dapat memanggil fungsi tersebut dan memasukkan e sebagai argumen di dalamnya. Setelah perintah `log(e)` dieksekusi, kita memperoleh hasil 1 sebagai nilai pengembaliannya. Hal ini sesuai dengan rumus matematika yang telah kita pelajari bahwa $\ln e$ atau $\log_e e$ adalah 1.

Ada banyak fungsi yang terdapat pada pustaka `math`. Beberapa fungsi tersebut tergolong dalam dua kategori, yakni fungsi matematika dasar serta fungsi trigonometri dan hiperbolik. Tabel 7.1 berisi daftar fungsi matematika dasar yang tersedia di dalam pustaka `math` sedangkan tabel 7.2 berisi daftar fungsi trigonometri dan hiperbolik yang tersedia di dalam pustaka `math`.

Tabel 7.1. Daftar fungsi matematika dasar

Fungsi	Deskripsi
<code>fabs(x)</code>	Mengembalikan nilai harga mutlak/ <i>absolute</i> dari x .

<code>sqrt(x)</code>	Mengembalikan nilai akar kuadrat dari x dengan $x \geq 0$.
<code>pow(x, y)</code>	Mengembalikan nilai x^y .
<code>ceil(x)</code>	Mengembalikan nilai pembulatan ke atas dari x .
<code>floor(x)</code>	Mengembalikan nilai pembulatan ke bawah dari x .
<code>exp(x)</code>	Mengembalikan nilai e^x , dimana e merupakan basis dari logaritma natural.
<code>log(x)</code>	Mengembalikan nilai logaritma natural dari x (berbasis e), dengan $x > 0$
<code>log10(x)</code>	Mengembalikan nilai logaritma basis 10 dari x , dengan $x > 0$

Tabel 7.2. Daftar fungsi trigonometri dan hiperbolik

Fungsi	Deskripsi
--------	-----------

Fungsi

$\sin(x)$	Mengembalikan nilai sinus dari x , dimana x yang digunakan adalah x dalam radian.
$\cos(x)$	Mengembalikan nilai cosinus dari x , dimana x yang digunakan adalah x dalam radian.
$\tan(x)$	Mengembalikan nilai tangen dari x , dimana x yang digunakan adalah x dalam radian.
$\text{asin}(x)$	Mengembalikan nilai arcus sinus dari x , dimana $-1 < x < 1$.
$\text{acos}(x)$	Mengembalikan nilai arcus cosinus dari x , dimana $-1 < x < 1$.
$\text{atan}(x)$	Mengembalikan nilai arcus tangen dari x .
$\text{atan2}(y, x)$	Mengembalikan nilai arcus tangen dari y/x .
$\sinh(x)$	Mengembalikan nilai sinus hiperbolik dari x .
$\cosh(x)$	Mengembalikan nilai cosinus hiperbolik dari x .

$\tanh(x)$	Mengembalikan nilai tangen hiperbolik dari x .
------------	--

Selain fungsi trigonometri dasar yang kita kenal (\sin , \cos , dan \tan). Pustaka `math` juga menyediakan fungsi-fungsi trigonometri lain seperti \arcsin dan arctan . Masing-masing fungsi itu juga bekerja pada tiga fungsi trigonometri dasar tersebut.

\arcsin menandakan kebalikan dari pencarian nilai sinus, cosinus, atau tangen. Melalui \arcsin , kita dapat mencari sudut jika diketahui nilai sinus, cosinus atau tangen-nya. Misalnya kita ingin mencari berapa sudut dari hasil sinus 0.5, kita dapat menggunakan $\arcsin(0.5)$ yang menghasilkan nilai 0.5235987755982989. Kita perlu mengingat bahwa hasil dari perintah tersebut berada dalam skala radian. Oleh karena itu, kita perlu mengubahnya menjadi satuan derajat (sudut). Kita dapat sedikit memodifikasinya menjadi $\arcsin(0.5) / \pi * 180$. Dari perintah itu, kita memperoleh hasil 30 yang berarti sudut 30 derajat, dimana sinus dari sudut tersebut adalah 0.5. Hal ini juga berlaku sama pada cosinus dan tangen. Khusus tangen, kita dapat menghitung arctan tangennya dari dua argumen y dan x . Dimana nilai yang dicari arctan tangennya dari y/x .

Fungsi-fungsi hiperbolik juga dapat diterapkan menggunakan pustaka `math`. Fungsi-fungsi tersebut antara lain `sinh()` yang digunakan untuk menghitung sinus hiperbolik, fungsi `cosh()` digunakan untuk menghitung cosinus hiperbolik, dan yang terakhir `tanh()` digunakan untuk menghitung tangen hiperbolik. Semua fungsi tersebut akan mengembalikan hasilnya sesuai dengan ekspresi atau persamaan matematikanya masing-masing.

Fungsi Rekursif

Sebuah fungsi rekursif merupakan fungsi yang memanggil dirinya sendiri. Hal ini terdengar tidak seperti biasanya. Misalkan kita menghadapi tugas yang berat, contoh membersihkan seluruh rumah. Kita mungkin berkata kepada diri kita sendiri, “Saya akan memilih satu kamar dan membersihkannya, lalu saya akan membersihkan kamar yang lain”. Dengan kata lain, tugas pembersihan tersebut akan memanggil dirinya sendiri, tetapi menggunakan masukan yang lebih sederhana. Pada akhirnya, setelah fungsi tersebut selesai dieksekusi, semua kamar pada rumah tersebut akan selesai dibersihkan.

Dalam Python, sebuah fungsi rekursif menggunakan prinsip yang sama. Misalnya, kita ingin mencetak pola segitiga berikut ini.

```

[]
[] []
[] [] []
[] [] [] []

```

Secara spesifik, fungsi yang diperlukan untuk mencetak segitiga tersebut adalah sebagai berikut ini.

```
def cetak_segitiga(panjang_sisi) :
```

Segitiga tersebut dicetak dengan memanggil `cetak_segitiga(4)`. Untuk melihat cara rekursi tersebut bekerja, kita dapat menganggap sebuah segitiga dengan panjang sisi 4 dapat diperoleh dari sebuah segitiga dengan panjang sisi 3.

```

[]
[] []
[] [] []
[] [] [] []

```

```
Cetak segitiga dengan panjang sisi 3
```

Fungsi

Cetak sebuah baris baru dengan 4 []

Secara umum, instruksi Python untuk mencetak segitiga dengan panjang sisi bebas. Sebuah rekursif memecahkan masalah dengan menggunakan solusi

```
def cetak_segitiga(panjang_sisi) :  
    cetak_segitiga(panjang_sisi -  
1)  
    print("[]" * panjang_sisi)
```

Tetapi ada suatu masalah dengan ide di atas. Ketika panjang_sisi-nya adalah 1, kita tidak mungkin memanggil fungsi cetak_segitiga dengan parameter 0 atau -1. Solusi yang memungkinkan untuk masalah tersebut dengan tidak mencetak apapun ketika panjang_sisi kurang dari 1.

```
def cetak_segitiga(panjang_sisi) :  
    if panjang_sisi < 1 : return  
    cetak_segitiga(panjang_sisi -  
1)  
    print("[]" * panjang_sisi)
```

Pada program di atas terdapat sebuah percabangan. Jika panjang sisi adalah 0, tidak ada yang perlu dicetak. Bagian selanjutnya adalah pencetakan segitiga dengan panjang sisi yang lebih kecil. Selanjutnya, program tersebut akan mencetak [] sesuai dengan isi dari variabel panjang sisi-nya pada saat itu. Pada akhirnya, program tersebut

menghasilkan bentuk segitiga sesuai dengan panjang sisi yang diinginkan.

Ada dua persyaratan utama untuk memastikan fungsi rekursif tersebut berhasil, yakni:

1. Setiap pemanggilan rekursif harus dapat menyederhanakan permasalahan dalam beberapa bagian yang lebih kecil (harus ada kondisi ketika melakukan pemanggilan rekursif);
2. Harus ada sebuah *case special* untuk menangani permasalahan yang sederhana secara langsung atau kondisi untuk menghentikan rekursi.

Fungsi `cetak_segitiga` memanggil dirinya sendiri dengan isi variabel `panjang_sisi` yang lebih kecil dan semakin kecil sampai `panjang_sisi` mencapai 0 (sampai fungsi tersebut berhenti memanggil dirinya sendiri). Berikut ini yang terjadi ketika kita mencetak sebuah segitiga dengan panjang sisi 4.

- Fungsi `cetak_segitiga(4)` memanggil `cetak_segitiga(3)`.
- Fungsi `cetak_segitiga(3)` memanggil `cetak_segitiga(2)`.

Fungsi

- Fungsi `cetak_segitiga(2)` memanggil `cetak_segitiga(1)`.
- Fungsi `cetak_segitiga(1)` memanggil `cetak_segitiga(0)`.
- Fungsi `cetak_segitiga(0)` tidak mengembalikan hasil karena tidak melakukan apa-apa.
- Fungsi `cetak_segitiga(1)` mencetak `[]`.
- Fungsi `cetak_segitiga(2)` mencetak `[] []`.
- Fungsi `cetak_segitiga(3)` mencetak `[] [] []`.
- Fungsi `cetak_segitiga(4)` mencetak `[] [] [] []`.

Pola pemanggilan dari sebuah fungsi rekursif terlihat rumit, namun kunci desain yang baik dari fungsi rekursif adalah tidak perlu berpikir terlalu rumit.

Rekursi tidak satu-satunya cara yang digunakan untuk mencetak bentuk segitiga. Kita dapat menggunakan perulangan bersarang, seperti berikut ini.

```
def cetak_segitiga(panjang_sisi) :  
    for i in range(panjang_sisi) :  
        print("[]" * i)
```

Namun, perulangan ini agak sedikit rumit. Banyak orang menganggap solusi rekursif lebih mudah untuk

dipahami. Contoh program lengkap dalam pencetakan segitiga menggunakan fungsi rekursif dapat dilihat pada bagian berikut ini.

```
#####
#####Program ini menunjukkan
cara pencetakan segitiga menggunakan fungsi
rekursif.
#####
#####
def main() :
    cetak_segitiga(4) #          pemanggilan
fungsi rekursif

# Pencetakan sebuah segitiga dengan panjang
sisi yang diberikan
# Parameter panjang_sisi : variabel integer
dari panjang sisi segitiga
def cetak_segitiga(panjang_sisi) :
    if panjang_sisi < 1 : return
    cetak_segitiga(panjang_sisi - 1)

    #Penambahan baris di bagian bawah
    print("[]" * panjang_sisi)

# Memulai program
main() #pemanggilan fungsi utama
```

Kesimpulan



Fungsi merupakan unit modular dasar dari sebuah program. Suatu fungsi harus didefinisikan terlebih dahulu sebelum dapat digunakan. Penggunaan fungsi tersebut dapat dilakukan dengan dipanggil oleh perintah-perintah yang ada di dalam sebuah program utama. Suatu fungsi tidak hanya dapat dipanggil tetapi perintah yang ada di dalam fungsi tersebut juga dapat memanggil fungsi-fungsi yang lain.

Proses pemanggilan ini melibatkan beberapa mekanisme untuk pengiriman data. Oleh karena itu, fungsi terbagi menjadi beberapa jenis berdasarkan pengiriman data tersebut. Jenis yang pertama adalah fungsi sederhana. Fungsi sederhana ini merupakan fungsi yang paling dasar karena dalam fungsi ini tidak terjadi proses pengiriman data antara fungsi yang dipanggil dengan perintah pemanggilnya. Jenis selanjutnya adalah fungsi yang mengembalikan nilai ke perintah yang memanggilnya. Pemanggilan fungsi yang mendefinisikan satu atau lebih parameter di dalamnya melibatkan pengiriman nilai dari perintah pemanggil ke fungsi yang dipanggilnya. Nilai tersebut akan dijadikan masukan oleh fungsi yang dipanggil untuk mengisi parameter yang ada di dalam fungsi tersebut.

Komputasi rekursif dapat memecahkan masalah dengan menggunakan solusi untuk masalah yang sama dengan masukan yang lebih sederhana. Dalam pembuatan fungsi rekursif, ada beberapa hal yang perlu diperhatikan, antara lain:

- Harus ada sebuah *case* khusus dalam menerima masukan paling sederhana, yang digunakan untuk menghentikan rekursi.
- Kunci untuk menemukan solusi rekursif adalah dengan mengurangi parameter fungsi ke parameter yang lebih kecil untuk masalah yang sama.
- Pada saat merancang sebuah solusi rekursif, kita tidak perlu mengkhawatirkan beberapa pemanggilan fungsi secara bersarang. Kita hanya cukup fokus pada pengurangan sebuah masalah menjadi masalah yang lebih sederhana.

BAB 7

ANIMASI

Pemrograman Python memungkinkan kita untuk memanipulasi angka, teks, dan struktur data seperti *list*, tabel, dan string. Pada era komputer sekarang, berbagai jenis grafik telah digunakan. Kemajuan teknologi perangkat keras membuat para developer perangkat lunak dapat berkreasi lebih dalam membuat berbagai animasi grafis. Perkembangan animasi dari 2D dan 3D membuat pergerakan setiap obyek grafis menjadi lebih hidup dan nyata. Pada bagian ini, kita belajar membuat grafik dan animasi 3D sederhana menggunakan VPython.

Tujuan

- Menjelaskan konsep VPython dan cara penggunaannya untuk mempermudah pemrograman;
- Membuat obyek 3D sederhana menggunakan perintah yang ada di dalam pustaka VPython;

- Animasi
- Menjelaskan konsep dasar dari grafika komputer, khususnya peran sistem koordinat dan transformasi koordinat;
 - Menerapkan konsep *variable assignment* dalam pembuatan obyek 3D sederhana;
 - Menerapkan perulangan dalam pembuatan animasi obyek grafis;
 - Menerapkan *list* dalam mempermudah pembuatan beberapa obyek grafis.

Pendahuluan

VPython merupakan bahasa pemrograman yang mudah dipelajari dan cocok digunakan untuk membuat model interaktif 3D dari sebuah sistem. VPython membuat penulisan program yang menghasilkan animasi 3D menjadi lebih mudah. Hal ini berdasar pada bahasa pemrograman Python yang banyak dikenalkan dalam materi pengantar bahasa pemrograman dan banyak digunakan dalam bidang sains dan bisnis. VPython memiliki tiga komponen di dalamnya, yakni:

1. Python, sebuah bahasa pemrograman yang ditemukan pada tahun 1990 oleh Guido van Rossem, seorang ilmuwan komputer Belanda.

Python merupakan bahasa pemrograman berorientasi obyek yang mudah dipelajari.

2. Visual, modul grafis 3D untuk Python yang dibuat pada tahun 2000 oleh David Scherer saat ia masih mahasiswa di Carneige Melon University. Visual memungkinkan kita untuk membuat dan membuat animasi pada obyek 3D, serta menavigasi dalam adegan 3D dengan memutar dan memperbesarnya menggunakan tetikus.
3. IDLE, sebuah lingkungan pengeditan interaktif, yang dibuat oleh van Rossem dan dimodifikasi oleh Scherer, yang memungkinkan kita untuk memasukkan kode program, mencoba program tersebut, dan memperoleh informasi mengenai program kita.

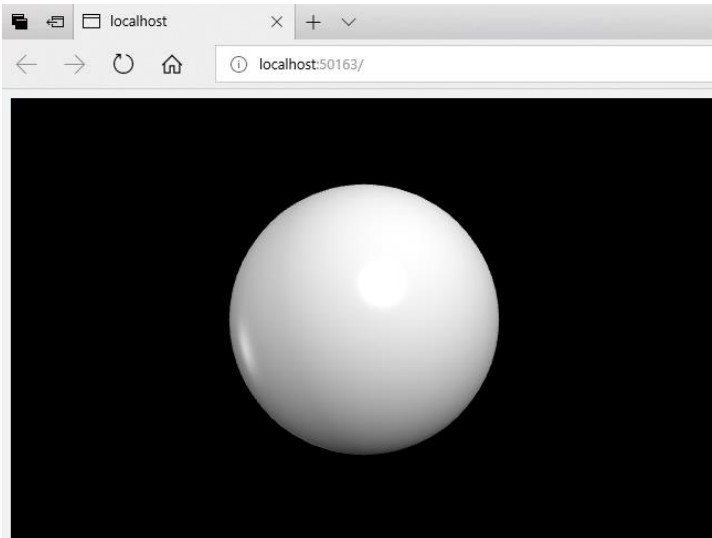
Obyek 3D

Visual Python memudahkan kita dalam membangun obyek-obyek 3D. Kita dapat memanfaatkannya dengan mengimpor fungsi-fungsi yang diperlukan dalam pustaka **vpython**. Setelah itu, kita dapat menuliskan nama obyek 3D yang ingin kita buat dan

diikuti dengan tanda kurung. Salah satu contoh kita dapat menggunakan fungsi **sphere()** untuk membuat sebuah bola. Berikut ini merupakan perintah yang diperlukan untuk menggambar sebuah bola.

```
from vpython import *  
sphere()
```

Setelah program tersebut dieksekusi, kita dapat melihat sebuah bola pada *browser* kita (secara *default*). Tampilan bola tersebut terlihat pada Gambar 7.1. Sebuah bola yang berada pada tengah *scene* dan memiliki jari-jari dan warna yang telah ditentukan secara *default*. Dua properti ini sering kali disebut dengan nama atribut. Semua bola pada fungsi **sphere()** memiliki jenis atribut yang sama. Kita dapat mengubah nilai dari atribut yang dimiliki oleh bola tersebut.



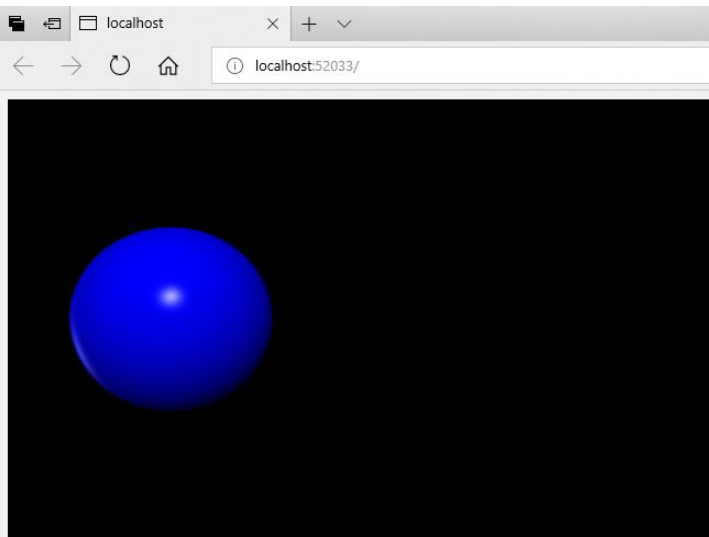
Gambar 7.1. Tampilan bola pada browser

Posisi bola dapat diubah berdasarkan keinginan kita. Misalnya kita ingin meletakkan bola tersebut ke kiri layar dengan mengubah koordinat x dari posisi bola menjadi -1 yang mana atribut posisi tersebut adalah sebuah vektor. Jadi nilai negatif dan positif dari sebuah angka menandakan arah dari obyek bola tersebut. Selanjutnya, kita dapat mengubah jari-jari dari bola tersebut. Misal, kita mengubah nilai jari-jari menjadi 0.5 sehingga besar atau volume bola yang dihasilkan adalah setengah dari bola yang sebelumnya. Warna bola juga dapat diubah menjadi warna lain seperti hijau, merah, cyan, magenta, kuning, dan lain-lain. Pada contoh ini, kita menggunakan warna biru sebagai warna dasar dari

bola. Kita dapat menuliskan kode program seperti di bawah ini.

```
from vpython import *  
sphere(pos = vector(-1,0,0), radius = 0.5,  
color = color.blue)
```

Apabila program tersebut dijalankan. Bola yang dihasilkan dapat dilihat pada Gambar 7.2.



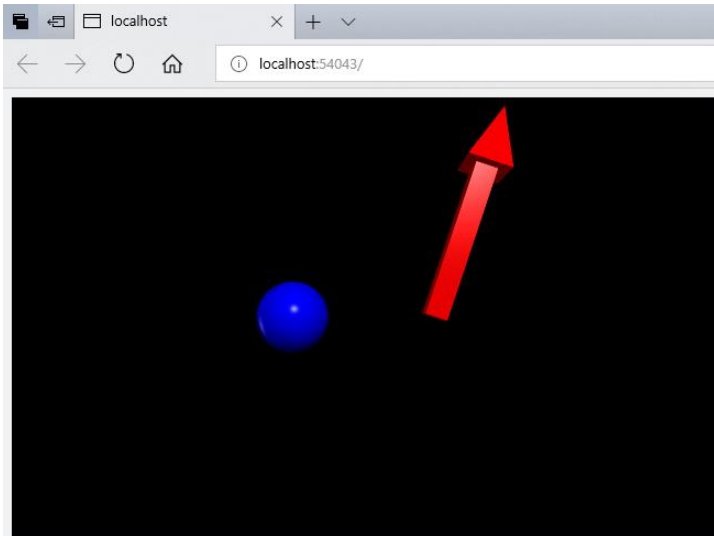
Gambar 7.2. Perubahan posisi dan warna bola

Setiap obyek 3D memiliki atribut yang berbeda. Seperti pada obyek panah, obyek 3D ini tidak memiliki semua atribut yang dimiliki oleh bola. Sebagai contoh, atribut jari-jari tidak dimiliki oleh atribut panah. Obyek panah ditentukan dari besar dan

arah bentangan dari obyek tersebut, yang merupakan *properties* dari obyek itu. Selanjutnya, kita belajar membuat obyek panah berwarna merah dengan posisi vektor $(1, 0, 0)$. Atribut posisi ini merupakan posisi dari pangkal (ekor) panah. Berikutnya, kita menentukan sumbu peregangan dari panah dengan memasukkan 1 unit pada sumbu x dan memasukkan 3 unit pada sumbu y serta memasukkan 0 unit pada sumbu z. Tanda positif dan negatif pada unit yang dimasukkan menentukan arah pada panah tersebut terhadap sumbu x, y, dan z sesuai koordinat ruang dari panah tersebut. Berikut ini merupakan kode program untuk pembuatan panah tersebut.

```
from vpython import *
arrow(pos = vector(1,0,0), axis =
vector(1,3,0), color = color.red)
```

Ketika program pembuatan obyek bola dan panah digabung, pengguna akan melihat gabungan dua obyek tersebut pada Gambar 7.3.



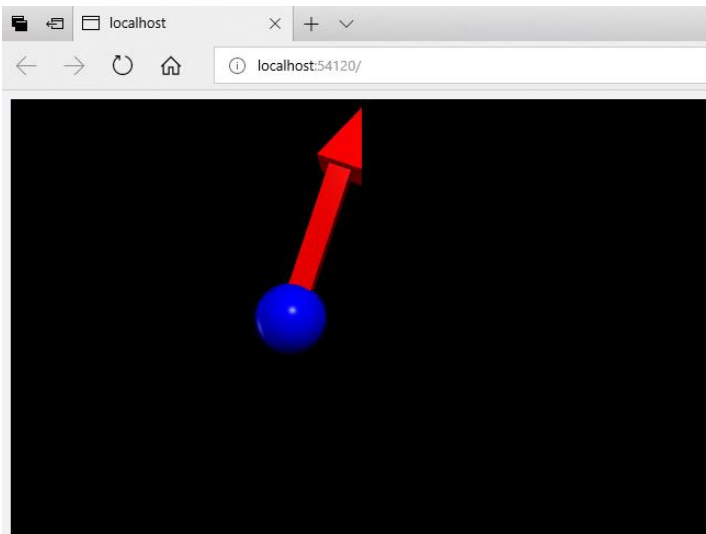
Animas Gambar 7.3. Obyek bola dan panah

Kita dapat memodifikasi dari kedua obyek tersebut. Misalnya, kita ingin bereksperimen terhadap obyek panah. Kita ingin memindahkan posisi dari pangkal (ekor) panah ke posisi bola berada. Kita perlu merubah sintaks program dari pembuatan obyek panah tersebut menjadi sama dengan posisi yang dimiliki oleh obyek bola. Oleh karena obyek bola berada pada posisi vektor $(-1,0,0)$ maka posisi vektor obyek panah juga perlu diubah menjadi $(-1,0,0)$. Kode program pembuatan obyek bola yang menempel dengan obyek panah akan terlihat seperti berikut ini.

```
from vpython import *
```

```
sphere(pos = vector(-1,0,0), radius = 0.5,
color = color.blue)
arrow(pos = vector(-1,0,0), axis =
vector(1,3,0), color = color.red)
```

Kode program tersebut akan menghasilkan luaran gambar obyek panah yang melekat di atas obyek bola. Gambar kedua obyek tersebut dapat dilihat pada Gambar 7.4.



Gambar 7.4. Obyek bola melekat dengan obyek panah

Pada Gambar 7.4, kita dapat melihat perubahan hanya terjadi pada pergeseran titik awal yang menjadi titik pangkal (ekor) dari panah tersebut. Kita juga dapat

memodifikasi arah bentangan panah berdasarkan sumbu *Animasi* x, y, dan z dengan mengubah kode program pada fungsi `axis` yang ada pada obyek panah.

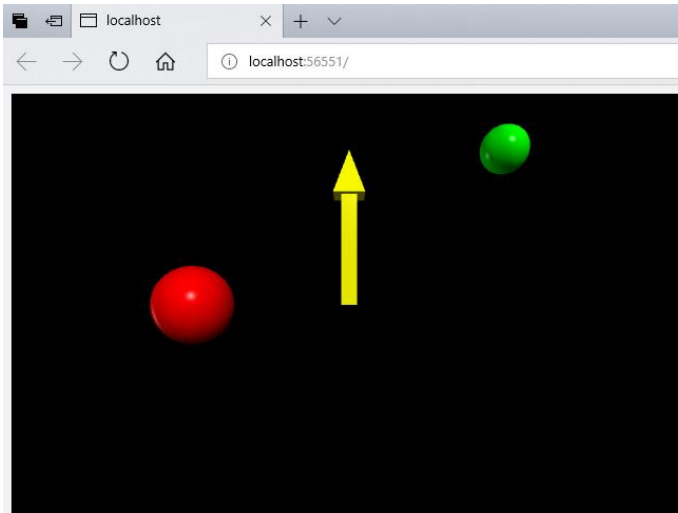
Kita dapat mengeksplorasi obyek-obyek 3D yang lain seperti *cylinders*, *cones*, dan *helixes* dengan membaca VPython *manual* yang berada bagian dokumentasi di halaman *website* VPython. Kita dapat mencoba berbagai contoh sintaks program pembuatan obyek-obyek 3D di halaman <http://www.glowscript.org/docs/VPythonDocs/index.html>.

Variable Assignment

Pada bagian ini, kita akan belajar untuk memasukkan sebuah obyek ke dalam sebuah variabel sehingga dapat digunakan lebih lanjut. Sebagai langkah awal, kita membuat dua obyek bola dan sebuah obyek panah dengan menggunakan kode program berikut ini.

```
from vpython import *  
sphere(pos = vector(-1,0,0), radius = 0.25,  
color=color.red)  
sphere(pos = vector(1,1,0), radius = 0.15,  
color=color.green)  
arrow(pos = vector(0,0,0), axis =  
vector(0,1,0), color = color.yellow)
```

Luaran dari program adalah dua obyek bola dan sebuah obyek panah di antara kedua bola tersebut. Luaran tersebut dapat dilihat pada Gambar 7.5.



Gambar 7.5. Dua obyek bola dan obyek panah

Kita dapat memberikan referensi pada setiap atribut *sphere* pada sebuah variabel. Misal atribut pada obyek bola pertama dirujuk pada variabel `bundar` dan variabel `bulat` merujuk pada obyek bola kedua. Hal ini dapat dilihat pada potongan kode program berikut ini.

```
bundar = sphere(pos = vector(-1,0,0), radius  
= 0.25, color=color.red)  
bulat = sphere(pos = vector(1,1,0), radius  
= 0.15, color=color.green)
```

Ketika kita memanggil salah satu variabel (`bundar` atau `bulat`), kita juga dapat mengakses semua informasi di dalamnya (posisi, jari-jari, dan warna dari obyek bola yang dirujuk oleh variabel tersebut). Hal ini dapat menjadi sangat berguna ketika kita ingin meletakkan pangkal (ekor) dari obyek panah pada lokasi yang ada pada variabel `bundar`.

Cara mengakses atribut yang ada dalam sebuah variabel cukup mudah. Kita dapat menuliskan nama variabel tersebut disambung dengan tanda *dot* (.) dan dilanjutkan dengan nama atribut yang ingin diakses. Format penulisannya dapat dilihat di bawah ini.

```
nama_variabel.nama_atribut
```

Sebagai contoh, kita ingin mengakses atribut posisi dari variabel `bundar`. Kita dapat menuliskan `bundar.pos` untuk merepresentasikan posisi vektor dari obyek bola yang dirujuk oleh variabel `bundar`. Kita dapat memanfaatkan hal tersebut ketika kita ingin mengatur posisi awal dari sebuah obyek panah agar pangkal (ekor) dari panah tersebut berada di posisi yang sama pada obyek bola yang dirujuk oleh variabel `bundar`. Contoh kode programnya dapat dilihat pada bagian di bawah ini.

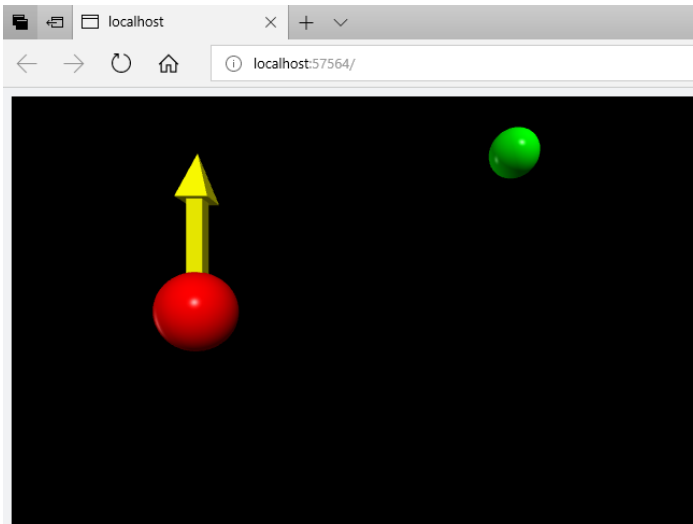
```
from vpython import *
```

```

bundar = sphere(pos = vector(-1,0,0), radius
= 0.25, color=color.red)
bulat = sphere(pos = vector(1,1,0), radius
= 0.15, color=color.green)
#posisi panah disamakan dengan posisi di
variabel bundar
arrow(pos = bundar.pos, axis =
vector(0,1,0), color = color.yellow)

```

Ketika program tersebut dijalankan, luaran yang dihasilkan dapat dilihat pada Gambar 7.6.



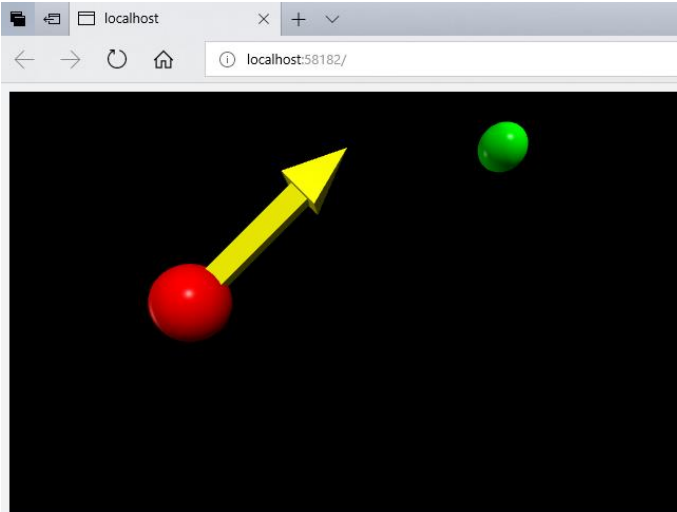
Gambar 7.6. Obyek panah menempel di atas salah satu obyek bola

Kita juga dapat melakukan eksperimen lain dengan menyesuaikan arah panah agar menuju ke posisi obyek bola yang dirujuk oleh variabel `bulat`.

Arah panah tersebut dikendalikan oleh atribut `axis` dari obyek panah. Isi dari atribut `axis` tersebut adalah `posisi_x`, `posisi_y`, dan `posisi_z` yang menandakan panjang dan arah dari obyek panah. Ketiga posisi tersebut merupakan sumbu koordinat 3 dimensi yang sering dipelajari pada bidang matematika. Kita dapat mengganti isi dari atribut `axis` tersebut menjadi posisi yang ada di variabel `bulat` seperti pada program di bawah ini.

```
from vpython import *
bundar = sphere(pos = vector(-1,0,0), radius
= 0.25, color=color.red)
bulat = sphere(pos = vector(1,1,0), radius
= 0.15, color=color.green)
#arah panah disesuaikan dengan posisi di
variabel bulat
arrow(pos = bundar.pos, axis = bulat.pos,
color = color.yellow)
```

Luaran dari program tersebut setelah dieksekusi dapat dilihat pada Gambar 7.7.



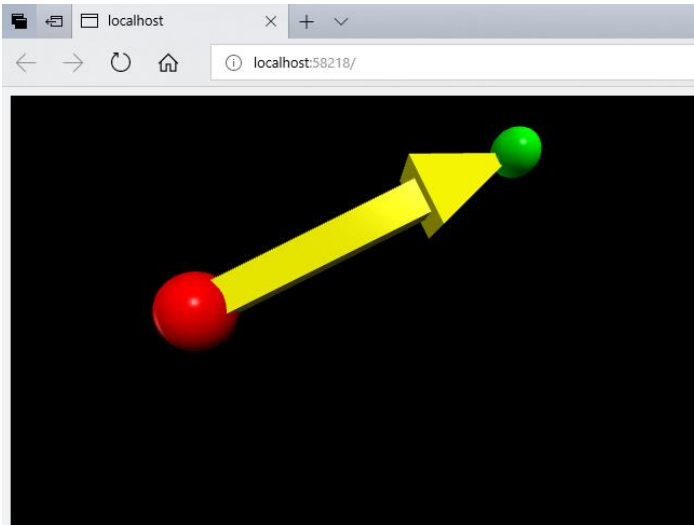
Gambar 7.7. Obyek panah mengarah pada satu obyek bola

Pada luaran yang dihasilkan, tampak obyek panah hanya mengarah tapi tidak sepenuhnya menjangkau dan menempel pada obyek bola yang dirujuk oleh variabel `bulat`. Kita ingin obyek panah dapat meraih obyek bola variabel `bulat` dari posisi obyek bola variabel `bundar`. Agar membuat hal tersebut terjadi, kita dapat mengganti isi dari atribut `axis` dengan mengurangkan posisi dari variabel `bulat` dengan posisi yang dimiliki oleh variabel `bundar`. Kita jadi mengingat cara menghitung posisi relatif dari sebuah titik menuju titik yang lain. Kita dapat menghitungnya dengan mengurangkan koordinat titik akhir dengan

koordinat titik awal. Hal tersebut yang digunakan dalam menghitung atribut `axis` agar obyek panah dapat mencapai obyek bola yang dirujuk oleh variabel `bulat`. Kode program yang diperlukan untuk melakukan hal tersebut adalah sebagai berikut.

```
from vpython import *
bundar = sphere(pos = vector(-1,0,0), radius
= 0.25, color=color.red)
bulat = sphere(pos = vector(1,1,0), radius
= 0.15, color=color.green)
#atribut axis = posisi bulat - posisi bundar
arrow(pos = bundar.pos, axis = bulat.pos -
bundar.pos, color = color.yellow)
```

Luaran yang dihasilkan dapat dilihat pada Gambar 7.8. Pada gambar tersebut, terlihat obyek panah menempel dari salah satu obyek bola dan menuju pada obyek bola yang lainnya. Obyek panah dapat menghubungkan kedua obyek bola yang ada.



Gambar 7.8. Obyek panah menempel pada kedua obyek bola

Perulangan pada Animasi

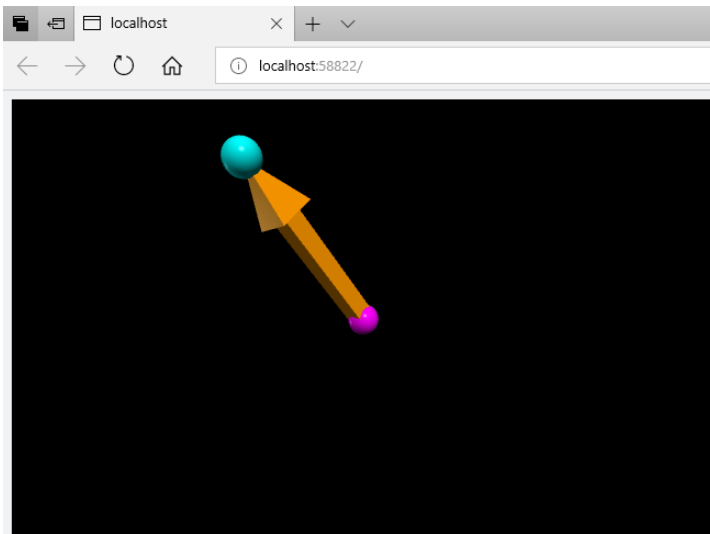
Tujuan dari perulangan adalah menyederhanakan kode-kode program yang sama sehingga tidak perlu dituliskan secara berulang. Perulangan ini telah kita pelajari dalam berbagai hal penyelesaian masalah di beberapa bab sebelumnya. Pada bagian ini, kita membahas penggunaan perulangan pada pembaharuan isi variabel dan pemodelan pergerakan obyek.

Animasi

Pertama, mari kita membuat dua buah obyek bola dan sebuah obyek panah menggunakan kode program berikut ini.

```
from vpython import *
bola1 = sphere(pos = vector(0,0,0), radius
= 0.5, color=color.magenta)
bola2 = sphere(pos = vector(-3,4,3), radius
= 0.5, color=color.cyan)
arrow(pos = bola1.pos, axis = bola2.pos -
bola1.pos, color = color.orange)
```

Luaran dari kode program tersebut dapat dilihat pada Gambar 7.9.



Gambar 7.9. Obyek panah mengarah dari satu bola ke bola lain

Lalu, ada variabel r yang merujuk pada suatu vektor dengan komponen x adalah -3 , komponen y adalah 4 dan komponen z adalah 0 . Kita juga menambahkan perulangan di bawah kode tersebut untuk menghitung nilai yang baru dari vektor r dengan menambahkan konstanta 1 di setiap komponen x pada setiap waktu. Melalui perulangan tersebut, kita dapat membuat beberapa obyek bola baru (sesuai jumlah perulangan) dengan posisi yang ada pada vektor r dengan jari-jari 0.5 dan berwarna *cyan*. Kita juga menambahkan perintah `rate` pada awal perulangan untuk menurunkan kecepatan komputer dalam mengeksekusi komputasi perulangan. Perintah `rate(10)` berarti komputer mengerjakan hanya 10 komputasi dalam satu detik. Kode program yang telah ditambahkan tersebut dapat dilihat pada bagian di bawah ini.

```
from vpython import *
bola1 = sphere(pos = vector(0,0,0), radius
= 0.5, color=color.magenta)
bola2 = sphere(pos = vector(-3,4,3), radius
= 0.5, color=color.cyan)
arrow(pos = bola1.pos, axis = bola2.pos -
bola1.pos, color = color.orange)

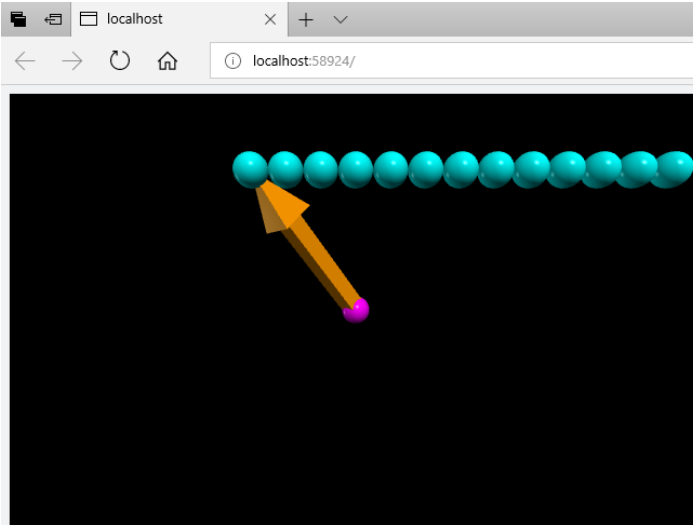
r = vector(-3,4,3)

while r.x < 10:
    rate(10)
```

Animasi

```
sphere(pos = r, radius = 0.5, color =  
color.cyan)  
r.x = r.x + 1  
  
print("Akhir dari program")
```

Luaran dari program tersebut dapat dilihat pada Gambar 7.10. Selain itu, pada layar perintah akan terlihat tulisan `Akhir dari program` yang menandakan program tersebut telah berakhir. Pada gambar 7.10, terlihat ada 13 bola berwarna *cyan*. Hal ini terjadi karena posisi awal bola baru pada sumbu x adalah -3 (sesuai dengan nilai -3 pada komponen x di vektor `r`). Perulangan tersebut memungkinkan penambahan 1 terhadap nilai komponen x pada bola tersebut. Perulangan berhenti ketika nilai komponen dari x mencapai angka 10. Oleh karena perulangan tersebut adalah perulangan `while - do`, perulangan tersebut dilakukan selama syarat yang ada (di dalam perintah `while`) terpenuhi. Ketika nilai komponen x berisi angka 10, syarat di dalam `while` tidak terpenuhi, sehingga kode program yang seharusnya diulang tidak dilakukan perulangan. Oleh sebab itu, perulangan hanya dilakukan 13 kali (mulai dari nilai -3 sampai 9 pada nilai komponen x).



Gambar 7.10. Perulangan pembuatan beberapa obyek bola baru

Pada Gambar 7.10 terlihat penambahan sejumlah bola berwarna *cyan* dengan tanpa menghapus bola-bola yang telah dibuat sebelumnya. Misalnya kita ingin menampilkan sebuah bola yang bergerak pada layar dengan tanpa menambahkan sejumlah bola yang lain. Kita telah memiliki bola berwarna *cyan* dengan jari-jari 0.5 yang dirujuk oleh variabel `bola2`. Selanjutnya, kita dapat melakukan *reassign* posisi dari obyek bola yang dirujuk oleh variabel `bola2` dengan vektor `r` dengan menuliskan `bola2.pos = r` di dalam perulangan `while` tersebut. Program ini memanfaatkan obyek bola yang dirujuk variabel `bola2` untuk berubah posisi sesuai dengan isi

Animasi

dari vektor r (dengan komponen x yang bertambah 1 dalam setiap iterasi perulangan) sehingga kode program tersebut menjadi seperti yang terlihat di bawah ini.

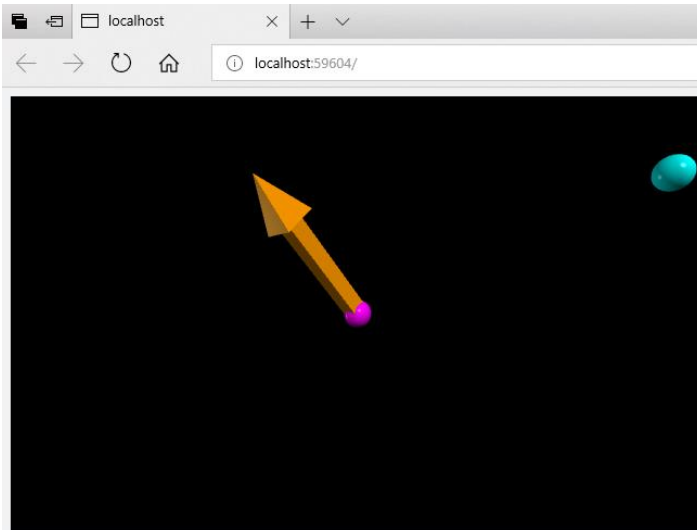
```
from vpython import *
bola1 = sphere(pos = vector(0,0,0), radius
= 0.5, color=color.magenta)
bola2 = sphere(pos = vector(-3,4,3), radius
= 0.5, color=color.cyan)
arrow(pos = bola1.pos, axis = bola2.pos -
bola1.pos, color = color.orange)

r = vector(-3,4,3)

while r.x < 10:
    rate(10)
    #posisi pada bola2 diubah sesuai isi
dari vektor r
    bola2.pos = r
    r.x = r.x + 1

print("Akhir dari program")
```

Luaran dari program tersebut dapat dilihat pada gambar 7.11. Pada Gambar 7.11, terjadi pergeseran bola berwarna *cyan* yang dirujuk oleh variabel `bola2` ke arah kanan sebesar 1 *point* sampai nilai komponen $x < 10$.



Gambar 7.11. Perulangan pergerakan pada sebuah obyek bola

Perbedaan dengan perulangan sebelumnya, perulangan ini dilakukan hanya pada suatu obyek bola dengan menambahkan 1 pada nilai komponen x obyek tersebut sehingga membuat sebuah animasi pergerakan perpindahan bola tersebut ke kanan selama 13 kali. Jadi perulangan ini tidak melakukan pembuatan beberapa obyek bola baru dengan nilai x yang berbeda (seperti yang dilakukan pada perulangan sebelumnya).

Lists pada Animasi

List telah kita kenal pada bab-bab sebelumnya. Penampungan beberapa nilai dapat dilakukan dengan menggunakan *list* tersebut. Dalam beberapa program sebelumnya, *list* dapat digunakan dalam menyelesaikan beberapa permasalahan.

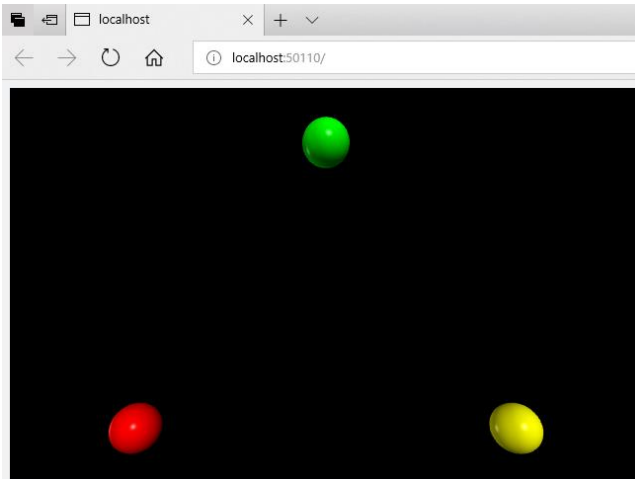
Pada bagian ini, pembahasan dilakukan pada pemanfaatan *list* dalam model komputasional yang melibatkan banyak obyek. Model komputasional akan menjadi mudah ketika sebuah model komputasional hanya menggunakan satu atau dua obyek yang mengacu pada masing-masing obyek dengan setiap nama yang telah ditetapkan. Namun, hal tersebut akan menjadi sebuah masalah yang cukup besar, ketika kita perlu memasukkan sepuluh atau seratus atau seribu obyek dengan mengetikkan masing-masing namanya. Hal tersebut menjadi tidak praktis. Kita dapat membuatnya lebih mudah dan sederhana dengan membuat *list* dari semua obyek yang kita inginkan dan kemudian merujuknya ke masing-masing obyek tersebut tidak menggunakan nama tetapi menggunakan posisi obyek tersebut di dalam *list*.

Pada contoh berikut ini, kita menggunakan tiga obyek bola yang merepresentasikan tiga partikel bermuatan. Ketiga obyek tersebut berada di posisi

yang berbeda-beda dan memiliki warna yang berbeda juga, yakni warna merah, kuning, dan hijau. Masing-masing obyek tersebut memiliki jari-jari 0.005. Perintah untuk membuat ketiga obyek tersebut dapat dilihat pada kode program di bawah ini.

```
from vpython import *
a = sphere(pos = vector(-0.04,-0.03,0),
radius = 0.005, color=color.red)
b = sphere(pos = vector(0.04,-0.03,0),
radius = 0.005, color=color.yellow)
c = sphere(pos = vector(0,0.03,0), radius =
0.005, color=color.green)
```

Hasil luaran dari program tersebut dapat dilihat pada Gambar 7.12.



Gambar 7.12. Pembuaan tiga obyek partikel bermuatan

Kita dapat membuat sebuah *list* yang berisi ketiga obyek tersebut. Misal, kita beri nama *list* tersebut dengan `partikel`. Tanda kurung siku pada pembuatan *list* tersebut mengidentifikasi sebuah *list* yang mana setiap elemen di dalamnya dapat dirujuk berdasarkan posisinya. Posisi atau nomor urutan letak dari sebuah elemen di dalam *list* tersebut dinamakan indeks. Kebanyakan bahasa pemrograman komputer termasuk Python memulai indeks di dalam *list* tersebut dengan angka 0, tidak dengan angka 1. Jadi elemen pertama dalam *list* tersebut berada pada indeks ke-0.

Kita menambahkan perintah untuk mencetak elemen pertama dan terakhir dari *list* tersebut dengan memasukkan angka indeks 0 (pertama) dan 2 (terakhir) di dalam tanda siku. Selanjutnya, kita dapat memilih atribut yang dimiliki obyek bola pertama dan terakhir yang ingin ditampilkan. Pada contoh ini, kita memilih atribut posisi dari kedua obyek bola yang ditampilkan pada layar. Kita dapat menuliskannya menggunakan perintah-perintah sebagai berikut.

```
partikel = [a,b,c]

print(partikel[0].pos)
print(partikel[2].pos)
```

Pada kode program di atas, terlihat ketiga obyek bola *a*, *b*, dan *c* berada di dalam *list* *partikel*. Perintah `partikel[0].pos` merujuk pada posisi yang dimiliki obyek bola *a* (obyek bola pertama dalam *list* *partikel*) sedangkan `partikel[2].pos` merujuk pada posisi yang dimiliki obyek bola *c* (obyek bola terakhir dalam *list* *partikel*). Hasil yang terlihat dari pengekseskusion kode program di atas tampak seperti berikut ini.

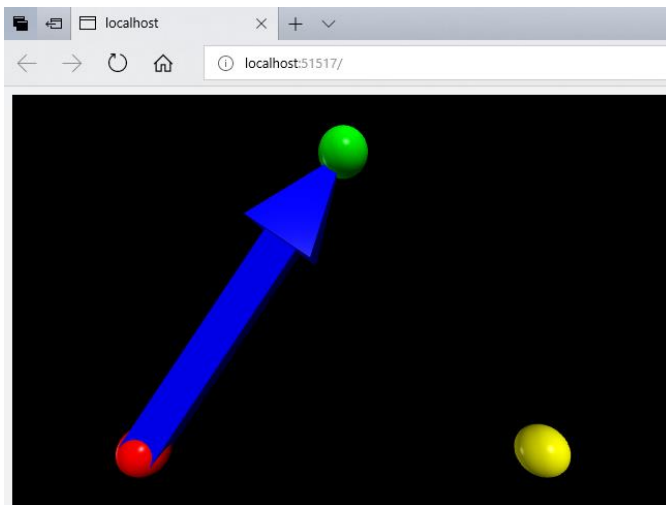
```
<-0.04, -0.03, 0>
<0, 0.03, 0>
```

Kita dapat menggunakan posisi-posisi ini untuk menggambar sebuah obyek panah dengan pangkal (ekor) berada pada posisi obyek bola pada indeks *list* pertama dan membentang menuju ke posisi obyek bola pada indeks *list* terakhir. Oleh sebab itu, atribut *axis* pada panah tersebut diisi dengan sebuah vektor yang menandakan posisi relatif dari obyek bola pada indeks *list* terakhir terhadap obyek bola pada indeks *list* pertama. Hal ini dapat mengurangi nilai posisi obyek bola pada indeks *list* terakhir dengan posisi bola pada indeks *list* pertama. Selanjutnya, kita memberikan warna biru terhadap obyek panah tersebut. Kode program pembuatan obyek panah tersebut dapat dilihat pada potongan program berikut ini.

Animasi

```
arrow(pos = partikel[0].pos, axis =  
partikel[2].pos - partikel[0].pos, color =  
color.blue)
```

Hasil luaran gambar panah yang diperoleh dapat dilihat pada Gambar 7.13. Pada gambar 7.13, obyek panah berwarna biru terlihat membentang dari partikel bermuatan yang berwarna merah (obyek bola pada indeks ke-0 dalam *list* *partikel*) menuju partikel bermuatan yang berwarna hijau (obyek bola pada indeks ke-2 dalam *list* *partikel*).



Gambar 7.13. Pembuatan obyek panah penghubung kedua partikel

Kesimpulan

Obyek 3D dan animasi pergerakan sebuah obyek sering kali dijumpai dalam beberapa aplikasi. Obyek dan animasi tersebut mampu membuat aplikasi menjadi lebih menarik sehingga inti dari aplikasi tersebut dapat tersampaikan dengan baik. Pentingnya kedua hal tersebut membuat *programmer* aplikasi untuk mengembangkan aplikasinya dari *text based* menuju era grafis komputer.

VPython memberikan fasilitas-fasilitas dan kemudahan-kemudahan pada *programmer* dalam membuat beberapa obyek 3D dan mengolah beberapa animasi pergerakannya menggunakan bahasa pemrograman Python. Hal ini juga didukung oleh kesederhanaan yang ada di dalam penggunaan bahasa pemrograman Python. Beberapa fungsi tersedia dalam pembuatan obyek-obyek 3D. Pemakaian atau pemanggilan fungsi-fungsi tersebut juga cukup mudah dan *programmer* tinggal memasukkan nama fungsi beserta nilai-nilai dari atributnya.

Pada bab ini, kita belajar pembuatan beberapa obyek 3D sederhana beserta animasinya menggunakan VPython. Bab ini juga membahas penggunaan *variable assignment* terhadap obyek-obyek 3D yang telah dibuat. Beberapa konsep dasar

pemrograman seperti perulangan dan pembuatan *list* juga digunakan dalam pembuatan obyek 3D maupun animasinya. Bab ini juga dapat menjadi dasar dalam pembuatan animasi dan obyek-obyek 3D berikutnya yang lebih kompleks.

DAFTAR PUSTAKA

- Chabay, R. et al. *VPython GlowScript VPython and VPython* 7, <http://www.glowscript.org/docs/VPythonDocs/index.htm>, accessed on June 22, 2018 at 9:29 AM.
- Garrido, Jose M. 2016. *Introduction to Computational Models with Python*. Georgia: CRC Press.
- Horstmann, C., and Rance Necaise. 2016. United States of America: John Wiley & Sons, Inc.
- Lambert, Kenneth A. 2015. *Python Programming for Teens*. Boston: Cengage Learning PTR.
- Zelle, John M. 2017. *Python Programming: An Introduction to Computer Science*. Oregon: Franklin, Beedle & Associates Inc.

INDEKS

3D, 133, 134, 135, 136,
138, 142, 161, 162
Anaconda, 6, 7, 8, 11
animasi, 2, 133, 134,
135, 156, 161, 162
dialog, 94, 95, 96, 97,
98, 99
files, 44, 86, 93
flowchart, 24, 31
fungsi, i, 14, 15, 20, 43,
68, 81, 88, 92, 94,
95, 96, 97, 100, 101,
102, 103, 104, 105,
106, 107, 108, 109,
110, 111, 112, 113,
114, 115, 116, 117,
118, 119, 120, 121,
122, 123, 124, 125,
126, 127, 128, 129,
130, 131, 132, 136,
142, 162
GlowScript, 1, 2, 4
grafis, 103, 133, 134,
135, 162
list, 43, 44, 45, 46, 47,
48, 49, 50, 51, 52,
53, 54, 55, 56, 57,
58, 59, 60, 61, 62,
63, 64, 65, 66, 67,
68, 69, 70, 71, 73,
74, 77, 98, 133, 134,
156, 158, 159, 160,
162
luaran, 12, 16, 18, 20,
51, 52, 53, 86, 87,
91, 100, 109, 116,
117, 141, 145, 148,
158, 160
masukan, 12, 13, 14,
15, 20, 35, 71, 86,
87, 89, 90, 91, 92,
95, 96, 100, 109,
114, 125, 131, 132
modul, 95, 100, 102,
103, 112, 113, 116,
117, 119, 135
obyek, 87, 88, 133,
134, 135, 136, 137,
138, 139, 140, 141,
142, 143, 144, 146,
147, 148, 149, 150,
151, 153, 154, 155,
156, 157, 158, 159,
160, 161, 162

percabangan, 22, 23,
 25, 26, 29, 30, 31,
 33, 127
 perulangan, 34, 35, 36,
 37, 39, 41, 42, 50,
 51, 52, 73, 74, 75,
 79, 92, 129, 134,
 149, 151, 153, 154,
 156, 162
 Python, i, 1, 2, 6, 11,
 14, 15, 16, 20, 23,
 25, 31, 36, 37, 42,
 43, 44, 45, 46, 54,
 56, 72, 81, 85, 87,
 88, 91, 93, 94, 96,
 98, 99, 100, 101,
 102, 103, 104, 105,
 106, 115, 117, 125,
 126, 133, 134, 135,
 136, 158, 162
 rekursif, 125, 126, 128,
 129, 130, 132
 sintaks, 16, 23, 31, 32,
 35, 37, 38, 41, 140,
 142
 Spyder, 1, 8, 9, 10
 string, 14, 17, 18, 43,
 44, 81, 82, 83, 84,
 85, 86, 88, 89, 90,
 96, 98, 99, 133
 tabel, 72, 73, 74, 75,
 76, 77, 78, 79, 80,
 121, 133
 variabel, 13, 15, 21, 25,
 34, 35, 36, 39, 45,
 46, 47, 48, 49, 50,
 51, 52, 53, 54, 56,
 61, 62, 64, 65, 69,
 70, 74, 75, 81, 82,
 83, 88, 89, 96, 104,
 105, 110, 111, 113,
 114, 116, 118, 119,
 120, 127, 128, 130,
 142, 143, 144, 145,
 146, 147, 148, 150,
 151, 154, 155
 VPython, 1, 2, 4, 6, 7,
 8, 11, 133, 134, 142,
 162