Bab 6. Algoritma dan Module

OBJEKTIF:

1. Mahasiswa Mampu Memahami Algoritma pada Pyhton.
2. Mahasiswa Mampu Memahami Rekursif pada Python.
3. Mahasiswa Mampu Memahami Algoritma Pencarian pada Pyhton.
4. Mahasiswa Mampu Memahami Algoritma Penyortiran pada Python.
5. Mahasiswa Mampu Memahami *Module* pada Python.

6.1 Algoritma

Suatu persoalan dapat mempunyai lebih dari satu algoritma untuk menyelesaikannya. Misalkan, persoalan mencari nilai maksimum dari tiga angka. Salah satu cara untuk menyelesaikan persoalan tersebut adalah dengan menggunakan struktur seleksi seperti berikut:

```
if x1 >= x2 and x1 >= x3:
    max = x1
elif x2 >= x1 and x2 >= x3:
    max = x2
else:
    max = x3
```

Pada kode di atas, kita menguji setiap nilai terhadap nilai-nilai lainnya dan jika nilai tersebut lebih besar maka kita tetapkan menjadi nilai `max`.

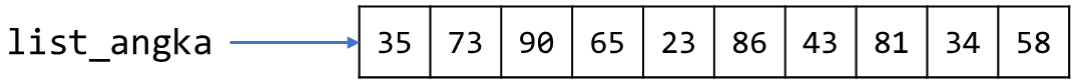
Cara kedua untuk menyelesaikan persoalan pencarian nilai maksimum dari tiga angka adalah dengan mengasumsikan salah satu nilai adalah nilai terbesar lalu bandingkan nilai tersebut dengan nilai-nilai lainnya dan menukar nilai terbesar dengan nilai yang dibandingkan jika nilai yang dibandingkan lebih besar. Kode berikut mencontohkan cara kedua ini:

```
max = x1
if x2 > max:
    max = x2
if x3 > max:
    max = x3
```

Pada kode cara kedua, kita mengasumsikan `x1` sebagai nilai terbesar dengan menyimpan nilainya sebagai `max`. Kemudian, kita membandingkan nilai `max` dengan nilai-nilai lainnya, dan jika nilai tersebut lebih besar dari nilai `max`, tukar nilai `max` dengan nilai tersebut.

Algoritma kedua lebih baik dibandingkan dengan algoritma sebelumnya karena kita dapat mengembangkannya untuk mencari nilai terbesar dari berapapun banyak angka.

Kita juga dapat menggunakan struktur loop untuk mengembangkan algoritma nilai terbesar kedua untuk mencari nilai terbesar dari angka-angka sebanyak berapapun. Misalkan kita mencari nilai terbesar dari 10 angka berikut: 35, 73, 90, 65, 23, 86, 43, 81, 34, 58. Pertama, untuk memudahkan mengolah nilai-nilai di atas kita menyimpannya dalam sebuah *list*:



Kemudian, kita dapat menggunakan struktur *loop* seperti berikut untuk mencari nilai maksimum:

```
# Inisialisasi nilai max dengan elemen pertama
max = angka[0]
for elm in angka:
    if elm >= max:
        max = elm
```

Kita telah melihat suatu persoalan dapat mempunyai lebih dari satu algoritma untuk menyelesaikannya. Namun dari sejumlah algoritma yang menyelesaikan suatu persoalan, algoritma terbaik adalah algoritma yang paling efisien dan biasanya algoritma yang menyelesaikan persoalan tercepat.

6.2 Fungsi Rekursif

Fungsi rekursif berarti fungsi yang memanggil dirinya sendiri. Perhatikan kode berikut:

```
# rekursif_tanpa_akhir.py
# Program ini mempunyai sebuah fungsi rekursif

def pesan():
    print('Ini adalah fungsi rekursif')
    pesan()

def main():
    pesan()

main()
```

Output dari kode di atas:

```
Ini adalah fungsi rekursif
Ini adalah fungsi rekursif
Ini adalah fungsi rekursif
Ini adalah fungsi rekursif
Ini adalah fungsi rekursif
Ini adalah fungsi rekursif
Ini adalah fungsi rekursif
.
.
.
```

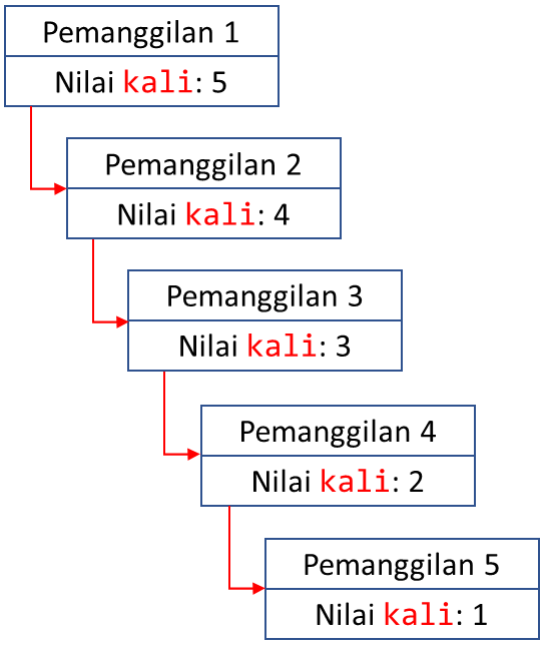
Pada kode di atas, fungsi `pesan` adalah fungsi rekursif karena di dalam definisi fungsi `pesan`, pada baris 6, terdapat sebuah *statement* yang memanggil fungsi `pesan` (dirinya sendiri).

Pada contoh di atas pemanggilan rekursif terjadi terus menerus, karena tidak ada yang menghentikannya. Sama seperti sebuah *loop*, sebuah fungsi rekursif harus mempunyai cara untuk mengendalikan berapa kali fungsi tersebut memanggil dirinya sehingga pemanggilan rekursif dapat berhenti. Kode berikut menambahkan kondisi pada kode sebelumnya:

```
# rekursi.py
# Program ini mempunyai sebuah fungsi rekursif

def pesan(kali):
    if kali > 0:
        print('Ini adalah fungsi rekursif')
        pesan(kali-1)

def main():
    pesan(5)

main()
```

Perhatikan pada kode di atas, kita mengubah definisi fungsi `pesan` dengan menambahkan parameter `kali`. Untuk menghentikan fungsi rekursif `pesan()`, kita memberikan kondisi `kali > 0` (baris 5) dan pada pemanggilan rekursinya kita memberikan argumen `kali - 1` (baris 7) yang dapat mengubah hasil evaluasi kondisi tersebut sehingga dapat menghentikan rekursi dari fungsi `pesan`.

Ilustrasi pemanggilan rekursif pada fungsi `pesan` dengan argumen 5, `pesan(5)`:



6.2.1 *Problem Solving* dengan Rekursif

Untuk mendesain algoritma dengan rekursif, kita mengidentifkasi dua hal:

1. *Base case* (kasus dasar): Kasus dasar adalah kasus dimana pemanggilan rekursif berakhir.
2. *Recursive case* (kasus rekursif): Kasus dimana fungsi memanggil dirinya sendiri.

Kita akan melihat contoh penggunaan algoritma rekursif untuk dua persoalan yaitu Faktorial dan *Fibonacci*.

6.2.1.1 Faktorial

Faktorial dari *integer* positif n dinotasikan dengan $n!$ adalah produk (perkalian) semua *integer* positif kurang dari atau sama dengan n, yaitu:

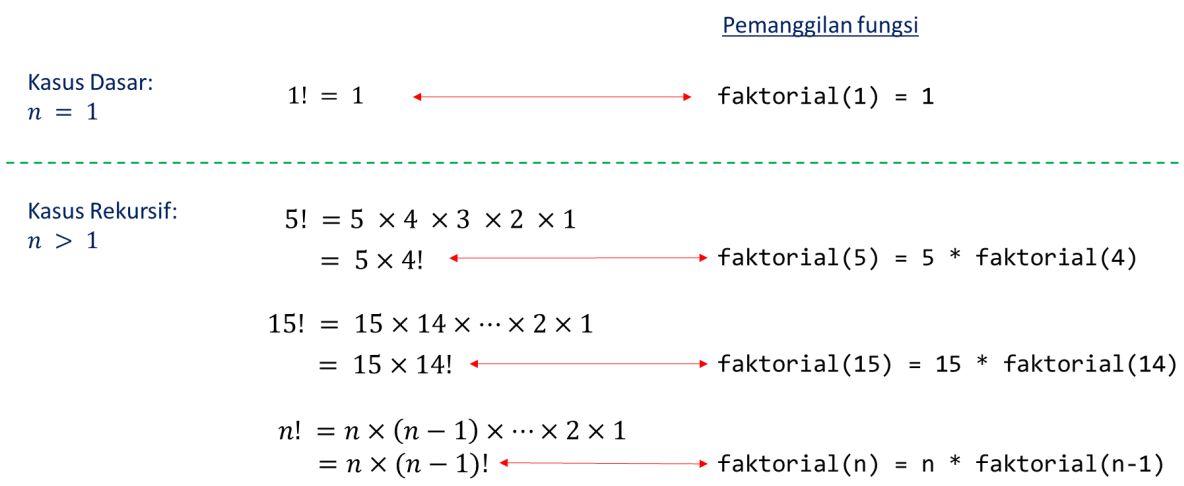
$$n! = n \times (n-1) \times (n-2) \times \ldots \times 3 \times 2 \times 1$$

Berikut adalah contoh-contoh faktorial:

- Faktorial 1: $1! = 1$
- Faktorial 5: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$
- Faktorial 15: $15! = 15 \times 14 \times \cdots \times 2 \times 1 = 1307674368000$

Kita dapat menghitung nilai faktorial suatu bilangan secara rekursif. Misalkan, faktorial dari 5 dapat dihitung dengan mengalikan 5 dengan faktorial 4, faktorial dari 15 dapat dihitung dengan mengalikan 15 dengan faktorial 14. Sehingga jika kita mendefinisikan sebuah fungsi `faktorial(n)` yang mencari nilai faktorial n, maka kita dapat mendefinisikannya secara rekursif dengan `faktorial(n) = n * faktorial(n-1)`. Ini adalah kasus rekursif (*recursive case*) dari `factorial(n)`. Selanjutnya kita memerlukan kasus dasar (*base case*) untuk `factorial(n)`. Dari definisi faktorial, kita mengetahui bahwa faktorial hanya didefinisikan untuk `n` lebih besar dari 0. Oleh karena ini, kasus dasar untuk rekursif dapat kita tentukan saat `n = 1`. Pada kasus dasar ini kita mengembalikan nilai dari faktorial ketika `n = 1`, yaitu `faktorial(1)` mengembalikan nilai 1.

Gambar berikut mengilustrasikan proses penentuan kasus dasar dan kasus rekursif dari fungsi `faktorial`:



Sehingga fungsi `faktorial` dapat kita definisikan sebagai berikut:

```
# Definisi fungsi faktorial
def faktorial(n):
    # Base case
    if n == 1:
        return 1
    # Recursive case
    else:
        return n * faktorial(n-1)
```

Program `faktorial.py` berikut mendemonstrasikan fungsi `faktorial`:

```
# faktorial.py
# Fungsi ini menghitung nilai faktorial
```

```python
# Definisi fungsi faktorial
def faktorial(n):
    # Base case
    if n == 1:
        return 1
    # Recursive case
    else:
        return n * faktorial(n-1)

# Fungsi main untuk menguji fungsi faktorial
def main():
    n = int(input('Masukkan nilai n: '))
    print(f'{n}! = {faktorial(n)}')

main()
```

Contoh *output* dari program `faktorial.py` di atas:

```
Masukkan nilai n: 5
5! = 120
```

```
Masukkan nilai n: 15
15! = 1307674368000
```

6.2.1.2 *Fibonacci*

Barisan *Fibonacci* didefinisikan sebagai barisan bilangan yang suku ke-n nya, dinotasikan dengan F_n, adalah hasil jumlah dari suku ke-$(n-1)$ dan suku ke-$(n-2)$, dimana $F_0 = 0$ dan $F_1 = 1$.

Suku ke-n dari barisan *Fibonacci* dituliskan dalam persamaan matematika sebagai berikut:

$$F_n = F_{n-1} + F_{n-2}$$

dengan $F_0 = 0$ dan $F_1 = 1$.

Barisan *Fibonacci* jika dituliskan dalam angka adalah sebagai berikut:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Pencarian suku ke-n dari barisan *Fibonacci* dapat diselesaikan secara rekursif dengan kasus dasar dan kasus rekursif seperti berikut:

- Kasus dasar:
  - $n = 0$ menghasilkan 0
  - $n = 1$ menghasilkan 1
- Kasus rekursif:
  - $n > 1$ dihitung dengan $F_n = F_{n-1} + F_{n-2}$

Kita dapat menuliskan fungsi yang mencari suku ke-n barisan *Fibonacci* sebagai berikut:

```
# Definisi fungsi Fibonacci
def fibo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibo(n - 1) + fibo(n - 2)
```

Program `fibonacci.py` di bawah mendemonstrasikan fungsi *Fibonacci*:

```
# fibonacci.py
# Program ini mendemonstrasikan barisan Fibonacci

# Definisi fungsi fibonacci
def fibo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibo(n - 1) + fibo(n - 2)

# Fungsi main untuk menguji fungsi Fibonacci
def main():
    n = int(input('Masukkan nilai n: '))
    f = fibo(n)
    print(f'Suku ke-{n} dari barisan Fibonacci: {f}')

main()
```

Output dari program `fibonacci.py` di atas:

```
Masukkan nilai n: 10
Suku ke-10 dari barisan Fibonacci: 55
```

```
Masukkan nilai n: 12
Suku ke-12 dari barisan Fibonacci: 144
```

6.2.2 Rekursif vs *Loop*

Perlu diketahui, semua persoalan yang dapat diselesaikan secara rekursif pasti dapat juga diselesaikan dengan struktur iterasi (*loop*).

Sebagai contoh, kode berikut adalah definisi fungsi faktorial dengan rekursif:

```
def faktorial(n):
    if n == 1:
        return 1
    else:
        return n * faktorial(n-1)
```

Fungsi faktorial dapat juga didefinisikan menggunakan struktur iterasi seperti terlihati pada kode berikut:

```
def faktorial(n):
    hasil = 1
    for i in range(1, num + 1):
        hasil = i * hasil
    return hasil
```

Kode berikut adalah definisi fungsi *Fibonacci* dengan rekursif:

```
def fibo(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibo(n - 1) + fibo(n - 2)
```

Fungsi *Fibonacci* dapat juga didefinisikan menggunakan struktur iterasi seperti kode berikut:

```
def fibo(n):
    a, b = 0, 1
    if n == 0:
        return a
    elif n == 1:
        return b
    else:
        for i in range(2, n + 1):
            suku_i = a + b
            a, b = b, suku_i
    return suku_i
```

Beberapa persoalan lebih mudah diselesaikan menggunakan pendekatan rekursif dibandingkan menggunakan *loop*. Rekursif umumnya lebih lambat dan lebih memakan *resource* komputer dibandingkan *loop*. Namun, dalam beberapa persoalan, pendekatan rekursif memudahkan logika program dan menjadikan program lebih mudah dibaca.

6.3 Algoritma Pencarian

Salah satu persoalan pemrograman yang paling umum adalah persoalan menentukan apakah suatu nilai terdapat di dalam suatu data. Sebagai contoh, misalkan kita ingin komputer menentukan apakah nilai 73 ada di dalam suatu barisan nilai berikut:

$$23, 34, 35, 43, 58, 65, 73, 81, 86, 90$$

Persoalan ini disebut persoalan pencarian dan terdapat dua algoritma dasar untuk persoalan pencarian:

1. Algoritma Pencarian *linear* (*Linear Search*)
2. Algoritma Pencarian biner (*Binary Search*)

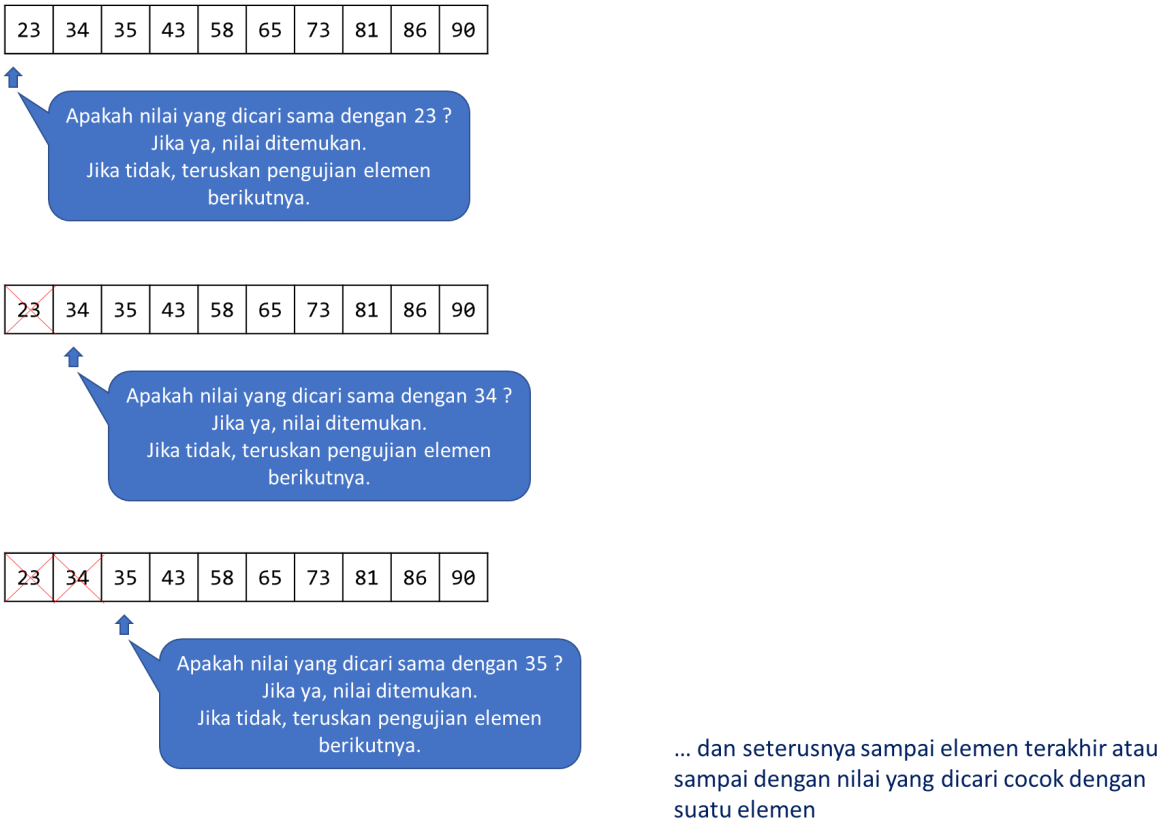
Kedua algoritma di atas bekerja pada suatu barisan data yang nilai-nilainya terurut.

6.3.1 Algoritma Pencarian *Linear*

Misalkan kita mencari nilai 73 dalam data yang disimpan dalam sebuah *list* terurut seperti berikut:

23	34	35	43	58	65	73	81	86	90

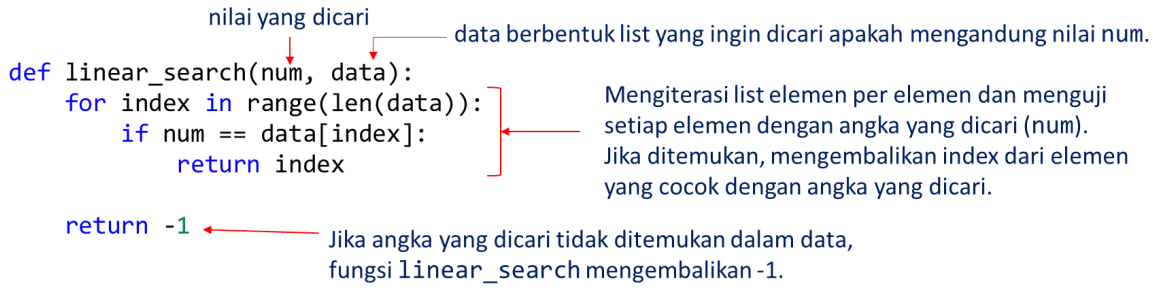
Algoritma pencarian *linear* bekerja dengan menguji setiap elemen pada data dengan nilai yang dicari secara berurut (dimulai dari elemen pertama, kedua dan seterusnya). Gambar berikut mengilustrasikan algoritma pencarian *linear*.



Kita dapat menuliskan algoritma pencarian *linear* :

```
def linear_search(num, data):
    for index in range(len(data)):
        if num == data[index]
            return index

    return -1
```

Fungsi `linear_search` melakukan pencarian nilai `num` (argumen pertama) pada sebuah *list* `data` (argumen kedua). Fungsi ini mengiterasi elemen-elemen *list* `data` dari elemen pertama sampai dengan elemen terakhir dan mencocokkannya dengan nilai `num`. Jika nilai `num` ditemukan dalam `data` maka fungsi ini mengembalikan nilai indeks dari lokasi ditemukannya nilai `num`. Sedangkan, jika `num` tidak ditemukan dalam `data`, maka fungsi ini mengembalikan nilai -1. Gambar berikut mengilustrasikan cara kerja fungsi `linear_search`.



Program berikut mendemonstrasikan algoritma pencarian *linear*:

```
# ... Definisi fungsi linear_search tidak ditulis disini.

# Fungsi main untuk mendemonstrasikan pencarian linear.
def main():
    num = 73
    data = [23, 34, 35, 43, 58, 65, 73, 81, 86, 90]

    index = linear_search(num, data)
    if index == -1:
        print('Data tidak ditemukan')
    else:
        print(f'Data ditemukan pada indeks ke-{index}')

main()
```

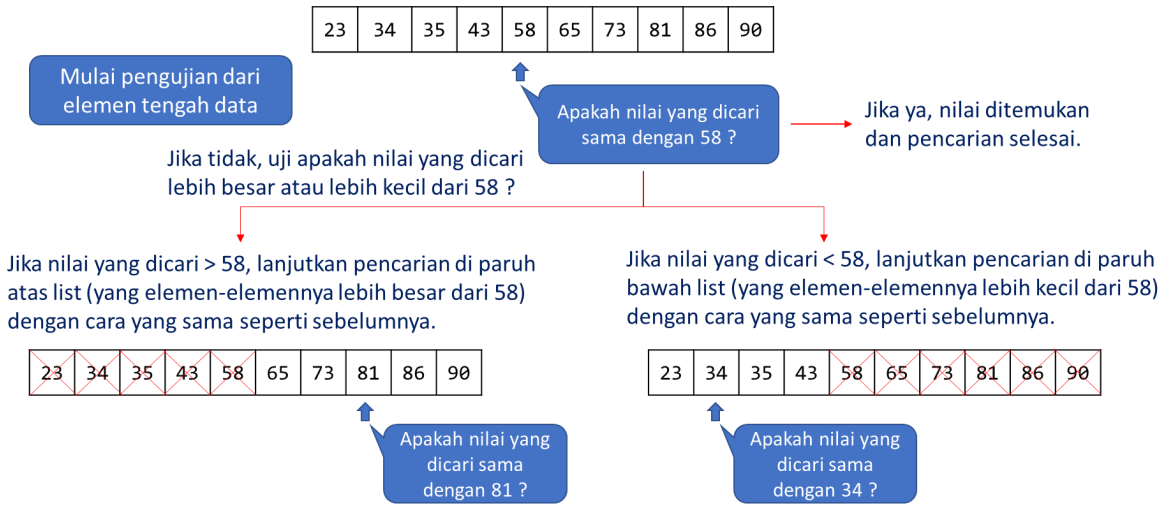
Output dari program di atas:

```
Data ditemukan pada indeks ke-6
```

Algoritma pencarian *linear* adalah algoritma pencarian paling sederhana namun algoritma ini paling tidak efisien. Jika kita mempunyai 99 angka pada data, algoritma ini dapat melakukan pengujian hingga 99 kali. Terdapat algoritma pencarian yang lebih efisien dari algoritma pencarian *linear* yaitu algoritma pencarian biner.

6.3.2 Algoritma Pencarian Biner

Ide dasar dari algoritma pencarian biner adalah membelah barisan data dan melakukan pencarian di salah satu paruhan dan mengulangi kembali pembelahan data pada paruhan tersebut sampai nilai yang dicari ditemukan atau tidak ditemukan. Pencarian biner dimulai dengan membandingkan nilai elemen tengah dari data dengan nilai yang dicari. Jika nilai yang dicari lebih kecil dari nilai pada elemen tengah maka pencarian dilanjutkan ke paruhan bawah. Sebaliknya, jika nilai yang dicari lebih besar dari nilai pada elemen tengah, maka pencarian dilakukan pada paruhan atas. Metode pencarian ini berulang sampai nilai ditemukan atau ketika data tidak dapat dibelah lagi yang berarti nilai tidak ditemukan. Gambar berikut mengilustrasikan algoritma pencarian biner yang mencari nilai 56 pada barisan nilai: 23, 34, 35, 43, 58, 65, 73, 81, 86, 90:



Penerapan algoritma pencarian biner dapat kita tuliskan dalam definisi fungsi berikut:

```
def binary_search(num, data):
    low = 0
    high = len(data) - 1

    while low <= high:
        mid = (low + high) // 2
        if num == data[mid]:
            return mid
        elif num < data[mid]:
            high = mid - 1
        else:
            low = mid - 1
    return -1
```

Fungsi `binary_search` melakukan pencarian biner nilai `num` pada sebuah *list* `data`. Variabel `low` digunakan untuk menyimpan indeks paling bawah dari paruhan-paruhan. Variabel ini diinisialisasi dengan nilai 0. Variabel `high` digunakan untuk menyimpan indeks paling atas dari paruhan-paruhan. Variabel `high` diinisialisasi dengan indeks paling atas *list* `data`, yaitu panjang *list* `data` dikurangi 1. Lalu, *loop* `while` dengan kondisi selama paruhan masih bisa dibelah membandingkan nilai elemen tengah paruhan dengan nilai yang dicari. Jika di dalam iterasinya, nilai tengah dari paruhan sama dengan nilai yang dicari, maka fungsi ini mengembalikan indeks dari nilai tengah tersebut. Jika nilai yang dicari lebih kecil dari nilai elemen tengah paruhan, maka variabel `high` ditetapkan ke indeks tengah sehingga pada iterasi berikutnya *loop* melakukan pencarian pada paruhan bawah. Jika nilai yang dicari lebih besar dari nilai tengah, maka variabel `low` ditetapkan ke indeks tengah sehingga pada iterasi berikutnya *loop* melakukan pencarian pada paruhan atas.

Kode berikut mendemonstrasikan algoritma pencarian biner:

```
# ... Definisi fungsi binary_search tidak dituliskan disini.

# Fungsi main mendemonstrasikan penggunaan fungsi binary_search
def main():
    num = 73
    data = [23, 34, 35, 43, 58, 65, 73, 81, 86, 90]

    index = binary_search(num, data)
    if index == -1:
        print('Data tidak ditemukan')
```

```
    else:
        print(f'Data ditemukan pada indeks ke-{index}')

main()
```

Output dari prorgram di atas:

```
Data ditemukan pada indeks ke-6
```

Algoritma pencarian biner lebih efisien dibandingkan algoritma pencarian linear karena maksimum banyaknya iterasi adalah sebanyak setengah dari banyaknya data sedangkan pada algoritma pencarian linear maksimum banyaknya iterasi adalah sebanyak dari banyaknya data.

6.4 Algoritma Penyortiran

Persoalan pemrograman dasar lain yang juga merupakan salah satu persoalan pemrograman yang umum adalah persoalan penyortiran data. Penyortiran daata yang dimaksud disini adalah pengurutan data berdasarkan nilainya dari yang terkecil hingga yang terbesar. Gambar berikut mengilustrasikan barisan data awal dan hasil barisan data setelah dilakukan penyortiran.

35	73	90	65	23	86	43	81	34	58

23	34	35	43	58	65	73	81	86	90

Terdapat dua algoritma penyortiran dasar yaitu:

1. *Selection sort*
2. *Quicksort*

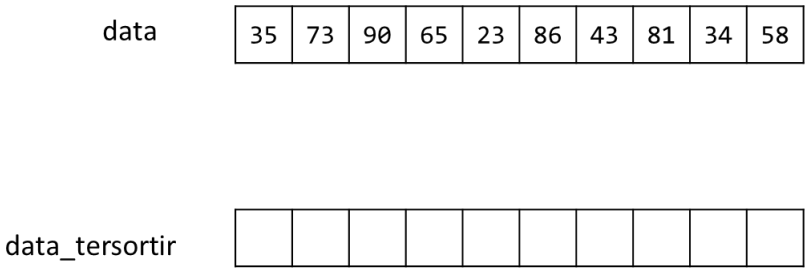
6.4.1 *Selection Sort*

Langkah-langkah dari *selection sort* adalah sebagai berikut:

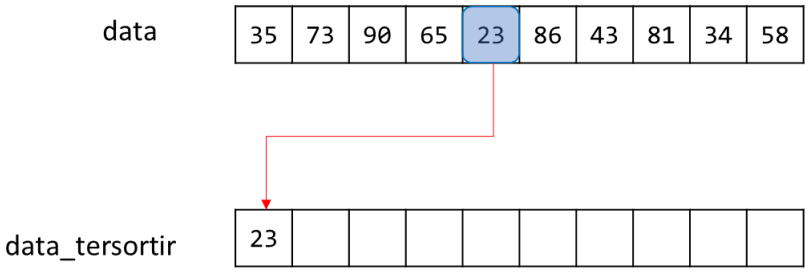
- Cari nilai terkecil pada data tak tersortir dan tempatkan nilai tersebut sebagai elemen pertama pada data tersortir. Lalu, hapus nilai terkecil tersebut dari data tak tersortir.
- Kemudian, cari kembali nilai terkecil pada data tak tersortir dan tempatkan nilai tersebut ke elemen berikutnya dari data tersortir.
- Ulangi langkah-langkah sebelumnya sampai semua elemen dari data disalin ke data tersortir.

Gambar berikut mengilustrasikan proses *selection sort*.

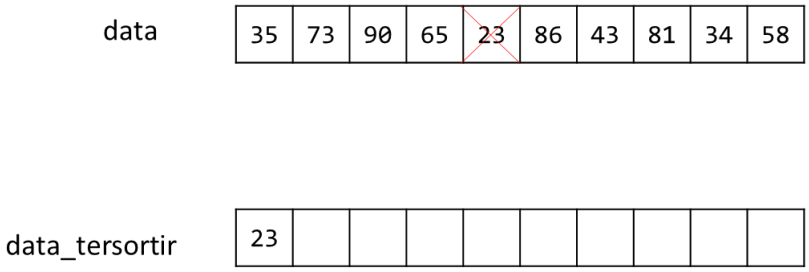
Mulai dengan list kosong untuk menyimpan data tersortir



Cari nilai terkecil pada data, lalu salin nilai tersebut ke elemen pertama dari data tersortir

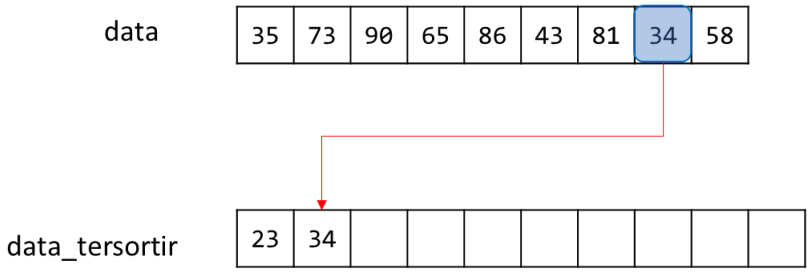


Hapus nilai terkecil pada data yang telah dipindahkan ke data tersortir

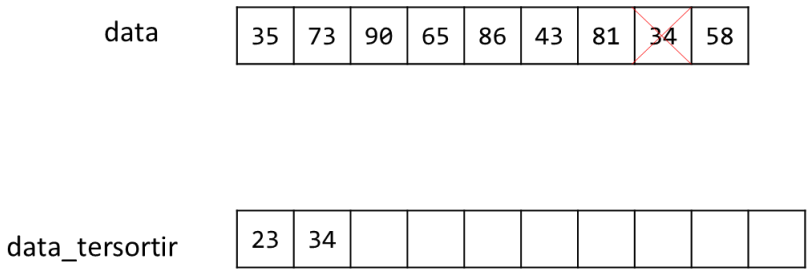


Cari kembali nilai terkecil pada data (yang telah dihapus nilai

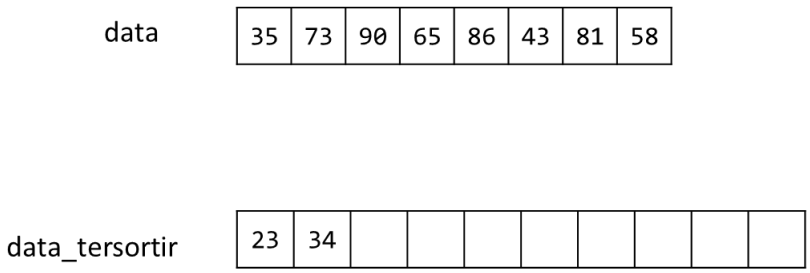
terkecil sebelumnya) lalu salin nilai tersebut ke elemen kedua dari data tersortir



Hapus nilai terkecil pada data yang telah dipindahkan ke data tersortir



... ulangi langkah-langkah sebelumnya sampai semua elemen dari data disalin ke data tersortir



Kita dapat menerapkan algoritma *selection sort* pada sebuah fungsi seperti berikut:

```
# Fungsi yang mencari indeks minimun dari data dalam list
def indeks_minimum(data):
    indeks_min = 0
    min = data[indeks_min]
```

```
    for indeks in range(1, len(data)):
        if data[indeks] <= min:
            indeks_min = indeks
            min = data[indeks_min]

    return indeks_min

# Fungsi yang mengimplementasikan algoritma selection sort
def selection_sort(data):
    data_tersortir = []
    for i in range(len(data)):
        indeks_min = indeks_minimum(data)
        data_tersortir.append(data[indeks_min])
        data.pop(indeks_min)

    return data_tersortir
```

Pada kode di atas kita mendefinisikan dua fungsi: fungsi `indeks_minimum` yang digunakan sebagai fungsi bantu untuk mencari indeks nilai terkecil dari sebuah *list* dan fungsi `selection_sort` yang melakukan proses penyortiran data dengan algoritma *selection sort*.

Program berikut mendemonstrasikan penggunaan fungsi `selection_sort`:

```
# ... Kode fungsi indeks_minimum dan fungsi selection_sort tidak dituliskan disini.

# Fungsi main untuk menguji selection sort
def main():
    data = [35, 73, 90, 65, 23, 86, 43, 81, 34, 58]
    data_tersortir = selection_sort(data)
    print(data_tersortir)

main()
```

Output dari program di atas:

```
[23, 34, 35, 43, 58, 65, 73, 81, 86, 90]
```

6.4.2 *Quicksort*

Algoritma *quicksort* adalah algoritma penyortiran yang jauh lebih cepat dibandingkan dengan algoritma *selection sort*. Algoritma *quicksort* umumnya diterapkan secara rekursif dengan kasus dasar (*base case*) dan kasus rekursif (*recursive case*) berikut:

- Kasus Dasar:
  - *List* kosong `[]`: *list* kosong tidak perlu disortir.
  - *List* satu elemen: *list* satu elemen tidak perlu disortir juga.
- Kasus Rekursif:
  - *List* dengan panjang lebih dari satu elemen.

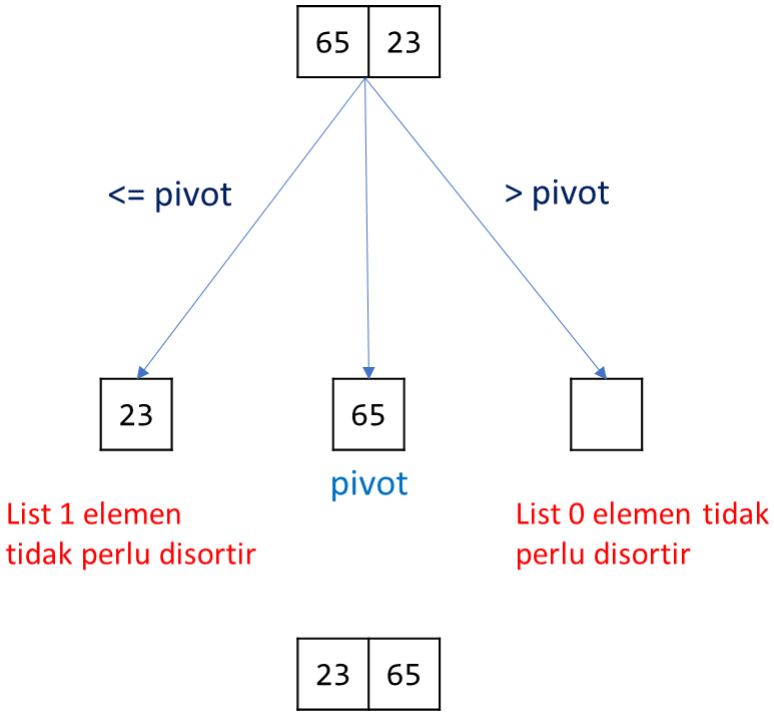
Langkah-langkah dari algoritma *quicksort* adalah sebagai berikut:

1. Jika *list* mempunyai elemen kurang dari dua elemen, tidak ada yang dilakukan.
2. Selain itu, pilih salah satu elemen dalam *list*. Elemen ini kita sebut sebagai *pivot*. Kita dapat memilih elemen manapun sebagai *pivot*. Untuk menyederhanakan proses, umumnya kita

memilih elemen pertama.

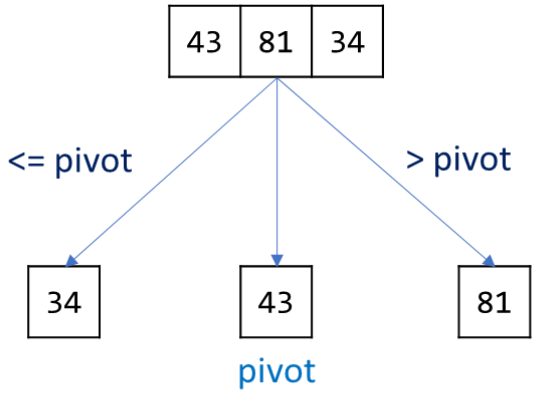
3. Partisi *list* dengan pertama-tama membuat dua buah *list* baru. Salah satu *list* digunakan untuk menampung nilai-nilai yang lebih kecil atau sama dengan nilai *pivot* dan *list* lainnya digunakan untuk menampung nilai-nilai yang lebih besar dari *pivot*. Tempatkan *list* dengan nilai-nilai lebih kecil atau sama dengan *pivot* di sebelah kiri *pivot* dan *list* dengan nilai-nilai lebih besar dari *pivot* di sebelah kanan *pivot*. Sehingga, kita mempunyai bentuk partisi seperti berikut: `[nilai-nilai < pivot], [pivot], [nilai-nilai > pivot]`.
4. Secara rekursif, sortir *list* dengan nilai-nilai yang lebih kecil atau sama dengan *pivot* dan *list* dengan nilai-nilai yang lebih besar dari *pivot*. Lalu, tempatkan *list* tersebut sesuai penempatannya dari kiri ke kanan.

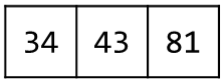
Langkah-langkah algoritma *quicksort* untuk *list* dengan dua elemen, sebagai contoh *list* `[65, 23]` dapat dilihat pada gambar berikut.



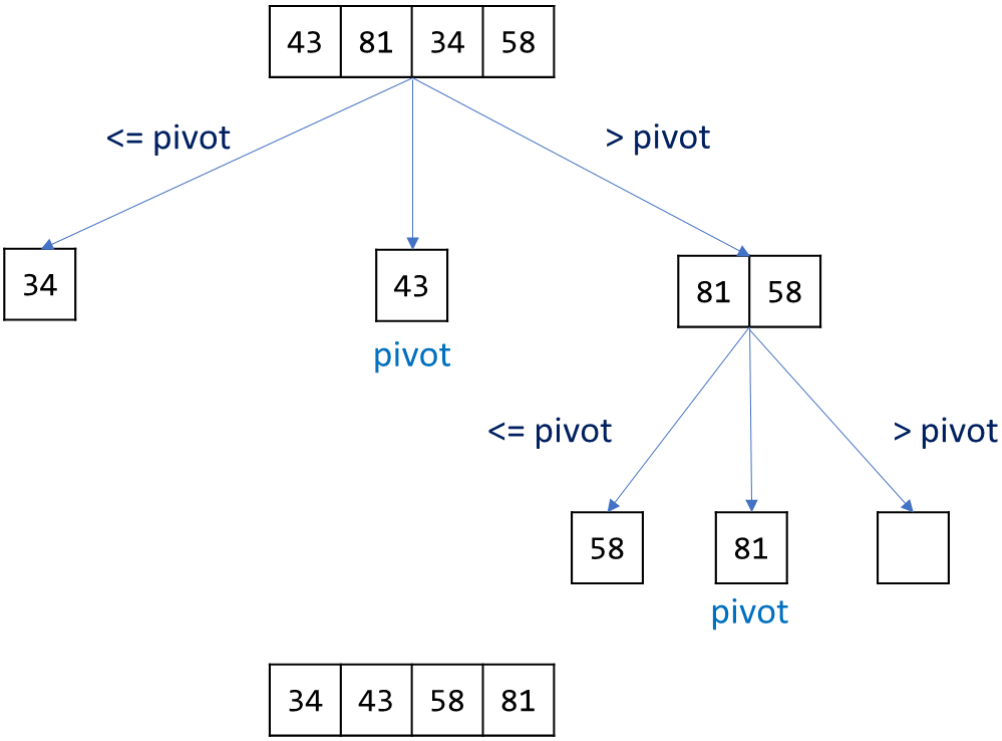
Pada gambar di atas, kita menetapkan nilai elemen pertama yaitu 65 sebagai nilai *pivot* sehingga *list* sebelah kiri akan hanya berisi satu nilai yaitu 23 dan *list* sebelah kanan tidak mempunyai elemen. Karena *list* sebelah kiri hanya berisi satu elemen dan *list* sebelah kanan tidak mempunyai elemen maka tidak ada langkah penyortiran kembali. Sehingga, *list* akan menghasilkan `[23, 65]`.

Langkah-langkah algoritma *quicksort* untuk *list* dengan tiga elemen, sebagai contoh `[43, 81, 34]` dapat dilihat pada gambar berikut.





Langkah-langkah algoritma *quicksort* untuk *list* dengan empat elemen, sebagai contoh `[43, 81, 34, 58]` dapat dilihat pada gambar berikut.

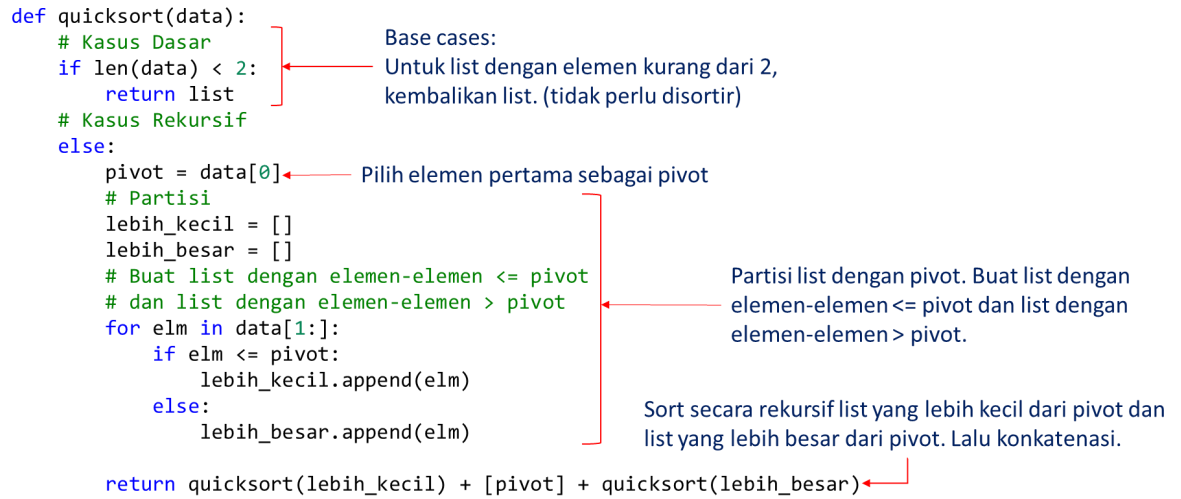


Algoritma *quicksort* dapat kita tuliskan pada sebuah definisi fungsi seperti berikut:

```
def quicksort(data):
    # Kasus Dasar
    if len(data) < 2:
        return data
    # Kasus Rekursif
    else:
        pivot = data[0]
        # Partisi
        lebih_kecil = []
        lebih_besar = []

        # Buat list dengan elemen-elemen <= pivot
        # dan list dengan elemen-elemen > pivot
        for elm in data[1:]:
            if elm <= pivot:
```

```
                lebih_kecil.append(elm)
            else:
                lebih_besar.append(elm)

        return quicksort(lebih_kecil) + [pivot] + quicksort(lebih_besar)
```

Penjelasan dari kode definisi fungsi `quicksort` di atas dapat dilihat pada gambar berikut.



Program berikut mendemonstrasikan penggunaan fungsi *quicksort*:

```
# ... Definisi fungsi quicksort tidak ditulis disini.

def main():
    data = [35, 73, 90, 65, 23, 86, 43, 81, 34, 58]
    data_tersortir = quicksort(data)
    print(data_tersortir)

main()
```

Output dari program di atas:

```
[23, 34, 35, 43, 58, 65, 73, 81, 86, 90]
```

6.5 Module

Dengan semakin membesarnya dan semakin kompleks program yang kita tulis, kita perlu cara untuk mengorganisasi program tersebut. Kita telah melihat bahwa kita dapat memecah program menjadi fungsi-fungsi yang masing-masing fungsi mengerjakan tugas spesifik tertentu. Lebih lanjut, kita dapat mengelompokkan fungsi-fungsi yang mengerjakan tugas-tugas terkait ke dalam suatu *module-module*. *Module* adalah sebuah *file* berekstensi `.py` yang berisi definisi-definisi fungsi. Pengorganisasian fungsi-fungsi ke dalam *module* ini disebut dengan modularisasi. Dengan modularisasi kita dapat menggunakan fungsi-fungsi dalam program-program berbeda, selain itu modularisasi juga membuat program akan lebih mudah dipahami, diuji, dan dikelola.

Misalkan kita membuat sebuah program yang menghitung

1. Luas dari sebuah lingkaran
2. Keliling dari sebuah lingkaran
3. Luas dari sebuah persegi panjang
4. Area dari sebuah persegi panjang

Kita dapat mengelompokkan tugas-tugas dari program kita menjadi dua *module*:

1. *Module* yang berisi fungsi-fungsi yang terkait dengan lingkaran (misalkan kita namakan dengan `lingkaran.py`)
2. *Module* yang berisi fungsi-fungsi yang terkait dengan persegi panjang (misalkan kita namakan dengan `persegi_panjang.py`)

Kita dapat menuliskan *module* lingkaran yang disimpan dalam file `lingkaran.py` dengan kode berikut:

```
# lingkaran.py
# Module lingkaran berisi fungsi-fungsi yang melakukan
# kalkulasi terkait lingkaran
import math

# Fungsi luas menerima radius (jari-jari) dari lingkaran sebagai argumen
# dan mengembalikan luas dari lingkaran
def luas(radius):
    return math.pi * radius ** 2

# Fungsi keliling menerima radius (jari-jari) dari lingkaran sebagai argumen
# dan mengembalikan keliling dari lingkaran
def keliling(radius):
    return 2 * math.pi * radius
```

Lalu, kita dapat menuliskan *module* persegipanjang yang disimpan dalam file `persegipanjang.py` dengan kode berikut:

```
# persegipanjang.py
# Module persegipanjang berisi fungsi-fungsi yang melakukan
# kalkulasi terkait persegipanjang

# Fungsi luas menerima lebar dan panjang sebagai argumen
# dan mengembalikan luas persegi panjang
def luas(lebar, panjang):
    return lebar * panjang

# Fungsi keliling menerima lebar dan panjang sebagai argumen
# dan mengembalikan keliling persegi panjang
def keliling(lebar, panjang):
    return 2 * (panjang + lebar)
```

Untuk menggunakan *module-module* yang telah kita buat dalam suatu program, kita harus meng*import*-nya dengan *statement import* yang diikuti nama *module* (nama *file* tanpa ekstensi):

```
import <nama_module>
```

Misalkan, untuk meng*import module* lingkaran, kita menuliskan statement `import` berikut:

```
import lingkaran
```

Setelah meng*import module*, kita dapat memanggil fungsi dalam *module* menggunakan notasi *dot*. Misalkan untuk memanggil fungsi `luas` dari *module* lingkaran dan menugaskan nilai kembali fungsi `luas` tersebut ke suatu variabel, kita menuliskan:

```
luas_lingkaran = lingkaran.luas(radius)
kel_lingkaran = lingkaran.keliling(radius)
```

Dua hal yang perlu diperhatikan dalam membuat *module*:

1. *Module* harus disimpan dalam *file* berekstensi `.py` (jika tidak berekstensi `.py`, kita tidak akan dapat meng*import*-nya ke program)
2. Nama *module* tidak boleh sama dengan *keyword* Python

Program berikut mendemonstrasikan penggunaan *module-module* yang telah kita tulis sebelumnya:

```
# geometri.py
# Program ini meminta pengguna untuk memilih kalkulasi
# geometri dari sebuah menu. Program mengimport module
# lingkaran dan persegipanjang
import lingkaran
import persegipanjang

# CONSTANT untuk pilihan menu
PILIHAN_LUAS_LINGKARAN = 1
PILIHAN_KELILING_LINGKARAN = 2
PILIHAN_LUAS_PERSEGIPANJANG = 3
PILIHAN_KELILING_PERSEGIPANJANG = 4
PILIHAN_KELUAR = 5

# Fungsi tampilkan_menu menampikan menu ke pengguna
def tampilkan_menu():
    print('MENU')
    print('1) Luas lingkaran')
    print('2) Keliling lingkaran')
    print('3) Luas persegi panjang')
    print('4) Keliling persegi panjang')
    print('5) Keluar')

# Fungsi main
def main():
    # variabel pilihan untuk
    # menyimpan pilihan pengguna
    pilihan = 0

    while pilihan != PILIHAN_KELUAR:
        # Tampilkan menu
        tampilkan_menu()
        # Minta pilihan pengguna
        pilihan = int(input('Masukkan pilihan Anda: '))
        # Lakukan perhitungan pilihan penguna
        if pilihan == PILIHAN_LUAS_LINGKARAN:
            radius = float(input('Masukkan radius lingkaran: '))
            print('Luas lingkaran adalah', lingkaran.luas(radius))
        elif pilihan == PILIHAN_KELILING_LINGKARAN:
            radius = float(input('Masukkan radius lingkaran: '))
            print('Keliling lingkaran adalah', lingkaran.keliling(radius))
        elif pilihan == PILIHAN_LUAS_PERSEGIPANJANG:
            lebar = float(input('Masukkan lebar persegi panjang: '))
            panjang = float(input('Masukkan panjang persegi panjang: '))
            print('Luas persegi panjang adalah',
```

```
                    persegipanjang.luas(lebar, panjang))
        elif pilihan == PILIHAN_KELILING_PERSEGIPANJANG:
            lebar = float(input('Masukkan lebar persegi panjang: '))
            panjang = float(input('Masukkan panjang persegi panjang: '))
            print('Keliling persegi panjang adalah',
                  persegipanjang.keliling(lebar, panjang))
        elif pilihan == PILIHAN_KELUAR:
            print('Keluar dari program...')
        else:
            print('Error: pilihan tidak valid.')

# Panggil fungsi main
main()
```

Contoh *output* dari program di atas:

```
MENU
1) Luas lingkaran
2) Keliling lingkaran
3) Luas persegi panjang
4) Keliling persegi panjang
5) Keluar
Masukkan pilihan Anda: 1
Masukkan radius lingkaran: 12.5
Luas lingkaran adalah 490.8738521234052
MENU
1) Luas lingkaran
2) Keliling lingkaran
3) Luas persegi panjang
4) Keliling persegi panjang
5) Keluar
Masukkan pilihan Anda: 4
Masukkan lebar persegi panjang: 24
Masukkan panjang persegi panjang: 35
Keliling persegi panjang adalah 118.0
MENU
1) Luas lingkaran
2) Keliling lingkaran
3) Luas persegi panjang
4) Keliling persegi panjang
5) Keluar
Masukkan pilihan Anda: 5
Keluar dari program...
```

REFERENSI

[1] Gaddis, Tony. 2012. *Starting Out With Python Second Edition*. United States of America: Addison-Wesley.