# Bab 1. Struktur Data

**OBJEKTIF**:

1. Mahasiswa mampu memahami Tipe Data dan Struktur Data (Abstraksi dan *Abstract Data Type*).
2. Mahasiswa mampu mengimplementasikan *Abstract Data Type* dan mendefinisikan ADT Bag menggunakan bahasa pemrograman Python.

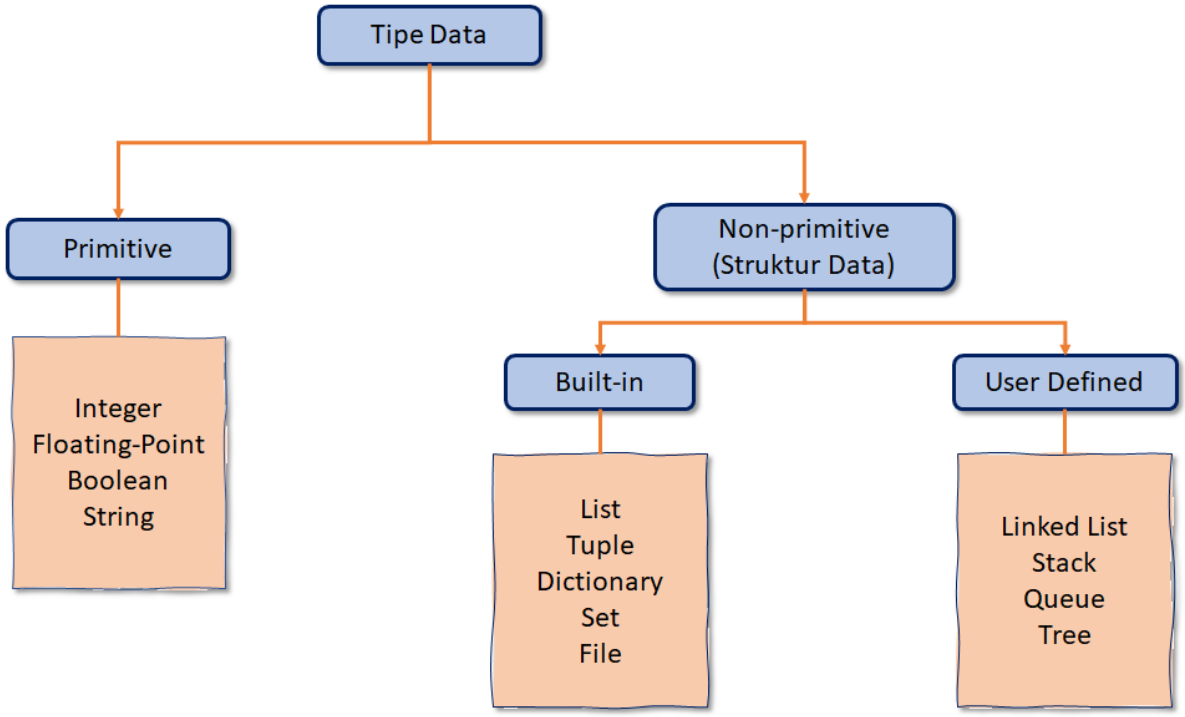
## 1.1 Tipe Data dan Struktur Data

Data direpresentasikan dalam komputer sebagai barisan digit-digit biner. Barisan-barisan digit biner ini dapat terlihat sama namun mempunyai arti berbeda karena komputer dapat menyimpan dan memanipulasi tipe-tipe data berbeda. Sebagai contoh, barisan biner `01001100110010110101011011011100` dapat berarti sebuah string karakter, bilangan bulat, atau bilangan riil. Tipe data yang kita tentukan dalam program  yang memberitahukan komputer bagaimana menerjemahkan barisan biner tersebut.

Bahasa pemrograman umumnya menyediakan sejumlah tipe-tipe data sebagai bagian dari bahasa itu sendiri. Tipe-tipe data ini disebut sebagai **tipe data *built-in*** dan dibagi menjadi dua kategori: ***primitive*** dan ***non-primitive***. Tipe data *primitive* terdiri dari nilai-nilai dalam bentuk paling dasar dan tidak dapat dipecah menjadi bagian-bagian yang lebih kecil. Tipe integer dan tipe floating point adalah contoh dari tipe data *primitive*. Sedangkan, tipe data *non-primitive* adalah tipe data yang dikonstruksi dari lebih dari satu tipe data *primitive* atau tipe data *non-primitive* lainnya. Dalam Python, list, tuple, dan dictionary, yang dapat berisi lebih dari satu nilai, adalah contoh dari tipe data *non-primitive*. Tipe data *non-primitive* disebut juga sebagai **struktur data**.

Tipe-tipe data *built-in* yang disediakan oleh bahasa pemrograman umumnya tidak cukup untuk menyelesaikan persoalan besar yang kompleks. Sehingga, sebagian besar bahasa pemrograman memungkinkan untuk mengkonstruksi tipe-tipe data tambahan, yang disebut sebagai **tipe data *user-defined*** (didefinisikan pengguna). Disebut dengan tipe data *user-defined* karena tipe-tipe data ini didefinisikan oleh programmer dan bukan oleh bahasa itu sendiri.

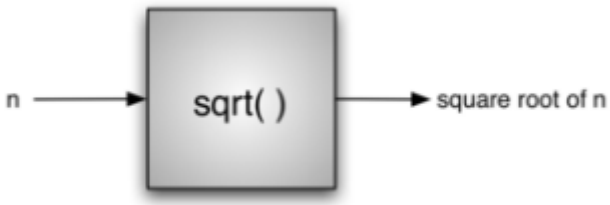
Gambar berikut mengilustrasikan pengklasifikasian tipe-tipe data dalam Python:



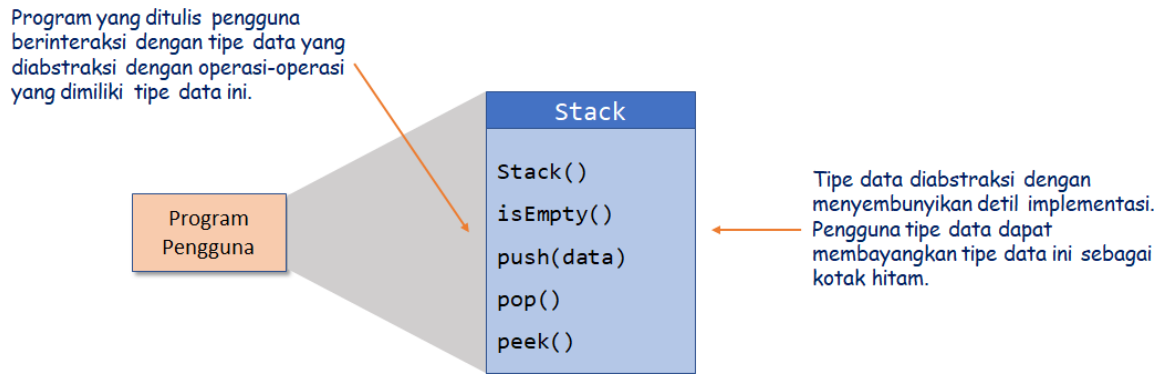

## Abstraksi

Untuk membantu mengelola persoalan kompleks dan tipe data kompleks, ilmuwan komputer biasanya bekerja dengan abstraksi. Abstraksi adalah mekanisme untuk menyembunyikan detail implementasi dari suatu objek yang kompleks sehingga pengguna dari objek tersebut berfokus pada cara penggunaan *object* dan tidak ke detail cara kerja objek tersebut. Di dalam ilmu komputer terdapat dua macam abstraksi yang umum: abstraksi prosedur dan abstraksi data.

Abstraksi prosedur adalah penggunaan fungsi atau *method* untuk menyembunyikan detail cara kerja suatu proses. Sebagai contoh, pada Python, terdapat fungsi-fungsi *built-in* yang sudah disertakan dalam bahasa Python, salah satunya adalah fungsi `sqrt()` yang menghitung akar kuadrat dari suatu nilai. Ketika kita menggunakan fungsi `sqrt` kita hanya perlu mengetahui bahwa fungsi tersebut menerima sebuah nilai dan menghasilkan akar kuadrat dari nilai yang diberikan. Kita tidak perlu mengetahui bagaimana langkah-langkah proses komputasi akar kuadrat di dalam fungsi tersebut. Ketika kita menggunakan fungsi `sqrt`, kita dapat membayangkannya sebagai kotak hitam yang menerima *input* berupa suatu nilai dan memberikan *output* berupa akar kuadrat dari nilai tersebut, seperti terlihat pada gambar berikut:




Sedangkan, abstraksi data adalah penyembunyian detail implementasi karakteristik dari suatu data. Karakteristik dari data meliputi bagaimana nilai dari tipe data tersebut distruktur dan operasi-operasi yang dapat dilakukan terhadap tipe data tersebut. Gambar berikut mengilustrasikan contoh abstraksi data:




Ketika kita membuat sebuah tipe data *user-defined* kita melakukan abstraksi data. Kita ingin pengguna tipe data yang kita buat hanya berfokus pada penggunaan tipe data termasuk operasi-operasi yang dapat dilakukan terhadap tipe data tersebut tanpa harus mengetahui bagaimana implementasi tipe data tersebut. Langkah awal dari membuat suatu tipe data *user-defined* adalah menuliskan deskripsi mengenai tipe data tersebut dan operasi-operasi yang dimilikinya. Deskripsi ini disebut dengan *Abstract Data Type* (ADT).

## Abstract Data Type

*Abstract Data Type* (ADT) diterjemahkan sebagai Tipe Data Abstrak adalah pendefinisian tipe data yang menjelaskan himpunan nilai-nilai data dan operasi-operasi yang dapat dilakukan terhadap nilai-nilai data tersebut. ADT harus diimplementasikan dalam program untuk dapat digunakan. Wujud nyata hasil implementasi ADT adalah struktur data.

Implementasi ADT dalam bahasa pemrograman yang mendukung *Object Oriented Programming* umumnya menggunakan class. ADT diimplementasikan menggunakan sebuah class dengan *field-field* yang diperlukan untuk menyimpan data dan mendefinisikan *method-method* dalam class tersebut sebagai operasi-operasi yang dapat dilakukan terhadap struktur data tersebut.

## Bahasa Pemrograman untuk Implementasi ADT

Untuk mengimplementasikan ADT menjadi struktur data kita perlu memilih sebuah bahasa pemrograman. Kita menggunakan Python karena Python adalah bahasa pemrograman *high-level* yang populer dan mendukung *Object Oriented Programming* (OOP). Selain itu, Python mempunyai syntax-syntax yang cenderung lebih mudah dibandingkan bahasa-bahasa pemrograman lain.

# 1.2 ADT Bag

Untuk mencontohkan mendesain dan mengimplementasi *Abstract Data Type* (ADT) kita mendefinisikan ADT Bag. *Bag* (tas) adalah *container* (wadah) sederhana seperti kantong belanja yang dapat diisi berbagai barang.

***Definisi ADT Bag***

Sebuah *bag* adalah wadah yang menyimpan sebuah koleksi nilai-nilai yang di dalamnya nilai-nilai duplikat diperbolehkan. Nilai-nilai dalam bag tidak mempunyai urutan tertentu tetapi masing-masing nilai dapat dibedakan satu sama lain. Operasi-operasi yang dimiliki ADT Bag:

- `Bag()` : Membuat sebuah bag dengan isi kosong.
- `length()` : Mengembalikan banyaknya elemen yang disimpan dalam bag. Diakses menggunakan fungsi `len()`.
- `contains(nilai)` : Menentukan apakah `nilai` berada dalam bag dan mengembalikan `true` jika ada dan `false` jika tidak ada. Diakses menggunakan operator `in`.
- `add(nilai)` : Menambahkan `nilai` ke bag.

- `remove(nilai)` : Menghapus `nilai` dari bag. Jika di dalam bag terdapat lebih dari satu `nilai` yang sama, hanya satu `nilai` yang dihapus. Jika tidak ada `nilai` dalam bag, maka sebuah eksepsi `ValueError` di-raise.
- `iterator()` : Membuat dan mengembalikan sebuah iterator yang dapat digunakan untuk mengiterasi nilai-nilai dalam bag. Diakses menggunakan *loop* `for`.

Kode berikut memperlihatkan contoh penggunaan ADT Bag:

```
# Membuat sebuah instance dari Bag
myBag = Bag()

# Operasi add(nilai) digunakan untuk menambahkan nilai-nilai ke bag.
myBag.add(19)
myBag.add(74)
myBag.add(23)
myBag.add(19)
myBag.add(12)

# Operasi length() digunakan untuk mengetahui berapa banyak data dalam bag.
# Operasi ini diakses menggunakan fungsi len().
print('Banyak data di dalam bag:')
print(len(myBag))

# Operasi remove(nilai) digunakan untuk menghapus nilai tertentu dari bag.
myBag.remove(74)    # Menghapus nilai 74 dari bag
myBag.remove(19)    # Menghapus salah satu nilai 19 dari bag


nilai = int(input("Tebak nilai yang disimpan dalam bag: "))

# Operasi contains() digunakan untuk mengetahui apakah suatu nilai berada dalam
bag.
# Operasi ini diakses dengan operator in.
if nilai in myBag:
    print("Bag berisi nilai", nilai)
else:
    print("Bag tidak berisi nilai", nilai)


# Operasi iterator() diguankan untuk meng-traverse atau mengiterasi
# semua nilai-nilai dalam bag.
# Operasi ini diakses dengan loop for.
for elm in myBag:
    print(elm)
```

## Implementasi ADT *Bag*

*Abstract Data Type* pada bahasa pemrograman yang mendukung OOP umumnya diimplementasikan dalam class. Karena Python mendukung OOP, maka kita akan mengimplementasi semua ADT yang akan kita buat menggunakan class. Untuk mengimplementasikan ADT Bag, kita mendefinisikan sebuah class bernama `Bag`:

```
class Bag:
    # ... Definisi method-method
```

Definisi class ini akan kita tuliskan dalam sebuah *module* bernama `bag` (dalam file `bag.py`).

## Constructor `Bag()`

Pada Python, *method* `__init__` adalah *method constructor* yang dipanggil otomatis ketika *object* dibuat. Pada *method* `__init__`, kita mendefinisikan *field-field* apa saja yang dimiliki oleh *object* dari class `Bag` dan menginisialisasi *field-field* tersebut. Class `Bag` hanya memerlukan satu *field* yang digunakan untuk menampung semua nilai-nilai yang berada dalam bag. Untuk mendefinisikan *field* yang dapat menampung nilai-nilai, kita menggunakan struktur data `list` yang telah tersedia dalam Python. Kita menamakan *field* ini dengan nama `_isi` dan *field* tersebut kita inisialisasi dengan `list` kosong. Sehingga kita dapat menuliskan *method* `__init__` dari class `Bag` seperti berikut:

```
def __init__(self):
    self._isi = list()
```

> **Catatan.** Kita menamakan *field* dari class dengan awalan karakter *underscore* (`_`) untuk menandakan bahwa *field* tersebut adalah *private*. Ini untuk memenuhi kaidah *encapsulation* pada OOP.

## Method `length()`

Python menyediakan fungsi *built-in* `len()` yang digunakan untuk mendapatkan banyaknya elemen dalam tipe data koleksi. Untuk membuat ADT Bag bekerja dengan fungsi `len()` kita harus mendefinisikan *method* spesial `__len__`. *Method* `__len__` mengembalikan banyaknya nilai-nilai dalam bag.

Karena class `Bag` mempunyai *field* `_isi` yang berupa `list` dan fungsi `len()` juga bekerja terhadap `list`, kita dapat menggunakan fungsi `len` dengan argumen field `_isi` untuk mendapatkan banyaknya elemen dalam Bag. Sehingga, *method* `__len__` dapat dituliskan seperti berikut:

```
def __len__(self):
    return len(self._isi)
```

## Method `add(nilai)`

Method `add` digunakan untuk menambahkan sebuah nilai ke bag. *Method* ini menerima sebuah argumen berupa nilai yang ingin dimasukkan ke bag. *Method* ini tidak mengembalikan nilai.

Untuk mengimplementasi *method* `add` kita dapat menggunakan *method* `append()` dari `list`, sehingga kode implementasi dari *method* `add` dapat dituliskan seperti berikut:

```
def add(self, nilai):
    return self._isi.append(nilai)
```

## Method `remove(nilai)`

*Method* `remove` menghapus sebuah nilai dari bag. *Method* ini menerima sebuah argumen berupa nilai yang ingin dihapus dari bag. Jika nilai yang ingin dihapus tidak terdapat di dalam bag, maka *method* ini meng-*raise* eksepsi `ValueError`. Jika di dalam bag terdapat lebih dari satu nilai yang sama dengan nilai yang ingin dihapus, *method* ini menghapus salah satu nilai.

Kita dapat mengimplementasikan *method* ini dengan pertama dengan menggunakan *method* `index` pada list yang menyimpan nilai-nilai bag. *Method* `index` ini melakukan pencarian terhadap sebuah nilai dan mengembalikan indeks dari elemen pertama yang ditemukan yang mempunyai nilai sama dengan nilai yang dicari. Setelah mendapatkan indeks dari nilai yang dicari, kita dapat menggunakan *method* `pop` dari list untuk menghapus nilai tersebut dari list.

Kode berikut adalah implementasi *method* `remove`:

```
def remove(self, nilai):
    if nilai not in self._isi:
        raise ValueError('Nilai tidak ada dalam bag.')
    else:
        indeks = self._isi.index(nilai)
        return self._isi.pop(indeks)
```

## Method `contains(nilai)`

Python menyediakan operator `in` untuk mencari suatu nilai di dalam suatu struktur data. Operator `in` mengembalikan nilai Boolean `True` jika nilai ditemukan dan mengembalikan nilai Boolean `False` jika nilai tidak ditemukan. Untuk membuat class `Bag` dapat bekerja dengan operator `in` kita harus mendefinisikan *method* spesial `__contains__` dengan sebuah parameter berupa nilai yang diuji terhadap nilai-nilai di dalam bag. Kita dapat menuliskan header dari *method* `__contains__` seperti berikut:

```
def __contains__(self, nilai):
```

Pada *header* di atas kita menamakan parameter untuk menyimpan nilai yang dicari dengan nama `nilai`.

*Method* `__contains__` mengembalikan `true` jika nilai yang dicari ditemukan dan mengembalikan `false` jika tidak ditemukan. Karena isi dari Bag disimpan dalam sebuah `list` dan operator `in` juga dapat digunakan terhadap `list`, kita dapat menuliskan *statement* `return` dari *method* `__contains__` dengan eksrepsi: `data in self._isi`. Sehingga, kode lengkap *method* `__contains__` dapat kita tuliskan seperti berikut:

```
def __contains__(self, nilai):
    return nilai in self._isi
```

## Method `iterator()`

*Traversal* adalah proses mengunjungi setiap elemen-elemen dari struktur data. Proses *traversal* ini sangat umum dilakukan pada struktur data. Pada struktur data yang disertakan oleh Python, `list`, `string`, `tuple`, dan `dictionary`, proses *traversal* dilakukan dengan menggunakan *loop* `for`. Sebagai contoh:

```
myList = [2, 4, 7, 8, 10]
for elm in myList:
    print(myList)
```

*Loop* `for` di atas meng-*traverse* list `myList` dan mencetak nilai dari setiap elemen dari `myList`.

Untuk menggunakan mekanisme *traversal* pada Python pada ADT Bag, kita harus mendefinisikan sebuah class iterator dengan dua *method* spesial: `__iter__` dan `__next__`. Implementasi class iterator untuk ADT Bag, kita namakan dengan `_BagIterator` dapat dituliskan seperti berikut:

```
class _BagIterator:
    def __init__(self, isi):
        self._isiBag = isi
        self._curIndeks = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._curIndeks < len(self._isiBag):
            item = self._isiBag[self._curIndeks]
            self._curIndeks += 1
            return item
        else:
            raise StopIteration
```

Pada *constructor* atau *method* `__init__` dari class `_BagIterator`, kita mendefinsikan dua *field*. *Field* pertama adalah referensi ke `list` yang digunakan untuk menyimpan isi dari bag, dan *field* kedua digunakan sebagai variabel indeks *loop* yang digunakan saat iterasi terhadap `list`. Kita menamakan *field* pertama dengan `_isiBag` dan *field* kedua dengan `_curIndeks`. Field variabel indeks *loop*, `curIndeks` kita inisialisasi ke 0 untuk membuat iterasi pertama dari *loop* dimulai dari indeks ke-0 dari `list`.

*Method* `__iter__` pada class `_BagIterator` hanya perlu untuk mengembalikan sebuah referensi ke *object* dari class ini.

*Method* `__next__` dipanggil setiap iterasi. *Method* ini mengembalikan nilai dari elemen dengan indeks dari variabel indeks *loop* dan memajukan variabel indeks *loop* ke indeks dari elemen berikutnya. Pada *method* `__next__` kita juga harus meng-*raise* eksepsi `StopIteration` setelah semua elemen di-iterasi. Eksepsi ini digunakan oleh *loop* untuk menghentikan iterasi.

Setelah menuliskan class `_BagIterator`, kita harus menuliskan *method* `__iter__` pada class `Bag` seperti berikut:

```
def __iter__(self):
    return _BagIterator(self._isi)
```

*Method* ini, membuat dan mengembalikan sebuah *instance* dari class `_BagIterator`. *Method* ini dipanggil secara otomatis pada awal dari *loop* `for` untuk membuat *object* iterator yang digunakan pada *loop* tersebut.

## Kode Lengkap Impelementasi ADT Bag

Berikut adalah kode lengkap implementasi ADT Bag:

***Module*** *`bag.py`*

```
# Class Bag adalah implementasi dari ADT Bag
class Bag:
    # Constructor ADT Bag.
```

```
    # Mendefinisikan field _isi berupa list untuk menyimpan
    # nilai-nilai dalam bag.
    def __init__(self):
        self._isi = list()

    # Method length() mengembalikan banyaknya data dalam bag.
    # Method ini diakses dengan fungsi built-in len()
    def __len__(self):
        return len(self._isi)

    # Method contains(nilai) mencari apakah suatu nilai terdapat dalam bag.
    # Method ini mengembalikan True jika nilai terdapat dalam bag
    # dan False jika nilai tidak terdapat dalam bag.
    # Method ini diakses dengan operator in.
    def __contains__(self, nilai):
        return nilai in self._isi

    # Method add() digunakan untuk menambahkan nilai data ke bag.
    # Method ini tidak mengembalikan nilai.
    def add(self, nilai):
        self._isi.append(nilai)

    # Method remove() digunakan untuk menghapus suatu nilai dalam bag.
    # Jika nilai yang ingin dihapus tidak ada di dalam bag, method ini
    # meng-raise eksepsi ValueError. Jika nilai yang ingin dihapus
    # ada di dalam bag, method ini menghapus salah satu nilai (jika terdapat
lebih
    # dari satu nilai.)
    # Method ini mengembalikan nilai yang dihapus.
    def remove(self, nilai):
        if nilai not in self._isi:
            raise ValueError('Nilai tidak ada di bag.')
        else:
            index = self._isi.index(nilai)
            return self._isi.pop(index)

    # Method iterator() digunakan untuk meng-traverse atau meng-iterasi setiap
nilai
    # dalam bag. Method iterator() mengembalikan class iterator.
    def __iter__(self):
        return _BagIterator(self._isi)

# Class _BagIterator adalah class iterator untuk membuat sebuah object
# dapat diiterasi.
class _BagIterator:
    # Constructor
    # Field 1: struktur data yang diiterasi
    # Field 2: variabel indeks loop
    def __init__(self, isi):
        self._isiBag = isi
        self._curIndeks = 0

    # Method __iter__ mengembalikan object dari class iterator.
    def __iter__(self):
        return self

    # Method _next_ mengembalikan nilai dari elemen dengan indeks
    # variabel indeks loop dan menginkrementasi variabel indeks loop
```

```
    def __next__(self):
        if self._curIndeks < len(self._isiBag):
            item = self._isiBag[self._curIndeks]
            self._curIndeks += 1
            return item
        else:
            raise StopIteration
```

## Menguji Implementasi ADT Bag

Kita dapat menguji implementasi ADT Bag menggunakan kode berikut:

```
from bag import Bag

# Membuat sebuah instance dari Bag
myBag = Bag()

# Operasi add(nilai) digunakan untuk menambahkan nilai-nilai ke bag.
myBag.add(19)
myBag.add(74)
myBag.add(23)
myBag.add(19)
myBag.add(12)

# Operasi length() digunakan untuk mengetahui berapa banyak data dalam bag.
# Operasi ini diakses menggunakan fungsi len().
print('Banyak data di dalam bag:')
print(len(myBag))

# Operasi remove(nilai) digunakan untuk menghapus nilai tertentu dari bag.
myBag.remove(74)    # Menghapus nilai 74 dari bag

nilai = int(input("Tebak nilai yang disimpan dalam bag: "))

# Operasi contains() digunakan untuk mengetahui apakah suatu nilai berada dalam
bag.
# Operasi ini diakses dengan operator in.
if nilai in myBag:
    print("Bag berisi nilai", nilai)
else:
    print("Bag tidak berisi nilai", nilai)

# Operasi iterator() diguankan untuk meng-traverse atau mengiterasi
# semua nilai-nilai dalam bag.
# Operasi ini diakses dengan loop for.
for elm in myBag:
    print(elm)
```

*Output* dari kode di atas:

```
Banyak data di dalam bag:
5
Tebak nilai yang disimpan dalam bag: 56
Bag tidak berisi nilai 56
19
23
19
12
```

**REFERENSI**

[1] Necaise, Rance D. 2011. Data structures and algorithms using Python .