

BAB I

OPERATOR TERNARY

Operator ternary umumnya dianggap sebagai ekspresi kondisional pada Python. Operator ini melakukan evaluasi apakah sebuah kondisi bernilai true atau false. Operator ternary sudah menjadi bagian dari Python sejak versi 2.4. Operator ternary dapat dituliskan sebagai berikut:

```
condition_is_true if condition else  
condition_is_false
```

Pada dasarnya operasi ternary sama dengan ekspresi if-else normal.

```
condition_is_true if condition else  
condition_is_false  
  
if condition:  
    condition_is_true  
else:  
    condition_is_false
```

Dibawah ini contoh penggunaan operator ternary:

```
is_fat = True  
state = "fat" if is_fat else "not fat"
```

Operator ternary memungkinkan kita menguji suatu kondisi secara cepat dibandingkan jika menggunakan statement if secara multiline. Seringkali operator ternary sangat bermanfaat dan dapat membuat kode kita lebih kompak namun tetap dapat terkelola dengan baik. Selain dengan cara diatas, terdapat cara lain yang menggunakan tuple, namun cara ini tidak terlalu banyak digunakan. Dibawah ini formatnya:

```
(if_test_is_false, if_test_is_true)[test]
```

Contoh penggunaanya:

```
fat = True
fitness = ("skinny", "fat")[fat]
print("Ali is ", fitness)
# Output:
# Ali is fat
```

Cara kerja kode diatas sangatlah sederhana. Karena True == 1 dan False == 0, maka kode diatas dapat diterjemahkan menjadi:

```
fitness = ("skinny", "fat")[1]
fitness = "fat"
```

Contoh diatas tidak terlalu banyak digunakan dan tidak disukai oleh programmer Python karena tidak Pythonic (tidak sesuai dengan kaidah Python). Selain itu, terkadang statement ini membingungkan dan sering terbalik saat meletakkan nilai saat true dan false. Alasan lain untuk tidak menggunakan tupled ternary adalah kedua elemen dari tuple dievaluasi, sementara if-else ternary tidak.

```
condition = True
print(2 if condition else 1/0)
# Output:
# 2

condition = True
print((1/0, 2)[condition])
# Output:
# ZeroDivisionError: division by zero
```

Kondisi error diatas dikarenakan pada tupled ternary, tuple akan dibentuk terlebih dahulu kemudian diakses dengan menggunakan index. Pada saat pembentukan tuple itu, baik pada kondisi nilai true maupun false, keduanya dievaluasi. Sementara pada if-else ternary,

eksekusi kondisi mengikuti logika if-else. Untuk itu, pada tupled ternary, jika salah satu kasus berpotensi menghasilkan error, atau keduanya merupakan kasus dengan komputasi tinggi, maka sebaiknya penggunaan tupled ternary dihindari.

BAB II

STRUKTUR DATA set

Set merupakan sebuah struktur data yang sangat penting pada Python. Set pada dasarnya sama seperti list dengan elemen unique, sehingga tidak bisa memiliki nilai duplikat. Set dapat didefinisikan dengan 2 (dua) macam cara, yaitu:

```
set(['yellow', 'red', 'blue', 'green', 'black'])
```

dan

```
{'yellow', 'red', 'blue', 'green', 'black'}
```

Set sangat bermanfaat pada banyak kasus. Misalnya kita ingin mencari tahu apakah pada sebuah list terdapat nilai-nilai yang berulang. Terdapat 2 (dua) solusi yang dapat digunakan untuk menyelesaikannya. Cara pertama dapat menggunakan for loop, seperti contoh berikut:

```
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
duplicates = []

for value in some_list:
    if some_list.count(value) > 1:
        if value not in duplicates:
            duplicates.append(value)

print(duplicates)
# Output:
# ['b', 'n']
```

Akan tetapi, kita bisa menggunakan cara yang lebih elegan dengan menggunakan set.

```
some_list = ['a', 'b', 'c', 'b', 'd', 'm', 'n', 'n']
duplicates = set([x for x in some_list if
some_list.count(x) > 1])
print(duplicates)
# Output:
# set(['b', 'n'])
```

Selain fungsi dasar untuk menghilangkan duplikasi pada sebuah list, set juga mempunyai beberapa method bawaan yang sangat bermanfaat. Dibawah ini beberapa diantaranya:

1. Intersection

Intersection memungkinkan pencarian elemen pada input set yang memiliki nilai yang sama dengan elemen pada set yang lain. Sebagai contoh:

```
valid = set(['yellow', 'red', 'blue', 'green', 'black'])
input_set = set(['red', 'brown'])
print(input_set.intersection(valid))
# Output:
# set(['red'])
```

Seperti terlihat pada contoh diatas, elemen-elemen pada input set yang memiliki nilai yang sama dengan elemen-elemen pada valid set adalah `red`.

2. Difference

Difference merupakan kebalikan dari intersection. Difference memungkinkan pencarian elemen-elemen yang tidak terdapat pada set yang lain.

```
valid = set(['yellow', 'red', 'blue', 'green',  
            'black'])  
input_set = set(['red', 'brown'])  
print(input_set.difference(valid))  
# Output:  
# set(['brown'])
```

BAB III

MUTASI

Tipe data mutable dan immutable pada Python seringkali membuat bingung programmer Python pemula. Apakah tipe data mutable dan immutable itu? Dalam bahasa sederhana, mutable dapat diartikan dapat berubah, dan immutable berarti tidak dapat berubah atau konstan. Agar lebih jelas, perhatikan contoh berikut:

```
foo = ['hi']
print(foo)
# Output: ['hi']

bar = foo
bar += ['bye']

print(foo)
# Output: ['hi', 'bye']

print(bar)
# Output: ['hi', 'bye']
```

Apakah ada yang aneh? Kita tidak melakukan perubahan apapun pada variabel `foo`, tetapi kenapa nilainya berubah? Kapan dia berubah? Bukankah seharusnya hasilnya seperti ini:

```
foo = ['hi']
print(foo)
# Output: ['hi']

bar = foo
bar += ['bye']

print(foo)
# Output: ['hi']

print(bar)
# Output: ['hi', 'bye']
```

Kondisi diatas bukanlah suatu kesalahan. Bukan juga sebuah bug. Kondisi tersebut dinamakan mutasi. Ketika kita meng-assign sebuah variabel kedalam variabel lain, dan tipe data dari variabel tersebut bersifat mutable, maka segala bentuk perubahan akan terekam pada kedua variabel. Variabel yang baru hanyalah merupakan alias dari variabel lama. Kondisi tersebut hanya berlaku pada tipe data yang bersifat mutable.

BAB IV

COLLECTIONS

Python telah dilengkapi dengan sebuah modul yang mengandung sejumlah tipe data yang disebut Collections. Tipe data yang tergolong kedalam Collections mempunyai sifat sebagai kontainer bagi tipe data yang lain. Tipe-tipe tersebut antara lain:

1. defaultdict
2. OrderedDict
3. counter
4. deque
5. namedtuple
6. enum.Enum (Python 3.4+)

3. defaultdict

Pada dasarnya, defaultdict mempunyai karakteristik yang sama dengan dict. Bedanya, dengan menggunakan defaultdict kita tidak perlu untuk mengecek apakah key-nya ada atau tidak. Perhatikan contoh berikut:

```
from collections import defaultdict

colours = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)

favourite_colours = defaultdict(list)
for name, colour in colours:
```

```

    favourite_colours[name].append(colour)

print(favourite_colours)
# Output:
# defaultdict(<class 'list'>, {'Yasoob': ['Yellow',
'Red'], 'Ahmed': ['Silver'], 'Arham': ['Green'],
'Ali': ['Blue', 'Black']})

```

Pada contoh diatas, `favourite_colours[name].append(colour)` tidak perlu memperdulikan apakah key `name` tersedia pada dict `favourite_colours`. Berbeda dengan jika kita menggunakan dict. Pada dict, jika key tidak tersedia maka akan muncul error `KeyError`.

```

colours = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)

favourite_colours = dict()
for name, colour in colours:
    favourite_colours[name].append(colour)
# Output:
# KeyError: 'Yasoob'

```

Sebenarnya, dict juga menyediakan suatu mekanisme untuk men-set nilai default ketika sebuah key tidak tersedia, yaitu `setdefault`. Akan tetapi, `defaultdict` dapat melakukannya secara otomatis.

```

colours = (
    ('Yasoob', 'Yellow'),
    ('Ali', 'Blue'),
    ('Arham', 'Green'),
    ('Ali', 'Black'),
    ('Yasoob', 'Red'),
    ('Ahmed', 'Silver'),
)

```

```

favourite_colours = dict()
for name, colour in colours:
    favourite_colours.setdefault(name, [])
    favourite_colours[name].append(colour)

print(favourite_colours)
# Output:
# {'Yasoob': ['Yellow', 'Red'], 'Arham': ['Green'],
# 'Ali': ['Blue', 'Black'], 'Ahmed': ['Silver']}

```

Sekarang, mari kita coba dengan studi kasus yang sedikit lebih kompleks, yaitu nested dict.

```

some_dict = dict()
some_dict['colours']['favourite'] = "yellow"
# Output:
# KeyError: 'colours'

```

Seperti yang telah diduga, kode diatas akan menghasilkan error, karena key 'colours' tidak tersedia pada variabel some_dict. Jika ingin tetap menggunakan dict, maka kodenya harus dimodifikasi seperti ini:

```

some_dict = dict()
some_dict.setdefault('colours', dict())
some_dict['colours']['favourite'] = "yellow"

print(some_dict)
# Output:
# {'colours': {'favourite': 'yellow'}}

```

Namun, bagaimana jika jumlah bersarang-nya tidak pasti? Untuk itu penggunaan defaultdict dapat cukup membantu. Seperti ini solusinya:

```

from collections import defaultdict

dict_tree = lambda: defaultdict(dict_tree)

```

```
some_dict = dict_tree()
some_dict['colours']['favourite'] = "yellow"
```

Untuk mencetak hasilnya, kita perlu mengkonversikannya dengan library json:

```
import json
print(json.dumps(some_dict))
# Output:
# {"colours": {"favourite": "yellow"}}
```

4. OrderedDict

Tipe data OrderedDict merupakan tipe data dict yang urutannya tersusun secara acak. Urutan pada saat kita melakukan insert dan pada saat melakukan retrieve terkadang bisa berubah, sehingga urutannya tidak bisa diprediksi. Contoh:

```
colours = {"Red" : 198, "Green" : 170, "Blue" : 160}
for key, value in colours.items():
    print(key, value)

# Output:
# Blue 160
# Red 198
# Green 170
```

OrderedDict dapat digunakan untuk mengatasi masalah ini.

OrderedDict menjaga agar urutan entri-nya sama dengan pada saat di-insert. Perubahan nilai tidak akan mengubah urutan dari entri. Namun penghapusan dan peng-insert-an ulang dapat mengubah urutan dari sebuah entri karena pada saat dihapus indeks dari entri tersebut juga turut terhapus.

```
from collections import OrderedDict
```

```
colours = OrderedDict([("Red", 198), ("Green",
170), ("Blue", 160)])
for key, value in colours.items():
    print(key, value)

# Output:
# Red 198
# Green 170
# Blue 160
```

Pada kode diatas, terlihat bahwa urutan dari entri terjaga dengan menggunakan OrderedDict.

5. counter

Counter memungkinkan kita untuk menghitung kemunculan dari item tertentu. Sebagai contoh, kita dapat menggunakannya untuk menghitung kemunculan dari warna favorit.

```
from collections import Counter

colours = ('Yellow', 'Blue', 'Green', 'Black',
'Green', 'Yellow')
favs = Counter(colours)
print(favs)
# Output:
# Counter({'Yellow': 2, 'Green': 2, 'Black': 1,
'Blue': 1})
```

Selain menggunakan input dari list dan tuple, counter juga dapat menggunakan input dari file, sehingga dapat digunakan untuk menghitung frekuensi kemunculan masing-masing kata.

```
with open('filename', 'rb') as f:
    line_count = Counter(f)
    print(line_count)
```

6. deque

Deque pada dasarnya mirip dengan list. Perbedaannya, pada list kita hanya dapat menambahkan dan menghapus isian dari arah kanan (belakang), sementara pada deque kita dapat melakukannya pada dua arah, kanan (belakang) dan kiri (depan). Untuk itulah deque disebut double ended queue.

```
from collections import deque

d = deque()
d.append('a')
d.append('b')
d.append('c')
d.append('d')

print('Deque: ' + str(d))
# Output: Deque: deque(['a', 'b', 'c', 'd'])
print('Length: ' + str(len(d)))
# Output: Length: 4
print('Left end: ' + d[0])
# Output: Left end: a
print('Right end: ' + d[-1])
# Output: Right end: d
```

Dengan deque, kita dapat menambahkan element dari sisi kanan maupun kiri.

```
d.append('e')
print('Deque: ' + str(d))
# Output: Deque: deque(['a', 'b', 'c', 'd', 'e'])

d.appendleft('f')
print('Deque: ' + str(d))
# Output: Deque: deque(['f', 'a', 'b', 'c', 'd', 'e'])
```

Kita dapat juga meng-extend data dari sisi kanan maupun kiri.

```

d.extend(['g', 'h'])
print('Deque: ' + str(d))
# Output: Deque: deque(['f', 'a', 'b', 'c', 'd',
'e', 'g', 'h'])

d.extendleft(['i', 'j'])
print('Deque: ' + str(d))
# Output: Deque: deque(['j', 'i', 'f', 'a', 'b',
'c', 'd', 'e', 'g', 'h'])

```

Juga mengeluarkan nilai dari sisi kanan maupun kiri.

```

v = d.pop()
print('Pop: ' + v)
# Output: Pop: h
print('Deque: ' + str(d))
# Output: Deque: deque(['j', 'i', 'f', 'a', 'b',
'c', 'd', 'e', 'g'])

v = d.popleft()
print('Pop: ' + v)
# Output: Pop: j
print('Deque: ' + str(d))
# Output: Deque: deque(['i', 'f', 'a', 'b', 'c',
'd', 'e', 'g'])

```

Selain itu, kita juga membatasi jumlah element dalam sebuah deque. Jika kita membatasi jumlah element, maka ketika jumlah element telah melebihi batas, element yang berada pada arah yang berlawanan dari element yang baru ditambahkan akan dikeluarkan.

```

d = deque(maxlen=5)
d.extend('abcde')
print('Deque: ' + str(d))
# Output: Deque: deque(['a', 'b', 'c', 'd', 'e'],
maxlen=5)

d.append('f')
print('Deque: ' + str(d))

```

```
# Output: Deque: deque(['b', 'c', 'd', 'e', 'f'],
maxlen=5)

d.appendleft('g')
print('Deque: ' + str(d))
# Output: Deque: deque(['g', 'b', 'c', 'd', 'e'],
maxlen=5)
```

7. namedtuple

Kita telah sama-sama tahu tentang tuple. Tuple pada dasarnya adalah list yang tidak dapat diubah nilainya, disebut juga immutable list. Untuk mengakses elemen dari tuple dan list kita dapat menggunakan index yang bertipe integer.

```
man = ('Ali', 30)
print(man[0])
# Output: Ali
```

Namedtuple pada dasarnya adalah tuple yang memungkinkan kita untuk mengakses setiap element-nya dengan menggunakan nama, bukan index seperti tuple. Sekilas memang namedtuple mirip dengan dictionary, tetapi namedtuple bersifat immutable atau tidak dapat diubah nilainya.

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")

print(perry)
# Output: Animal(name='perry', age=31, type='cat')
print(perry.name)
# Output: perry
```

Dengan namedtuple, sekarang kita dapat mengakses element-nya dengan menggunakan nama, sangat mirip dengan kita mengakses

class variable. Jika kita perhatikan, sebuah namedtuple membutuhkan 2 (dua) argumen pada saat pendefinisian. Argumen pertama adalah nama tuple, dan argumen kedua adalah nama-nama field pada tuple. Pada contoh diatas, nama tuple-nya adalah 'Animal' dan nama-nama field-nya adalah 'name', 'age', dan 'type'. Dengan menggunakan namedtuple kode kita akan lebih mudah dibaca dan lebih mudah dikelola. Selain itu, instance dari namedtuple tidak memiliki dictionary, sehingga mereka sangat ringan dan tidak menggunakan memory yang berbeda dibandingkan dengan tuple reguler. Namedtuple juga lebih cepat dibandingkan dictionary. Akan tetapi, yang perlu diingat kembali adalah namedtuple bersifat immutable, sehingga kode berikut tidak akan dapat dijalankan:

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")

perry.age = 42
# Output:
# AttributeError: can't set attribute
```

Keunggulan lain adalah namedtuple kompatibel dengan tuple normal. Artinya, selain dapat diakses dengan menggunakan nama field, namedtuple juga dapat diakses dengan menggunakan indeks-nya.

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")

print(perry[0])
# Output: perry
```

Ditambah lagi, kita juga dapat mengkonversikan namedtuple menjadi dictionary.

```
from collections import namedtuple

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="perry", age=31, type="cat")

print(perry._asdict())
# Output:
# OrderedDict([('name', 'perry'), ('age', 31),
# ('type', 'cat')])
```

8. enum.Enum (Python 3.4+)

Tidak seperti fungsi-fungsi diatas yang berada dalam modul collection, Enum terdapat dalam modul enum. Modul ini hanya tersedia mulai Python versi 3.4 dan yang lebih baru. Enum pada dasarnya adalah suatu kelas yang mempermudah dalam mengorganisasi berbagai hal.

Mari kita kembali sejenak ke namedtuple Animal pada poin 5. Pada contoh tersebut, terdapat field bernama 'type' dan bertipe string. Penggunaan tipe string tersebut berpotensi menimbulkan masalah. Bagaimana jika user mengetikkan 'Cat' atau 'CAT' atau 'kitten'? Enum dapat membantu untuk mengantisipasi masalah tersebut dengan mengorganisasikan field 'type' tersebut. Perhatikan contoh berikut:

```
from collections import namedtuple
from enum import Enum

class Species(Enum):
    cat = 1
    dog = 2
    horse = 3
    aardvark = 4
    butterfly = 5
    owl = 6
```

```

platypus = 7
dragon = 8
unicorn = 9
# The list goes on and on...

# But we don't really care about age, so we can
use an alias.
kitten = 1
puppy = 2

Animal = namedtuple('Animal', 'name age type')
perry = Animal(name="Perry", age=31,
type=Species.cat)
drogon = Animal(name="Drogon", age=4,
type=Species.dragon)
tom = Animal(name="Tom", age=75, type=Species.cat)
charlie = Animal(name="Charlie", age=2,
type=Species.kitten)

print(charlie.type)
# Output: Species.cat
print(tom.type)
# Output: Species.cat
print(tom.type == charlie.type)
# Output: True

```

Penggunaan Enums lebih sedikit berpotensi error dibandingkan string, karena seluruh kemungkinan telah didefinisikan.

Terdapat 3 (tiga) cara untuk mengakses anggota Enums, yaitu:

```

Species(1)
Species['cat']
Species.cat

```

BAB V

ENUMERATE

Enumerate adalah sebuah fungsi bawaan pada Python. Fungsi ini sangat sederhana, namun mempunyai fungsi yang sangat vital. Baik para pemula maupun programmer yang telah mahir seringkali mengabaikannya. Fungsi dari enumerate adalah untuk mendapatkan counter secara otomatis ketika kita melakukan looping terhadap sebuah iterable. Berikut contoh penggunaannya:

```
my_list = ['apple', 'banana', 'grapes', 'pear']
for counter, value in enumerate(my_list):
    print(counter, value)

# Output:
# 0 apple
# 1 banana
# 2 grapes
# 3 pear
```

Enumerate juga dapat mempunyai argumen kedua yang bersifat opsional, yaitu 'start'. Argumen 'start' mendefinisikan dari angka berapa counter akan dimula. Contoh:

```
my_list = ['apple', 'banana', 'grapes', 'pear']
for counter, value in enumerate(my_list, 1):
    print(counter, value)

# Output:
# 1 apple
# 2 banana
# 3 grapes
# 4 pear
```

Kita juga dapat membuat sebuah tuple yang berisi counter dan item sekaligus dengan menggunakan list.

```
my_list = ['apple', 'banana', 'grapes', 'pear']
counter_list = list(enumerate(my_list, 1))
print(counter_list)
# Output:
# [(1, 'apple'), (2, 'banana'), (3, 'grapes'), (4, 'pear')]
```

BAB VI

FOR-ELSE

Loop adalah bagian penting dari semua bahasa pemrograman, termasuk Python. Akan tetapi terdapat fitur spesifik yang dimiliki Python, yang sebagian besar programmer tidak mengetahuinya. Sebelumnya kita telah ketahui tentang dasar penggunaan for loop:

```
fruits = ['apple', 'banana', 'mango']
for fruit in fruits:
    print(fruit.capitalize())
# Output:
# Apple
# Banana
# Mango
```

Kode diatas adalah cara normal dalam menggunakan for loop. Bagaimana jika suatu for loop tidak menghasilkan apapun, dan kita ingin menambahkan perlakuan tertentu. Python menambahkan fitur 'else' pada 'for' untuk mengakomodir keperluan tersebut. Format for-else adalah sebagai berikut:

```
for item in container:
    if search_something(item):
        # Found it!
        process(item)
        break
else:
    # Didn't find anything..
    not_found_in_container()
```

Perhatikan contoh berikut:

```
for n in range(2, 10):
    for x in range(2, n):
        if n % x == 0:
            print(n, 'equals', x, '*', n/x)
```

```
break
```

```
# Output:  
# 4 equals 2 * 2.0  
# 6 equals 2 * 3.0  
# 8 equals 2 * 4.0  
# 9 equals 3 * 3.0
```

Bagaimana jika kita ingin mencetak n yang tidak habis dibagi x? Kita bisa tambahkan klausa 'else' pada 'for'.

```
for n in range(2, 10):  
    for x in range(2, n):  
        if n % x == 0:  
            print(n, 'equals', x, '*', n/x)  
            break  
  
    else:  
        print(n, 'is a prime number')  
  
# Output:  
# 2 is a prime number  
# 3 is a prime number  
# 4 equals 2 * 2.0  
# 5 is a prime number  
# 6 equals 2 * 3.0  
# 7 is a prime number  
# 8 equals 2 * 4.0  
# 9 equals 3 * 3.0
```

Kode diatas pada dasarnya adalah cara ringkas dari kode dibawah ini.

```
for n in range(2, 10):  
    is_prime = True  
    for x in range(2, n):  
        if n % x == 0:  
            print(n, 'equals', x, '*', n/x)  
            is_prime = False  
            break
```

```
if is_prime:
    print(n, 'is a prime number')

# Output:
# 2 is a prime number
# 3 is a prime number
# 4 equals 2 * 2.0
# 5 is a prime number
# 6 equals 2 * 3.0
# 7 is a prime number
# 8 equals 2 * 4.0
# 9 equals 3 * 3.0
```


BAB VII

COMPREHENSION

Comprehension adalah fitur yang tersedia pada Python, yang jika kita melewatkannya akan sangat disayangkan, karena akan sangat mempermudah dalam pembuatan kode. Comprehension adalah struktur yang memungkinkan sebuah sequence dibuat dari sequence yang lain. Terdapat 3 (tiga) buah comprehension yang dapat digunakan baik pada Python 2.x maupun Python 3.x:

1. List comprehension
2. Dictionary comprehension
3. Set comprehension

9. List Comprehension

List comprehension memungkinkan pembuatan list secara lebih pendek dan ringkas. Cara ini hanya terdiri dari square bracket ([]) yang berisi ekspresi yang diikuti dengan klausa for, kemudian opsional if. Bisa juga terdapat lebih dari satu for yang ditulis sebaris. Ekspresi dapat berupa apapun, yang artinya kita dapat memasukkan apapun kedalam list. Hasilnya berupa list yang baru yang dibuat dari hasil evaluasi ekspresi pada konteks if dan for yang lama. Format dari list comprehension dapat ditulis sebagai berikut:

```
variable = [out_exp for out_exp in input_list if
out_exp == something]
```

Berikut contoh sederhana dari list comprehension:

```
multiples = [i for i in range(30) if i % 3 == 0]
print(multiples)
# Output:
# [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

Kode diatas pada dasarnya sama dengan kode dibawah ini, akan tetapi lebih ringkas:

```
multiples = []
for i in range(30):
    if i % 3 == 0:
        multiples.append(i)

print(multiples)
# Output:
# [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

List comprehension akan sangat bermanfaat untuk membuat list dengan cepat. Cara ini sebenarnya juga memiliki fungsionalitas yang sama dengan fungsi 'filter', tetapi lebih disukai karena lebih mudah dibaca.

```
multiples = filter(lambda i: i % 3 == 0, range(30))
print(list(multiples))
# Output:
# [0, 3, 6, 9, 12, 15, 18, 21, 24, 27]
```

10. Dict Comprehension

Dict comprehension pada dasarnya mempunyai fungsi yang sama dengan list comprehension, yaitu membuat dict baru secara ringkas. Perbedaan dari list comprehension adalah pada penggunaan braces ({}), dan ekspresi yang digunakan harus berupa pasangan key: value. Secara umum formatnya mengikuti:

```
variable = {k: v for v in input_list if v == something}
```

Untuk membuatnya menjadi jelas, kita gunakan contoh sederhana berikut. Contoh berikut akan membuat dict yang berisi pasangan angka dengan pangkat-dua-nya:

```
squares = {i: i**2 for i in range(30) if i % 3 == 0}
print(squares)
# Output:
# {0: 0, 18: 324, 3: 9, 21: 441, 6: 36, 24: 576, 9: 81, 27: 729, 12: 144, 15: 225}
```

Contoh lain penggunaan dict comprehension adalah untuk menukar key dan value dari sebuah dict:

```
mcase = {'a': 10, 'b': 34, 'A': 7, 'Z': 3}
mcase_inv = {v: k for k, v in mcase.items()}
print(mcase_inv)
# Output:
# {34: 'b', 3: 'Z', 10: 'a', 7: 'A'}
```

11. Set Comprehension

Set comprehension sangat mirip dengan list comprehension. Perbedaannya hanya pada penggunaan braces (`{}`), sementara list comprehension menggunakan squared braces (`[]`). Berikut contohnya:

```
squared = {x**2 for x in [1, 1, 2]}
print(squared)
# Output:
# {1, 4}
```

BAB VIII

GLOBAL DAN RETURN

Kita mungkin telah sering menjumpai beberapa fungsi di Python mempunyai keyword 'return' pada bagian akhirnya. Jadi apa sebenarnya keyword 'return' ini? Return pada fungsi sebenarnya sama dengan 'return' yang terdapat pada bahasa pemrograman lain. Mari kita coba dengan sebuah fungsi sederhana berikut:

```
def add(value1, value2):  
    return value1 + value2  
  
result = add(3, 5)  
print(result)  
# Output:  
# 8
```

Fungsi diatas mengambil nilai dari kedua argumen sebagai input, kemudian mengembalikan nilai hasil penjumlahannya. Kita juga dapat melakukannya dengan cara lain dengan menggunakan global variabel.

```
def add(value1,value2):  
    global result  
    result = value1 + value2  
  
add(3,5)  
print(result)  
# Output:  
# 8
```

Pertama, mari kita bahas tentang kode pertama yang menggunakan kata kunci return. Pada kode tersebut, apa yang dilakukan oleh fungsi adalah meng-assign nilai ke variabel yang memanggil fungsi tersebut, yaitu variabel result. Sementara kode kedua yang menggunakan kata kunci global, apa yang dilakukan fungsi tersebut

adalah membuat variable global, yaitu result. Apa yang dimaksud dengan global? Variabel global memungkinkan kita mengakses variabel dari luar scope fungsi. Mari kita demonstrasikan dengan contoh. Pertama kita akan membuat kode tanpa variabel global:

```
def add(value1, value2):  
    result = value1 + value2  
  
add(2, 4)  
print(result)  
# Output:  
# NameError: name 'result' is not defined
```

Apakah kalian mendapati sebuah error? Kenapa bisa terjadi? Hal ini dikarenakan interpreter Python memberitahukan kepada kita bahwa kita tidak mempunyai variabel bernama result. Ini terjadi karena variable result hanya dapat diakses dari dalam fungsi add, dengan kata lain variabel result bersifat lokal dan bukan global. Sekarang mari kita jalankan lagi, tetapi dengan terlebih dahulu mendefinisikan variable result sebagai global:

```
def add(value1, value2):  
    global result  
    result = value1 + value2  
  
add(2, 4)  
print(result)  
# Output:  
# 6
```

Seperti terlihat, kode diatas tidak menghasilkan error. Jadi kata kunci global membuat sebuah variabel lokal dapat diakses secara global dari luar fungsi. Meskipun secara fakta kondisi ini dapat berjalan dengan baik, namun sebaiknya kita menghindari penggunaan kata kunci global, karena hanya akan menyusahkan ketika kita melibatkan banyak variabel global.

12. Multiple Return Value

Seperti yang telah kita tahu, beberapa fungsi membutuhkan untuk mengembalikan sebuah nilai. Namun, terkadang ada lebih dari satu nilai yang ingin kita kembalikan, apakah hal itu memungkinkan?

Pada Python, terdapat beberapa pendekatan yang dapat dilakukan jika ingin mengembalikan beberapa buah nilai sekaligus.

Cara pertama dengan menggunakan variabel global. Seperti kita telah pelajari sebelumnya, keyword global memungkinkan sebuah variabel yang didefinisikan didalam sebuah fungsi dapat diakses dari luar fungsi. Perhatikan contoh berikut:

```
def profile():
    global name
    global age
    name = "Soedomoto"
    age = 30

profile()
print(name)
# Output: Soedomoto
print(age)
# Output: 30
```

Cara diatas dapat digunakan sebagai pendekatan untuk mengembalikan lebih dari satu nilai. Namun, cara diatas sangat tidak direkomendasikan. Sebagian besar programmer Python menghindari pendekatan ini. Sebagai alternatif, kita dapat menggunakan kata kunci return. Berikut contohnya:

```
def profile_tuple():
    name = "Soedomoto"
    age = 30
    return (name, age)

profile_data = profile_tuple()
print(profile_data[0])
```

```
# Output: Soedomoto
print(profile_data[1])
# Output: 30
```

Pada kode diatas, fungsi `profile_tuple` mengembalikan nilai berupa tuple. Tuple memiliki fungsionalitas yang sama dengan list, hanya saja tuple tidak dapat diubah datanya. Atau dengan kata lain tuple bersifat read-only. Index pertama pada tuple tersebut berisi nilai dari `name` dan indeks kedua berisi nilai dari `age`. Cara lain yang lebih umum, dan menjadi konvensi pada pemrograman Python adalah dengan comma-separated-value. Perhatikan contoh berikut:

```
def profile_multiple():
    name = "Soedomoto"
    age = 30
    return name, age
```

Return value pada fungsi `profile_multiple` pada dasarnya sama dengan pada fungsi `profile_tuple`. Perbedaanya hanya pada penggunaan parentheses `()` pada return value-nya. Sebenarnya, dengan ataupun tanpa menggunakan parentheses, return value yang berupa comma-separated-value akan secara otomatis bertipe tuple. Berikut buktinya:

```
print(type(profile_tuple()))
# Output: <class 'tuple'>
print(type(profile_multiple()))
# Output: <class 'tuple'>
```

Sehingga, kita bisa perlakukan sebagai layaknya tuple atau list:

```
profile_data = profile_multiple()
print(profile_data[0])
# Output: Soedomoto
print(profile_data[1])
# Output: 30
```

Catatan: Cara diatas adalah contoh mengakses tuple dengan menggunakan indeks .Cara lain yang lebih Pythonic dalam mengakses sebuah tuple atau list adalah dengan menampungnya pada comma-separated-value, seperti contoh berikut:

```
name, age = profile_multiple()  
print(name)  
# Output: Soedomoto  
print(age)  
# Output: 30
```

Cara kedua yang menggunakan tuple atau comma-separated-value lebih direkomendasikan untuk fungsi yang mengembalikan lebih dari satu nilai. Ingat, jangan menggunakan variabel global, karena variable global akan menimbulkan kebingungan ketika kode yang kita buat semakin kompleks.

BAB IX

*ARGS DAN **KWARGS

Saya mendapati bahwa sebagian besar programmer python yang baru belajar mengalami kesulitan dalam memahami tentang variabel magic *args dan **kwargs. Jadi apakah sebenarnya *args dan **kwargs itu? Pertama sekali, saya akan menyampaikan bahwa kita tidak harus menuliskannya tepat sebagai *args dan **kwargs, tetapi kita dapat menggunakan nama yang lain sebagai variabel, misalnya *var dan **vars. Yang terpenting hanyalah jumlah * (asterisk), sementara *args dan **kwargs hanyalah konvensi penamaan. Jadi mari kita bahas satu per satu tentang *args dan **kwargs.

13. Fungsi *args

*args dan **kwargs umumnya digunakan pada definisi sebuah fungsi. *args dan **kwargs memungkinkan kita untuk dapat melemparkan sejumlah argumen pada saat memanggil sebuah fungsi. Sejumlah argumen yang dimaksudkan disini adalah argumen yang kita tidak ketahui jumlahnya (dinamis). Sehingga pada kasus ini kita menggunakan kedua kata kunci tersebut. *args digunakan untuk mengirimkan sejumlah variabel yang **tidak memiliki keyword** kedalam sebuah fungsi. Agar mudah dipahami, perhatikan contoh berikut:

```
def test_var_args(f_arg, *argv):
    print("first normal arg:", f_arg)
    for arg in argv:
        print("another arg through *argv:", arg)

test_var_args('yasoob', 'python', 'eggs', 'test')
```

Akan menghasilkan output sebagai berikut:

```
('first normal arg:', 'yasoob')
('another arg through *argv:', 'python')
('another arg through *argv:', 'eggs')
('another arg through *argv:', 'test')
```

Saya harap sebuah contoh diatas cukup dapat memperjelas apa itu *args. Jadi sekarang mari kita bahas apa itu **kwargs.

14. Fungsi **kwargs

kwargs memungkinkan kita untuk melempar sejumlah variable yang **memiliki keyword kedalam sebuah fungsi. Kita harus menggunakan keyword jika kita ingin menangani argumen yang memiliki nama yang telah ditentukan. Berikut contoh yang akan memperjelas keterangan diatas:

```
def greet_me(**kwargs):
    for key, value in kwargs.items():
        print("{0} = {1}".format(key, value))

greet_me(name="soedomoto")
```

Akan menghasilkan output sebagai berikut:

```
name = soedomoto
```

Dari contoh diatas, terlihat bagaimana kita menangani argumen yang memiliki **keyword** pada sebuah fungsi. Contoh diatas hanyalah cara penggunaan secara mendasar dan kita dapat menggunakannya secara lebih luas lagi. Sekarang mari kita bahas bagaimana kita dapat menggunakan *args dan **kwargs untuk memanggil sebuah fungsi dengan sejumlah *dictionary* dari argumen.

15. Menggunakan *args dan **kwargs Sekaligus Pada Fungsi

*args dan **kwargs dapat digunakan pada sebuah fungsi secara bergantian. Untuk memperjelasnya, perhatikan contoh berikut. Misalkan kita punya sebuah fungsi sederhana:

```
def test_args_kwargs(arg1, arg2, arg3):  
    print("arg1:", arg1)  
    print("arg2:", arg2)  
    print("arg3:", arg3)
```

Sekarang, kita dapat menggunakan *args atau **kwargs untuk mengirimkan beberapa argumen sekaligus kepada fungsi diatas, seperti berikut:

```
# Dengan *args  
args = ("dua", 5, 8)  
test_args_kwargs(*args)  
# Output:  
# ('arg1:', 'dua')  
# ('arg2:', 5)  
# ('arg3:', 8)  
  
# Dengan **kwargs  
kwargs = {"arg3": 3, "arg2": "two", "arg1": 5}  
test_args_kwargs(**kwargs)  
# Output:  
# ('arg1:', 5)  
# ('arg2:', 'two')  
# ('arg3:', 3)
```

Kemudian, bagaimana jika kita ingin menggunakan normal argumen, *args, dan **kwargs sekaligus? Bisa, caranya dengan mengikuti urutan yang telah ditentukan:

```
some_func(fargs, *args, **kwargs)
```

Contohnya:

```
def test_args_kwargs(arg, *args, **kwargs):
    print("arg :", arg)

    for _arg in args:
        print("arg :", _arg)

    for key, _arg in kwargs.items():
        print(key, ":", _arg)

# Dengan *args dan **kwargs
args = (5, 8)
kwargs = {"karg1": "satu", "kargs2": 2}
test_args_kwargs("dua", *args, **kwargs)
# Output:
# ('arg :', 'dua')
# ('arg :', 5)
# ('arg :', 8)
# ('kargs2', ':', 2)
# ('karg1', ':', 'satu')
```

BAB X

GENERATOR

Sebelum kita membicarakan tentang generator, mari kita mulai dengan iterator. Berdasarkan wikipedia¹, iterator adalah sebuah obyek yang memungkinkan seorang programmer untuk melalui (traverse) sebuah kontainer, terutama sebuah list. Akan tetapi, meskipun sebuah iterator melintasi elemen-elemen pada kontainer, iterator tidak melakukan pembacaan. Istilah ini mungkin sedikit membingungkan, jadi mari kita pelajari pelan-pelan.

Terdapat 3 buah istilah yang saling terkait satu sama lain, yaitu:

1. Iterable
2. Iterator
3. Iteration

16. Iterable

Sebuah iterable dalam python adalah segala bentuk obyek yang memiliki method `__iter__` atau `__getitem__`, yang mana kedua method tersebut mengembalikan nilai dari iterator atau index. Kemudian, apakah iterator itu?

17. Iterator

Sebuah iterator dalam python adalah segala bentuk obyek yang memiliki method `next` (python 2) atau `__next__` (python 3). Sederhana bukan? Lantas apa itu iteration?

¹ <https://en.wikipedia.org/wiki/Iterator>

18. Iteration

Dalam bahasa sederhana, iterasi adalah proses mengambil item atau elemen dari sesuatu, misalnya sebuah list. Ketika kita menggunakan sebuah pengulangan untuk mengakses isi dari sebuah list, itulah yang disebut dengan iterasi.

19. Generator

Generator pada dasarnya adalah iterator, akan tetapi kita hanya dapat melakukan iterasi sekali. Hal ini dikarenakan generator tidak menyimpan nilainya pada memory, tetapi menghasilkan nilai on the fly. Sehingga, sebuah generator tidak dapat diketahui jumlah elemennya sampai dengan tidak ada yang di-generate lagi. Generator pada umumnya diimplementasikan sebagai sebuah fungsi. Agar fungsi tersebut dapat berperan sebagai generator, alih-alih mengembalikan nilai, fungsi tersebut akan meneriakkannya (yield). Agar lebih jelas, mari kita lihat contoh berikut:

```
# normal iterator
def normal_function():
    ret_val = []
    for i in range(10):
        ret_val.append(i**2)

    return ret_val

for i in normal_function():
    print(i)

# Output:
# 0
# 1
# 4
# 9
# 16
# 25
```

```
# 36
# 49
# 64
# 81
```

```
# generator
def normal_function():
    for i in range(10):
        yield i**2

for i in normal_function():
    print(i)

# Output:
# 0
# 1
# 4
# 9
# 16
# 25
# 36
# 49
# 64
# 81
```

Contoh diatas merupakan contoh sederhana penggunaan generator. Sekilas memang tidak terlalu beda jauh dengan iterator. Memang, generator baru terasa manfaatnya jika digunakan untuk menghitung data yang besar, terutama jika hasil dari satu iterasi berpengaruh terhadap iterasi berikutnya dan jumlah iterasinya bersifat fleksibel. Sejumlah fungsi library standar pada python 3 telah dimodifikasi untuk mengembalikan generator alih-alih mengembalikan list karena generator membutuhkan lebih sedikit sumber daya. Berikut contoh penggunaan generator untuk keperluan yang lebih kompleks, seperti melakukan penghitungan fibonacci.

```
# generator version
```

```
def fibon(n):  
    a = b = 1  
    for i in range(n):  
        yield a  
        a, b = b, a + b
```

Sekarang kita dapat memanggilnya seperti iterasi normal:

```
for x in fibon(1000000):  
    print(x)
```

Ketika dijalankan dengan menggunakan Python 3.5 pada komputer berbasis Linux 64 bit, proses diatas hanya membutuhkan memory 37MB. Akan tetapi jika menggunakan list sebagai variabel penampung, kemudian dijalankan pada mesin yang sama, maka akan menggunakan memory sebesar 1,2GB.

```
# list version  
def fibon(n):  
    a = b = 1  
    result = []  
    for i in range(n):  
        result.append(a)  
        a, b = b, a + b  
  
    return result
```

Diawal tadi telah saya sebutkan bahwa kita dapat melakukan iterasi pada fungsi generator hanya sekali, karena generator tidak menyimpan hasilnya pada memory. Pada bagian ini akan kita buktikan apakah teori tersebut benar. Akan tetapi, sebelum kita membuktikannya, terlebih dahulu perlu kita ketahui tentang sebuah fungsi bawaan (built-in) Python, yaitu `next()`. Fungsi ini digunakan untuk mengakses elemen berikutnya dari sebuah sequence. Agar lebih paham, perhatikan contoh berikut:

```
def generator_function():
```



```

    for i in range(3):
        yield i

gen = generator_function()
print(next(gen))
# Output:
# 0
print(next(gen))
# Output:
# 1
print(next(gen))
# Output:
# 2
print(next(gen))
# Output:
# StopIteration

```

Seperti terlihat pada contoh diatas, setelah semua nilai ditampilkan, fungsi next() menghasilkan error `StopIteration`. Error tersebut menunjukkan bahwa semua nilai telah ditampilkan (yield). Mungkin kita akan bertanya-tanya, kenapa jika kita menggunakan for ... loop error tersebut tidak muncul? Hal itu dikarenakan for- loop secara otomatis menghandle error ini dan menghentikan pemanggilan fungsi next(). Apakah kalian tahu jika beberapa tipe data bawaan Python juga mendukung iterasi? Mari kita coba:

```

my_string = "Soedomoto"
next(my_string)
# Output:
# TypeError: 'str' object is not an iterator

```

Contoh diatas menghasilkan error yang menyatakan bahwa str bukan sebuah iterator. Str memang termasuk sebuah iterable, tapi bukan sebuah iterator. Maksudnya adalah, str mendukung iterasi, tetapi kita tidak dapat melakukannya secara langsung. Jadi bagaimana cara melakukan iterasi pada sebuah string? Untuk melakukannya, kita perlu satu lagi fungsi bawaan Python, yaitu

iter(). Fungsi iter() mengembalikan obyek iterator dari sebuah iterable. Sekarang mari kita buktikan!

```
my_string = "Soedomoto"
my_iter = iter(my_string)
print(next(my_iter))
# Output:
# S
print(next(my_iter))
# Output:
# o
print(next(my_iter))
# Output:
# e
print(next(my_iter))
# Output:
# d
```

Setelah kita buktikan jika str adalah sebuah iterable, sekarang pertanyaannya adalah apakah int juga sebuah iterable? Mari kita buktikan.

```
int_var = 2017
int_iter = iter(int_var)
# Output:
# TypeError: 'int' object is not iterable
```

Sekarang kita telah pahami tentang generator dan kapan dia sebaiknya digunakan. Kalian dapat mengeksplorasi lebih jauh tentang generator dengan mencobanya sendiri.

BAB XI

COROUTINE

Coroutine pada dasarnya mirip dengan generator, tetapi dengan sedikit perbedaan. Perbedaan utama dari keduanya adalah:

1. Generator adalah produsen data
2. Coroutine adalah konsumen data

Pertama sekali, mari kita review kembali pembuatan generator. Kita dapat membuat generator dengan cara seperti ini:

```
def fib():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a+b
```

Kemudian kita dapat memanggilnya dengan cara:

```
for i in fib():
    print(i)
```

Seperti yang telah kita ketahui, cara ini cepat dan tidak membutuhkan banyak memory, karena generator menggenerate nilainya on the fly dan bukan dengan menyimpannya pada sebuah list. Sekarang, jika kita menggunakan yield pada contoh diatas, maka secara umum kita telah menggunakan coroutine. Coroutine mengkonsumsi nilai-nilai yang dikirimkan kepadanya. Sebuah contoh sederhana dari coroutine adalah seperti berikut:

```
def grep(pattern):
    print("Searching for", pattern)
    while True:
        line = (yield)
        if pattern in line:
```

```
print(line)

search = grep('coroutine')
next(search)
# Output: Searching for coroutine
search.send("I love you")
search.send("Don't you love me?")
search.send("I love coroutines instead!")
# Output: I love coroutines instead!
```

Sekarang kita telah memiliki contoh coroutine. Pada awalnya sebuah coroutine tidak memiliki data. Data akan disuplai dengan menggunakan method send. Setiap data baru yang masuk akan diterima oleh yield, dan kemudian operasi dilakukan terhadap data tersebut.

Lalu, apa maksud dari next? Next diperlukan untuk menjalankan coroutine. Seperti generator, coroutine tidak secara otomatis dijalankan. Mereka baru dijalankan ketika method `__next__()` dan `send()` dipanggil. Sehingga kita harus memanggil method `next()` agar coroutine dijalankan. Kita dapat menutup coroutine dengan memanggil method `close()`.

```
search = grep('coroutine')
# ...
search.close()
```

Dan masih banyak lagi hal-hal terkait coroutine. Kita juga dapat melihat presentasi dari David Beazley terkait coroutine pada [tautan ini](http://www.dabeaz.com/coroutines/Coroutines.pdf)².

² <http://www.dabeaz.com/coroutines/Coroutines.pdf>

BAB XII

DECORATOR

Decorator merupakan suatu bagian yang sangat penting dalam Python. Decorator, dalam bahasa sederhana, dapat diartikan sebagai fungsi yang memodifikasi fungsionalitas dari fungsi yang lain. Decorator dapat membantu dalam membuat kode menjadi lebih ringkas dan lebih Pythonic. Sebagian besar beginner belum cukup memahami bagaimana menggunakannya, jadi kita akan coba uraikan pada bab ini.

Pertama, mari kita diskusikan bagaimana kita membuat decorator. Ini mungkin akan sedikit sulit dipahami, jadi kita akan uraikan satu demi satu.

20. Segala Sesuatu Dalam Python Adalah Objek

Pertama-tama, mari kita pahami sebuah fungsi pada Python.

```
def hi(name="soedomoto") :  
    return "hi " + name  
  
print(hi())  
# Output:  
# 'hi soedomoto'
```

Kita juga dapat meng-assign sebuah fungsi seperti layaknya sebuah variabel.

```
greet = hi
```

Kita tidak menggunakan parentheses disini, karena kita tidak berniat memanggil (mengeksekusi) fungsi ini. Jadi kita hanya meletakkan fungsi hi kedalam variable greet. Mari kita coba jalankan.

```
print(greet())  
# Output:  
# 'hi soedomoto'
```

Bagaimana jika kita menghapus fungsi hi, kemudian memanggilnya? Bagaimana pula jika kita memanggil variabel greet yang notabene nilainya sama dengan fungsi hi?

```
del hi  
print(hi())  
# Outputs:  
# NameError: name 'hi' is not defined  
  
print(greet())  
# Outputs:  
# 'hi soedomoto'
```

Jadi, jelas disini assignment `greet = hi` tidak hanya berupa reference, namun `greet` merupakan objek yang terpisah dari fungsi `hi`.

21. Mendefinisikan Fungsi Didalam Fungsi

Sekarang, mari kita selangkah lebih maju. Pada Python, kita dapat mendefinisikan fungsi di dalam fungsi lain, atau dapat kita sebut sebagai nested function.

```
def hi(name="soedomoto"):  
    print("now you are inside the hi() function")  
  
    def greet():  
        return "now you are in the greet()  
function"  
  
    def welcome():  
        return "now you are in the welcome()  
function"  
  
    print(greet())
```

```
print(welcome())
print("now you are back in the hi() function")

hi()
# Output:
# now you are inside the hi() function
# now you are in the greet() function
# now you are in the welcome() function
# now you are back in the hi() function
```

Contoh diatas menunjukkan bahwa ketika kita memanggil fungsi hi(), maka fungsi greet() dan welcome() juga ikut dipanggil. Akan tetapi, fungsi greet() dan welcome() ini hanya berupa fungsi yang bersifat lokal, sehingga fungsi tersebut tidak dapat diakses dari luar fungsi hi().

```
greet()
# Output:
# NameError: name 'greet' is not defined
```

Jadi sekarang kita telah ketahui bahwa kita dapat mendefinisikan fungsi didalam fungsi lain. Dengan kata lain, kita dapat membuat nested function. Sekarang kita akan pelajari satu hal lagi tentang fungsi, yaitu fungsi juga dapat mengembalikan fungsi lain.

22. Mengembalikan Fungsi Dari Dalam Fungsi Lain

Terkadang kita tidak membutuhkan untuk menjalankan sebuah fungsi didalam fungsi lain, sehingga kita dapat menjadikan fungsi tersebut sebagai return value.

```
def hi(name="soedomoto") :
    def greet() :
        return "now you are in the greet()
function"

    def welcome() :
```

```

        return "now you are in the welcome()
function"

    if name == "soedomoto":
        return greet
    else:
        return welcome

a = hi()
print(a)
# Output:
# <function hi.<locals>.greet at 0x7f0c464b2f28>

b = hi('bukan soedomoto')
print(b)
# Output:
# <function hi.<locals>.welcome at 0x7f0c464ab048>

```

Contoh diatas jelas menunjukkan bahwa variabel a merujuk kepada fungsi greet() didalam fungsi hi(), dan variabel b merujuk kepada fungsi welcome() didalam fungsi hi(). Dengan begitu kita dapat memanggil fungsi greet dan welcome dengan cara memanggil fungsi a dan b. Mari kita coba.

```

print(a())
# Output:
# now you are in the greet() function

print(b())
# Output:
# now you are in the welcome() function

```

Mari lihat kembali kode diatas. Pada klausa if-else, nilai yang di-return adalah greet dan welcome (hanya referencenya saja, tidak dijalankan), bukan greet() dan welcome(). Jika nilai yang dikembalikan greet() dan welcome(), maka fungsi tersebut akan dieksekusi, dan nilai yang dikembalikan adalah hasil eksekusi fungsi greet() dan welcome().

Jika belum cukup jelas, lihat kembali poin 6.1. Pada saat baris kode `greet = hi`, maka fungsi `hi` tidak dieksekusi. Sama dengan baris diatas, pada saat `return greet`, maka fungsi `greet` tidak dieksekusi, sehingga fungsi `hi` ketika dieksekusi akan mengembalikan nilai berupa fungsi `greet` yang belum dieksekusi.

23. Menjadikan Sebuah Fungsi Sebagai Argumen Dari Fungsi Lain

Selain dapat dijadikan sebagai `return value`, fungsi juga dapat dijadikan sebagai argumen. Pada kasus ini, yang dijadikan argumen hanya merupakan nama dari fungsi tersebut tanpa dieksekusi (tanpa parentheses). Lebih jelasnya, mari perhatikan contoh berikut:

```
def hi():
    return "hi soedomoto!"

def doSomethingBeforeHi(func):
    print("I am doing some boring work before
executing hi()")
    print(func())

doSomethingBeforeHi(hi)
# Output
# I am doing some boring work before executing hi()
# hi soedomoto!
```

Pada contoh diatas, fungsi `hi` dijadikan sebagai parameter dari fungsi `doSomethingBeforeHi`. Pada saat dijadikan parameter, fungsi `hi` tidak dieksekusi (tidak menggunakan parentheses), sehingga variable `func` akan sama dengan fungsi `hi`. Ketika fungsi `func` dieksekusi dalam body fungsi `doSomethingBeforeHi`, maka hasilnya akan sama dengan ketika fungsi `hi` dieksekusi.

Sampai dengan tahap ini, kita sudah memiliki cukup pengetahuan untuk membuat decorator. Untuk itu kita akan lanjutkan kedalam pembahasan tentang pembuatan decorator.

24. Membuat Decorator

Apa sebenarnya fungsi dari sebuah decorator? Decorator memungkinkan kita mengeksekusi kode sebelum dan setelah sebuah fungsi dieksekusi. Agar jelas apa yang dimaksud, perhatikan contoh berikut:

```
def a_new_decorator(a_func):  
    def wrap_the_function():  
        print("I am doing some boring work before  
executing a_func()")  
        a_func()  
        print("I am doing some boring work after  
executing a_func()")  
  
    return wrap_the_function  
  
def a_function_requiring_decoration():  
    print("I am the function which needs some  
decoration to remove my foul smell")
```

`a_function_requiring_decoration` merupakan sebuah fungsi biasa. Sementara `a_new_decorator` adalah sebuah fungsi yang berfungsi sebagai decorator. Decorator ini akan mengembalikan sebuah nilai yang merupakan reference kepada fungsi lain `wrap_the_function` didalam `a_new_decorator` (Lihat kembali poin 22). Untuk melihat bagaimana kode diatas dieksekusi, perhatikan kode berikut:

```
a_function_requiring_decoration()  
# Outputs:  
# I am the function which needs some decoration to  
remove my foul smell
```

```

# now a_function_requiring_decoration will be
wrapped
# by a_new_decorator()
a_function_requiring_decoration =
a_new_decorator(a_function_requiring_decoration)

a_function_requiring_decoration ()
# Outputs:
# I am doing some boring work before executing
a_func()
# I am the function which needs some decoration to
remove my foul smell
# I am doing some boring work after executing
a_func()

```

Bagaimana, mudah dipahami bukan? Kita disini hanya mengaplikasikan prinsip-prinsip yang telah kita pelajari sebelumnya pada poin 20 sampai dengan 23. Inilah yang dimaksud decorator pada Python! Mereka membungkus sebuah fungsi, dan mengubah perilaku dari fungsi tersebut. Sekarang mungkin kita bertanya-tanya, kenapa kita menggunakan decorator tanpa menggunakan @ satupun. Penggunaan @ hanyalah sebuah cara ringkas untuk membuat sebuah fungsi terdekorasi. Dibawah ini kita akan memodifikasi kode kita dengan menggunakan @ sebagai decorator.

```

@a_new_decorator
def a_function_requiring_decoration():
    print("I am the function which needs some
    decoration to remove my foul smell")

```

Jika kita eksekusi fungsi diatas, maka hasilnya akan berbeda dengan apa yang di-print pada fungsi tersebut. Karena `@a_new_decorator` telah mengubah perilaku dari fungsi tersebut.

```

a_function_requiring_decoration()
# Outputs:

```

```
# I am doing some boring work before executing
a_func()
# I am the function which needs some decoration to
remove my foul smell
# I am doing some boring work after executing
a_func()
```

Jadi `@a_new_decorator` hanyalah sebuah cara ringkas untuk menyatakan `a_function_requiring_decoration = a_new_decorator(a_function_requiring_decoration)`.

Sampai disini saya harap kita telah memiliki pengetahuan dasar tentang bagaimana decorator bekerja di Python. Sekarang kita memiliki masalah baru dengan fungsi yang telah dilengkapi decorator. Jika kita jalankan kode berikut:

```
print(a_function_requiring_decoration.__name__)
# Output:
# wrap_the_function
```

Apakah ada yang aneh? Ya, metadata nama dari fungsi berubah, seharusnya fungsi tersebut bernama

`a_function_requiring_decoration`, tetapi disitu tertulis `wrap_the_function`. Jadi, fungsi

`a_function_requiring_decoration` telah diganti dengan `wrap_the_function` pada saat fungsi tersebut dieksekusi (runtime). Untungnya, Python mempunyai fungsi sederhana untuk mengatasi masalah tersebut, yaitu `functool.wraps`. Jadi mari kita ubah sedikit kode kita dengan mengakomodir penggunaan `functools.wraps`.

```
from functools import wraps

def a_new_decorator(a_func):
    @wraps(a_func)
    def wrap_the_function():
```

```

        print("I am doing some boring work before
executing a_func()")
        a_func()
        print("I am doing some boring work after
executing a_func()")

    return wrap_the_function

@a new decorator
def a_function_requiring_decoration():
    print("I am the function which needs some
decoration to remove my foul smell")

print(a_function_requiring_decoration.__name__)
# Output:
# a_function_requiring_decoration

```

Jadi dari contoh diatas, secara umum blueprint dari sebuah decorator dapat kita tulis sebagai berikut:

```

from functools import wraps

def decorator_name(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        if not can_run:
            return "Function will not run"
        return f(*args, **kwargs)

    return decorated

@decorator_name
def func():
    return("Function is running")

can_run = True
print(func())
# Output: Function is running

can_run = False
print(func())
# Output: Function will not run

```

Catatan: @wraps mengambil sebuah fungsi untuk didekorasi dan mengubah nama fungsi, dokumentasi fungsi, argumen-argumen, dll. Sekarang mari kita bahas area-area dimana penggunaan decorator akan sangat bermanfaat dan membuatnya benar-benar mudah untuk di-manage.

a. Authorisasi

Decorator dapat membantu untuk secara otomatis mengecek apakah seseorang telah terauthorisasi untuk mengakses sebuah endpoint pada aplikasi web berbasis Python. Decorator secara luas digunakan pada web framework berbasis Python, seperti Flask dan Django. Dibawah ini adalah contoh bagaimana decorator digunakan untuk melakukan autentikasi:

```
from functools import wraps

def requires_auth(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        auth = request.authorization
        if not auth or not
check_auth(auth.username, auth.password):
            authenticate()
        return f(*args, **kwargs)

    return decorated
```

b. Logging

Logging juga merupakan area dimana decorator sering digunakan.

```
from functools import wraps

def logit(func):
    @wraps(func)
    def with_logging(*args, **kwargs):
        print(func.__name__ + " was called")
```

```

        return func(*args, **kwargs)
    return with_logging

@logit
def addition_func(x):
    return x + x

result = addition_func(4)
# Output: addition_func was called

```

Sekarang, saya yakin kalian pasti telah mempunyai gambaran bagaimana menggunakan decorator pada area-area yang lain.

25. Decorator Dengan Argumen

Mari kita lihat kembali contoh penggunaan decorator untuk logging pada poin 6.5.2. Bagaimana jika kita ingin menambahkan fungsionalitas logging ke file?

```

from functools import wraps

def logit(logfile='out.log'):
    def logging_decorator(func):
        @wraps(func)
        def wrapped_function(*args, **kwargs):
            log_string = func.__name__ + " was
called"

            print(log_string)
            # Open the logfile and append
            with open(logfile, 'a') as opened_file:
                # Now we log to the specified
logfile
                opened_file.write(log_string +
'\n')

            return wrapped_function
        return logging_decorator

@logit()
def addition_func(x):

```

```

    return x + x

@logit(logfile='out2.log')
def square_func(x):
    return x**2

result = addition_func(4)
# Output: addition_func was called

result = square_func(4)
# Output: square_func was called

```

Jika fungsi `addition_func` dijalankan maka kita akan dapati sebuah file baru bernama `out.log`. Dan jika fungsi `square_func` dijalankan, maka akan terbentuk sebuah file baru bernama `out2.log`. Bagaimana decorator diatas bekerja? Pada saat `@logit()` dieksekusi, sebenarnya kita mendekorasi fungsi `addition_func` dengan `logging_decorator`. Hal ini dikarenakan fungsi `logit()` mengembalikan nilai berupa referensi ke `logging_decorator`.

```

print(logit())
# Output:
# <function logit.<locals>.logging_decorator at
0x7f65ec418f28>

```

26. Kelas Decorator

Pada subbab sebelumnya, kita telah pelajari bagaimana membuat decorator yang mengakomodir parameter. Pada subbab ini, kita akan mengubah fungsi decorator menjadi kelas decorator.

```

class logit(object):
    def __init__(self, logfile='out.log'):
        self.logfile = logfile

    def __call__(self, func):
        log_string = func.__name__ + " was called"
        print(log_string)

```



```

        # Open the logfile and append
        with open(self.logfile, 'a') as
opened_file:
            # Now we log to the specified logfile
            opened_file.write(log_string + '\n')
            # Now, send a notification
            self.notify()

    def notify(self):
        # logit not perform notify
        pass

@logit()
def myfunc1():
    pass

```

Kelas decorator diatas mempunyai fungsionalitas yang sama dengan decorator yang berupa fungsi pada poin 25. Akan tetapi decorator yang berupa kelas mempunyai keuntungan, yaitu decorator ini mudah untuk diturunkan menjadi decorator lain. Berikut cara penggunaannya:

```

class email_logit(logit):
    def __init__(self, email='admin@myproject.com',
*args, **kwargs):
        self.email = email
        super(email_logit, self).__init__(*args,
**kwargs)

    def notify(self):
        print('Email sent to ' + self.email)

@email_logit()
def myfunc1():
    pass

```

Kelas decorator **email_logit** merupakan turunan dari decorator **logit**, sehingga memiliki fungsionalitas yang dimiliki induknya.

BAB XIII

FUNCTION CACHING

Caching function memungkinkan kita untuk menyimpan sementara nilai dari return value dari sebuah fungsi tergantung dari argumennya. Hal ini akan menghemat waktu ketika sebuah fungsi sering dipanggil dengan argumen yang sama.

Sebelum Python 3.2 kita harus menuliskan implementasi sendiri untuk melakukan caching sebuah fungsi. Sementara semenjak Python 3.2 dan yang lebih baru terdapat dekorator `lru_cache` yang memungkinkan untuk mengimplementasikan cache dan `uncache` pada sebuah fungsi secara cepat.

Sekarang mari kita lihat bagaimana mengimplementasikan caching pada sebuah fungsi pada Python 3.2+ dan sebelumnya:

27. Python 3.2+

Sekarang perhatikan contoh berikut. Pada contoh ini kita mencoba mengimplementasikan kalkulator fibonacci dan penggunaan `lru_cache`.

```
from functools import lru_cache

@lru_cache(maxsize=32)
def fib(n):
    if n < 2:
        return n
    return fib(n-1) + fib(n-2)

print([fib(n) for n in range(10)])
# Output:
# [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Argumen `maxsize` pada dekorator `lru_cache` menunjukkan berapa maksimal nilai terbaru yang akan di-cache. Pada contoh diatas

maxsize=32 berarti terdapat 32 nilai terkini yang di-cache. Kita juga dapat dengan mudah menghilangkan cache dengan fungsi `uncache`.

```
fib.cache_clear()
```

28. Python 3.1-

Pada Python 3.1 dan versi-versi sebelumnya, tidak terdapat dekorator `lru_cache`, sehingga kita perlu mengimplementasikan mekanisme caching sendiri. Akan tetapi secara generik, caching pada Python 3.1 dan sebelumnya dapat diimplementasikan dengan menggunakan dekorator yang dirancang sendiri seperti berikut:

```
from functools import wraps

def memoize(function):
    memo = {}
    @wraps(function)
    def wrapper(*args):
        if args in memo:
            return memo[args]
        else:
            rv = function(*args)
            memo[args] = rv
            return rv
    return wrapper

@memoize
def fibonacci(n):
    if n < 2: return n
    return fibonacci(n - 1) + fibonacci(n - 2)

print(fibonacci(25))
# Output:
# 75025
```

BAB XIV

CONTEXT MANAGER

Context manager memungkinkan kita untuk mengalokasikan dan membebaskan resource dengan tepat ketika dibutuhkan. Kasus dimana context manager paling banyak digunakan adalah pada statement 'with'. Sebagai contoh, perhatikan kode berikut:

```
with open('some_file', 'w') as opened_file:  
    opened_file.write('Hola!')
```

Kode diatas merupakan kode untuk membuka file, menuliskan data, dan menutupnya. Jika error muncul ketika data sedang dituliskan, maka context manager akan berusaha menutupnya. Kode diatas sama dengan kode berikut:

```
file = open('some_file', 'w')  
try:  
    file.write('Hola!')  
finally:  
    file.close()
```

Ketika dibandingkan dengan kode pertama, kita dapat lihat terdapat banyak kode yang diringkas hanya dengan menggunakan 'with'. Keuntungan lain adalah 'with' dapat memastikan file yang kita buka otomatis tertutup tanpa perlu memberi perhatian lebih terhadap blok isian. Kasus yang umum dimana context manager digunakan adalah pada saat locking dan unlocking resource serta opening dan closing file.

Sekarang mari kita lihat bagaimana kita mengimplementasikan context manager secara custom.

29. Context Manager Class

Hal yang paling mendasar dalam membuat kelas context manager adalah mendefinisikan method `__enter__` dan `__exit__`. Sekarang mari kita buat kelas yang mengimplementasikan context manager dan kita pelajari dasar-dasarnya.

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        self.file_obj.close()
```

Pada kode diatas, hanya dengan mengimplementasikan method `__enter__` dan `__exit__` kita telah mengimplementasikan context manager dan kita dapat menggunakan statement 'with' dengannya.

```
with File('demo.txt', 'w') as opened_file:
    opened_file.write('Hola!')
```

Apa yang terjadi pada kode diatas adalah:

1. Setelah kelas File diinstansiasi, statement 'with' menyimpan method `__exit__` dari kelas File.
2. Kemudian statement 'with' memanggil method `__enter__`.
3. Method `__enter__` membuka file dan menjadikan file handler sebagai return valuenya.
4. Blok pada statement 'with' yang berisi penulisan file akan dieksekusi.
5. Setelah seluruh isi blok statement 'with' selesai dieksekusi, statement 'with' akan memanggil method `__exit__` yang sebelumnya telah tersimpan.
6. Pada method `__exit__` file handler akan ditutup.

30. Menangani Exception

Disini kita tidak membahas tentang exception yang terdapat dalam method `__exit__` karena pada method `__exit__` telah terdapat argument traceback yang secara otomatis menangkap pesan error. Bagaimana jika terjadi error antara tahap 4 dan tahap 6? Mari kita lihat contoh berikut:

```
with File('demo.txt', 'w') as opened_file:
    opened_file.undefined_function('Hola!')

# Output:
# AttributeError: '_io.TextIOWrapper' object has no
attribute 'undefined_function'
```

Pada kode diatas, Python mencoba mengakses method `undefined_function` yang tidak pernah didefinisikan sebelumnya, maka terjadilah error. Pada saat terjadi error, sebenarnya Python telah secara otomatis mengirimkan traceback-nya ke method `__exit__`. Hal ini memungkinkan method `__exit__` memutuskan langkah selanjutnya yang harus dilakukan terhadap error tersebut. Pada method `__exit__` dimungkinkan untuk mengembalikan sebuah nilai. Jika return value bernilai true, maka menandakan error telah berhasil ditangani. Selain itu maka error akan dimunculkan. Pada contoh diatas, karena method `__exit__` tidak mengembalikan nilai apapun, maka error dimunculkan.

```
class File(object):
    def __init__(self, file_name, method):
        self.file_obj = open(file_name, method)
    def __enter__(self):
        return self.file_obj
    def __exit__(self, type, value, traceback):
        print("Exception has been handled")
        self.file_obj.close()
        return True
```

Dengan mengembalikan nilai true, maka sekarang error tidak akan dimunculkan lagi.

```
with File('demo.txt', 'w') as opened_file:
    opened_file.undefined_function('Hola!')

# Output:
# Exception has been handled
```

31. Context Manager Sebagai Generator

Kita juga dapat mengimplementasikan context manager sebagai dekorator dan generator. Python memiliki module contextlib untuk keperluan ini. Kita mengimplementasikan context manager bukan dengan sebuah kelas, tetapi dengan fungsi generator. Mari kita lihat contoh sederhana ini:

```
from contextlib import contextmanager

@contextmanager
def open_file(name):
    f = open(name, 'w')
    yield f
    f.close()

with open_file('some_file') as f:
    f.write('hola!')
```

Kode diatas terlihat lebih sederhana untuk mengimplementasikan context manager dibandingkan dengan menggunakan kelas. Akan tetapi, metode ini membutuhkan pengetahuan tentang generator, yield, dan decorator. Pada contoh diatas, kita juga tidak harus mengimplementasikan error handling yang mungkin terjadi. Mari sedikit kita jabarkan tentang kode diatas:

1. Python menggunakan keyword 'yield' dan bukan 'return' karena kita menggunakan generator dan bukan fungsi normal.
2. Contextmanager digunakan sebagai dekorasi, sehingga contextmanager akan dipanggil dengan nama fungsi, yaitu "open_file" sebagai argumen-nya.
3. Fungsi contextmanager mengembalikan nilai sebuah generator yang di-wrap dengan objek GeneratorContextManager.
4. Objek GeneratorContextManager di-assign pada fungsi open_file, sehingga ketika kita memanggil fungsi open_file sebenarnya kita memanggil objek GeneratorContextManager.

BAB XV

MAP, FILTER, DAN REDUCE

Pada bab ini akan dibahas 3 (tiga) fungsi yang akan memfasilitasi pendekatan fungsional pada pemrograman Python, yaitu map, filter, dan reduce. Kita akan bahas satu per satu dan memahami use case-nya. Akan tetapi, sebelum kita membahas map, filter, dan reduce, terlebih dahulu kita akan pahami tentang lambda.

32. Lambda

Lambda secara umum dapat dipahami sebagai sebuah fungsi yang hanya terdiri dari satu baris kode. Fungsi yang dikonversi menjadi lambda umumnya sebuah fungsi sederhana yang hanya memiliki sedikit body, sebagian besarnya bahkan hanya memiliki body tunggal. Secara umum lambda dapat kita tulis dalam format:

```
lambda param: result
```

Dimana sebelah kiri dari kolon (:) adalah parameter, dan sebelah kanannya adalah hasilnya. Sebagai contoh, umumnya kita membuat fungsi sederhana seperti berikut:

```
def square(x):  
    return x**2  
  
print(square(5))  
# Output:  
# 25
```

Fungsi diatas, dapat kita sederhanakan menjadi lambda menjadi:

```
square = lambda x: x**2  
print(square(5))  
# Output:
```

33. Map

Map merupakan sebuah fungsi bawaan pada Python yang berguna untuk mengimplementasikan sebuah fungsi pada setiap elemen pada input list. Secara umum, map dapat dituliskan sebagai berikut:

```
map(function_to_apply, list_of_inputs)
```

Ketika kita ingin memanggil sebuah fungsi yang mengikutkan setiap elemen dalam list, umumnya kita akan melakukan iterasi dan memanggil fungsi pada setiap iterasinya kemudian mengumpulkan hasilnya. Contohnya:

```
items = [1, 2, 3, 4, 5]
squared = []
for i in items:
    squared.append(i**2)

print(squared)
# Output:
# [1, 4, 9, 16, 25]
```

Dengan menggunakan map yang dikombinasikan dengan lambda, kita dapat melakukannya dengan lebih mudah dan ringkas.

```
items = [1, 2, 3, 4, 5]
squared = list(map(lambda x: x**2, items))

print(squared)
# Output:
# [1, 4, 9, 16, 25]
```

Selain menggunakan list data, kita juga dapat menggunakan list fungsi sebagai input.

```
def multiply(x):
    return (x*x)

def add(x):
    return (x+x)

funcs = [multiply, add]
for i in range(5):
    value = list(map(lambda x: x(i), funcs))
    print(value)

# Output:
# [0, 0]
# [1, 2]
# [4, 4]
# [9, 6]
# [16, 8]
```

34. Filter

Seperti namanya, filter membuat list elemen yang mana sebuah fungsi yang menyaringnya menghasilkan true. Pada dasarnya filter mengimplementasikan for-loop, akan tetapi lebih cepat dalam eksekusinya. Berikut contoh sederhanannya, umumnya kita melakukan penyaringan nilai sebagai berikut:

```
def is_positive(x):
    if x >= 0:
        return True
    else:
        return False

number_list = range(-5, 5)
positive_numbers = []
for i in number_list:
    if is_positive(i):
        positive_numbers.append(i)

print(positive_numbers)
# Output:
```

```
# [0, 1, 2, 3, 4]
```

Dengan fungsi filter, kita dapat melakukannya dengan lebih mudah dan ringkas.

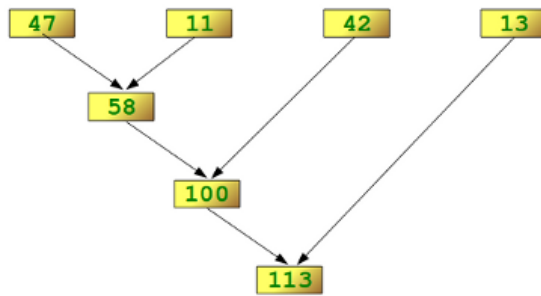
```
def is_positive(x):  
    if x >= 0:  
        return True  
    else:  
        return False  
  
number_list = range(-5, 5)  
positive_numbers = list(filter(is_positive,  
                                number_list))  
print(positive_numbers)  
# Output:  
# [0, 1, 2, 3, 4]
```

Dan bahkan dapat lebih ringkas lagi dengan mengkombinasikannya dengan lambda.

```
number_list = range(-5, 5)  
positive_numbers = list(filter(lambda x: x >= 0,  
                                number_list))  
print(positive_numbers)  
# Output:  
# [0, 1, 2, 3, 4]
```

35. Reduce

Reduce merupakan sebuah fungsi yang sangat berguna ketika melakukan komputasi pada sebuah list, yang mana komputasi dari sebuah elemen sangat bergantung pada hasil komputasi elemen sebelumnya. Misalnya, terdapat sebuah list `[47, 11, 42, 13]`, yang ingin kita lakukan operasi sum, maka langkah yang kita terapkan adalah:



Normalnya, pada Python akan diimplementasikan dengan alur seperti berikut:

```
input_list = [47,11,42,13]

sum_result = 0;
for i in input_list:
    sum_result += i

print(sum_result)
# Output:
# 113
```

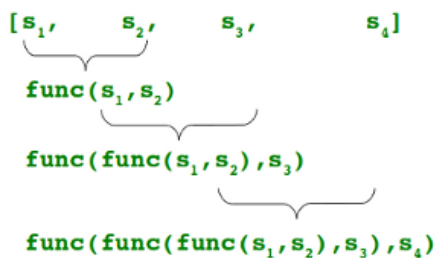
Dengan reduce, operasi ini dapat kita sederhanakan menjadi:

```
import functools

input_list = [47,11,42,13]
sum_result = functools.reduce(lambda a,b: a+b,
input_list)
print(sum_result)
# Output:
# 113
```

Pada reduce, sebuah fungsi atau lambda diimplementasikan pada sebuah list, yang mana fungsi tersebut mempunyai 2 (dua) buah parameter. Parameter pertama dari fungsi tersebut merupakan hasil

dari operasi sebelumnya. Operasi reduce dapat digambarkan sebagai berikut:



$[s_1, s_2, s_3, s_4]$

$\text{func}(s_1, s_2)$

$\text{func}(\text{func}(s_1, s_2), s_3)$

$\text{func}(\text{func}(\text{func}(s_1, s_2), s_3), s_4)$

BAB XVI

EXCEPTION

Penanganan kesalahan atau exception adalah sebuah seni dalam semua bahasa pemrograman. Tanpa penanganan kesalahan, ketika kode kita mengalami masalah, maka program kita akan terhenti. Sebaliknya, dengan penanganan yang tepat, maka kita dapat membuat keputusan apakah ingin program dihentikan atau tetap dijalankan..

Sekarang kita akan pelajari bagaimana kita seharusnya menangani kesalahan. Pada Python, terdapat mekanisme try-except. Kode yang berpotensi mengalami masalah kita letakkan pada blok 'try', dan penanganannya ketika masalah terjadi kita implementasikan pada blok 'except'.

Contoh sederhana, operasi yang melibatkan input/output selalu berpotensi munculnya masalah. Misalnya pembacaan file, akan berpotensi error jika file yang dibaca tidak tersedia. Contoh lain, pembacaan konten suatu halaman web juga berpotensi masalah jika terjadi masalah pada jaringan, misalnya tidak terkoneksi dengan internet.

Sekarang, mari kita coba kode sederhana dibawah ini:

```
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('An IOError occurred. {}'.format(e.args[-1]))

# Output:
# An IOError occurred. No such file or directory
```

Contoh diatas menunjukkan bagaimana kita menangani error terkait IO yang disebabkan karena file yang akan dibaca tidak tersedia.

Berikutnya kita akan bahas beberapa hal menarik terkait dengan exception.

36. Menangani Multiple Exception

Ada kalanya terdapat sebuah operasi yang berpotensi muncul error lebih dari satu. Terkadang error-error tersebut memiliki dampak yang setara sehingga kita cukup menggunakan sebuah penanganan error. Akan tetapi, terkadang kita perlu memisahkan antara penanganan error satu dengan error yang lain. Berikut ini kita akan contohkan bagaimana keduanya diterapkan.

```
try:
    file = open('test.txt', 'rb')
except (IOError, EOFError) as e:
    print("An error occurred. {}".format(e.args[-1]))
```

Contoh diatas menunjukkan bagaimana kita menangani dua buah error sekaligus, yaitu IOError dan EOFError, dengan satu buah penanganan. Cara yang lebih global untuk menangani kesalahan adalah dengan menggunakan Exception. Exception akan menangani segala bentuk kesalahan tanpa perlu menuliskannya satu per satu.

```
try:
    file = open('test.txt', 'rb')
except Exception as e:
    print("An error occurred. {}".format(e.args[-1]))
```

Jika kita ingin memisahkan penanganan antara satu kesalahan dengan kesalahan yang lain, kita dapat memisahkan blok except satu per satu menurut jenis kesalahannya.

```
try:
    file = open('test.txt', 'rb')
```



```
except EOFError as e:
    print("An EOF error occurred.")
    raise e
except IOError as e:
    print("An IO error occurred.")
    raise e
```

37. Klausula Finally

Pada saat kita menggunakan klausula try-except, blok 'try' hanya akan dijalankan secara lengkap jika tidak terjadi kesalahan. Sementara blok 'except' akan dijalankan ketika terjadi kesalahan. Bagaimana jika kita ingin menjalankan beberapa baris kode, baik ketika terjadi kesalahan maupun tidak? Untuk keperluan ini, kita dapat menggunakan klausula try-except-finally. Blok pada 'finally' akan dijalankan baik terjadi kesalahan maupun tidak. Biasanya blok 'finally' dimaksudkan untuk clean-up. Dibawah ini contoh penggunaannya:

```
try:
    file = open('test.txt', 'rb')
except IOError as e:
    print('An IOError occurred. {}'.format(e.args[-1]))
finally:
    print("This would be printed whether or not an exception occurred!")

# Output:
# An IOError occurred. No such file or directory
# This would be printed whether or not an exception occurred!
```

38. Klausula Try-Else

Dalam beberapa kasus, kita menginginkan beberapa baris kode dijalankan jika kode sebelumnya tidak terjadi kesalahan. Python

menawarkan klausa try-except-else untuk mengakomodirnya, meskipun klausa ini jarang digunakan oleh sebagian besar programmer karena telah diakomodir oleh klausa try-except.

```
try:
    print('I am sure no exception is going to
    occur!')
except Exception:
    print('exception')
else:
    print('This would only run if no exception
    occurs.')
finally:
    print('This would be printed in every case.')

# Output: I am sure no exception is going to occur!
# This would only run if no exception occurs.
# This would be printed in every case.
```

Kode diatas pada dasarnya sama dengan kode berikut:

```
try:
    print('I am sure no exception is going to
    occur!')
    print('This would only run if no exception
    occurs.')
except Exception:
    print('exception')
finally:
    print('This would be printed in every case.')

# Output: I am sure no exception is going to occur!
# This would only run if no exception occurs.
# This would be printed in every case.
```

BAB XVII

EXTENSI C PADA PYTHON

Salah satu fitur yang paling menarik dari Python adalah kita dapat membungkus kode bahasa C dengan Python. Mungkin kita bertanya-tanya, untuk apa kita menggunakan bahasa C pada Python? Terdapat beberapa alasan:

1. Kita ingin agar kode kita dapat dijalankan lebih cepat, terutama untuk kode yang melibatkan data dalam jumlah besar. Seperti kita tahu, kode yang ditulis dalam bahasa C lebih cepat 50x dari kode serupa yang ditulis dalam bahasa Python.
2. Beberapa library bahasa C mempunyai fungsionalitas yang kita inginkan, yang Python tidak dapat melakukannya.
3. Beberapa hal terkait pengaksesan resource yang bersifat low-level, seperti memory dan akses file.
4. Karena ingin saja, dan tidak ada alasan spesifik.

Untuk melakukannya, terdapat beberapa cara yang biasa digunakan oleh developer Python, yaitu ctypes, SWIG, dan Python/C API. Masing-masing memiliki kelebihan dan kekurangannya masing-masing.

Catatan: Untuk dapat membungkus C dengan Python, kita perlu menginstall C compiler, misalnya gcc. Untuk Windows, kita dapat menggunakan gcc pada Cygwin atau MinGW. C compiler berguna untuk mengkompilasi kode C ke dalam *.so (Linux dan Mac) atau *.dll (Windows).

39. Ctypes

Module ctypes pada Python mungkin merupakan cara paling mudah untuk memanggil fungsi C dari Python. Ctypes menyediakan tipe data yang kompatibel dengan tipe data di Python. Untuk membuktikannya, mari kita coba contoh berikut.

Pada contoh ini, kita akan membuat fungsi penjumlahan pada C. Pertama buat sebuah kode dengan ekstensi .c, misalnya add.c:

```
// filename: add.c
// sample C file to add 2 numbers - int and floats

#include <stdio.h>

int add_int(int, int);
float add_float(float, float);

int add_int(int num1, int num2){
    return num1 + num2;
}

float add_float(float num1, float num2){
    return num1 + num2;
}
```

Kemudian kita akan compile kode diatas sehingga terbentuk file .so atau .dll. Kode berikut akan menghasilkan file adder.so atau adder.dll, tergantung OS kita.

```
# For Linux or Cygwin
$ gcc -shared -Wl,-soname,adder -o adder.so -fPIC
adder.c

# For Mac
$ gcc -shared -Wl,-install_name,adder.so -o
adder.so -fPIC add.c
```

Sekarang dari kode Python kita dapat langsung mengaksesnya dengan cara seperti berikut:

```
from ctypes import *

adder = CDLL('./adder.so')

res_int = adder.add_int(4,5)
print("Sum of 4 and 5 = " + str(res_int))
# Output:
# Sum of 4 and 5 = 9

a = c_float(5.5)
b = c_float(4.1)
add_float = adder.add_float
add_float.restype = c_float
print("Sum of 5.5 and 4.1 = ", str(add_float(a,
b)))
# Output:
# Sum of 5.5 and 4.1 = 9.600000381469727
```

Pada contoh diatas, file *.c mengandung dua buah fungsi, fungsi pertama untuk menjumlahkan dua buah integer, dan fungsi kedua untuk menjumlahkan dua buah float. Kemudian pada file Python, langkah pertama adalah mengimport semua hal (*) pada modul ctypes. Setelah itu fungsi CDLL digunakan untuk men-load file *.so atau *.dll yang telah kita buat. Fungsi-fungsi yang telah didefinisikan pada C sekarang dapat diakses melalui Python. Ketika kita memanggil `adder.add_int(...)` pada Python, sebenarnya kita memanggil fungsi `add_int(...)` pada C.

Ctypes memungkinkan kita menggunakan tipe data native Python secara default untuk integer dan string. Sementara untuk tipe data yang lain misalnya boolean dan float, kita perlu menggunakan ctypes yang bersesuaian. Misalnya ketika kita memanggil fungsi `adder.add_float(...)`, terlebih dahulu kita menciptakan variable bertipe `c_float` yang akan digunakan sebagai parameter.

Ctypes mudah digunakan, tetapi memiliki keterbatasan. Misalnya, ctypes tidak memungkinkan untuk memanipulasi object pada sisi C.

40. SWIG

SWIG adalah kependekan dari Simplified Wrapper and Interface Generator. SWIG merupakan cara lain untuk mengakses kode C dari Python. SWIG relatif lebih sulit digunakan dibandingkan dengan ctypes. Akan tetapi SWIG menawarkan fungsionalitas yang lebih baik. Untuk menggunakan SWIG, kita terlebih dahulu harus menginstal SWIG. Cara instalasinya dapat dilihat pada [dokumentasi](#) dari SWIG.

Pada SWIG, kita juga perlu membuat sebuah file ekstra yang merupakan input dari SWIG dengan menggunakan command line utility. Untungnya, developer Python umumnya tidak membutuhkannya.

Agar kita paham bagaimana menggunakannya, mari kita mulai dengan terlebih dahulu membuat kode dalam bahasa C dan menyimpannya pada sebuah file, misalnya example.c:

```
// Filename: example.c
#include <time.h>
double My_variable = 3.0;

int fact(int n) {
    if (n <= 1) return 1;
    else return n*fact(n-1);
}

int my_mod(int x, int y) {
    return (x%y);
}

char *get_time() {
    time_t ltime;
    time(&ltime);
```

```
    return ctime(&lttime);  
}
```

Kemudian kita membuat file yang berfungsi sebagai interface. File ini mempunyai nama yang sama dengan file C, tetapi dengan ekstensi *.i:

```
/* Filename: example.i */  
%module example  
  
%{  
/* Put header files here or function declarations  
like below */  
extern double My_variable;  
extern int fact(int n);  
extern int my_mod(int x, int y);  
extern char *get_time();  
%}  
  
extern double My_variable;  
extern int fact(int n);  
extern int my_mod(int x, int y);  
extern char *get_time();
```

Kemudian kita compile dengan perintah berikut (asumsi OS Linux, Python 3.5):

```
$ swig -python -py3 example.i  
$ gcc -Wall -I/usr/include/python3.5 -lpython3.5 -  
fPIC -c example.c example_wrap.c  
$ ld -shared example.o example_wrap.o -o  
_example.so
```

Kemudian kita dapat mengaksesnya dari Python:

```
import example  
  
print(example.fact(5))  
# Output:  
# 120
```

```
print(example.my_mod(7,3))
# Output:
# 1

print(example.get_time())
# Output:
# Thu Dec 14 23:17:51 2017
```

Seperti terlihat diatas, SWIG memberikan hasil yang sama, tetapi membutuhkan effort yang lebih besar. Akan tetapi, effort yang dikeluarkan akan terbayar jika kita menargetkan beberapa bahasa pemrograman yang berbeda.

41. Python/C API

C/Python API bisa jadi merupakan metode yang paling banyak dipakai, bukan karena kemudahannya, tapi karena kemampuannya dalam memanipulasi objek Python dari dalam kode C. Metode ini mengharuskan kode C kita ditulis spesifik agar dapat berinteraksi dengan kode Python.

Seluruh objek dalam Python adalah representasi dari struct PyObject, dan Python.h header menyediakan beragam fungsi untuk memanipulasinya struct tersebut. Sebagai contoh, jika PyObject adalah PyListType (list dalam Python), maka kita dapat menggunakan fungsi PyList_Size() pada struct untuk mendapatkan panjang dari list. Fungsi ini sama dengna ketika kita memanggil len(list) pada Python. Sebagian besar dari fungsi dasar pada Python tersedia C headernya pada Python.h.

Contoh:

Pada contoh ini, kita akan membuat ekstensi Python dengan kode C yang berfungsi untuk menjumlahkan seluruh elemen pada list (semua elemen adalah number). Mari kita mulai dengan interface akhir yang diimplementasikan pada kode Python:


```
import addList

l = [1,2,3,4,5]
print("Sum of List - " + str(l) + " = " +
      str(addList.add(l)))
```

Kode diatas terlihat seperti kode Python biasa, yang melakukan import modul `addList` dan memanggil fungsi `add()`. Perbedaannya adalah bahwa modul `addList` tidak ditulis dalam kode Python sama sekali, tetapi ditulis dalam kode C. Berikutnya, kita akan lihat kode C yang dikemas kedalam modul Python `addList`. Ini terlihat sedikit membingungkan pada awalnya, tetapi sekali kita mampu memahaminya kita akan mendapati bahwa cara ini sangat mudah.

```
// Filename: adder.c

//Python.h has all the required function
definitions to manipulate the Python objects
#include <Python.h>

//This is the function that is called from your
python code
static PyObject* addList_add(PyObject* self,
PyObject* args){
    PyObject * listObj;

    //The input arguments come as a tuple, we parse
the args to get the various variables
    //In this case it's only one list variable,
which will now be referenced by listObj
    if (! PyArg_ParseTuple( args, "O", &listObj))
        return NULL;

    //length of the list
    long length = PyList_Size(listObj);
    //iterate over all the elements
    int i, sum =0;

    for(i = 0; i < length; i++){
```

```

        //get an element out of the list - the
        element is also a python objects
        PyObject* temp = PyList_GetItem(listObj,
i);
        //we know that object represents an integer
        - so convert it into C long
        long elem = PyInt_AsLong(temp);
        sum += elem;
    }

    //value returned back to python code - another
    python object
    //build value here converts the C long to a
    python integer
    return Py_BuildValue("i", sum);
}

//This is the docstring that corresponds to our
'add' function.
static char addList_docs[] = "add( ): add all
elements of the list\n";

/* This table contains the relevant info mapping -
<function-name in python module>, <actual-
function>,
<type-of-args the function expects>, <docstring
associated with the function>
*/
static PyMethodDef addList_funcs[] = {
    {"add", (PyCFunction)addList_add, METH_VARARGS,
addList_docs},
    {NULL, NULL, 0, NULL}
};

/*
addList is the module name, and this is the
initialization block of the module.
<desired module name>, <the-info-table>, <module's-
docstring>
*/
PyMODINIT_FUNC inittaddList(void){
    Py_InitModule3("addList", addList_funcs, "Add
all ze lists");
}

```

```
}
```

Berikut penjelasan dari kode diatas:

- File <Python.h> memuat seluruh representasi dalam kode C dari seluruh tipe data yang digunakan pada Python, serta seluruh definisi fungsi-fungsi yang digunakan untuk melakukan operasi terhadap objek tersebut.

```
#include <Python.h>
```

- Setelah itu kita buat fungsi dalam kode C yang akan dipanggil dari kode Python. Konvensi yang digunakan adalah {nama-modul}_{nama-fungsi}, yang pada contoh ini adalah `addList_add`.

```
static PyObject* addList_add(PyObject* self,  
    PyObject* args) {  
    ...  
}
```

Fungsi `addList_add` menerima argumen bertipe struct `PyObject`. Argumen yang akan digunakan pada kode Python bertipe tuple, tapi karena semua hal di Python adalah object, maka kita gunakan tipe generik `PyObject`.

Argumen tersebut kemudian diparse (split argumen menjadi beberapa elemen) dengan menggunakan fungsi

`PyArg_ParseTuple(...)`. Parameter pertama merupakan variabel yang akan diparse. Parameter kedua adalah sebuah string yang mendefinisikan bagaimana cara memparse setiap elemen pada tuple `args`. Sementara parameter-parameter berikutnya adalah variabel yang akan menampung setiap elemen hasil parse. Jumlah argument harus sama dengan jumlah argumen yang diharapkan module function, dan posisi dari setiap argumen harus sama dengan posisi yang

diharapkan. Sebagai contoh, jika kita mengharapkan sebuah string, integer, dan Python list secara berurutan, maka dapat kita tuliskan sebagai berikut:

```
int n;  
char *s;  
PyObject* list;  
PyArg_ParseTuple(args, "siO", &n, &s, &list);
```

Kemudian kita gunakan `PyList_Size(...)` untuk mendapatkan ukuran dari list. Fungsi ini sama dengan fungsi `len(list)` pada Python. Setelah itu, kita lakukan looping pada list, dan dapatkan nilai dari setiap elemen dengan fungsi `PyList_GetItem(..., ...)`. Parameter pertama merupakan list object, dan parameter kedua adalah indeks dari elemen. Fungsi `PyList_GetItem` akan mengembalikan nilai bertipe `PyObject*`, akan tetapi karena kita telah ketahui bahwa tipe data spesifiknya adalah int, maka kita dapat melakukan casting dengan menggunakan fungsi `PyInt_AsLong(...)`.

Setelah kita mendapatkan semua elemen, maka kita lakukan operasi penjumlahan. Hasil penjumlahan harus terlebih dahulu dikonversi agar sesuai dengan tipe data pada Python dengan menggunakan fungsi `Py_BuildValue`. Fungsi ini mempunyai 2 (dua) argumen. Argumen pertama adalah string yang merepresentasikan tipe data yang akan dihasilkan, dan argumen kedua adalah nilai yang akan dikonversikan. "i" pada para argumen pertama menunjukkan tipe data dari hasil konversi adalah integer.

- Kemudian, buat sebuah tabel info, yang memuat semua informasi yang relevan terkait fungsi-fungsi yang akan termuat didalam modul. Setiap baris berkorespondensi

terhadap sebuah fungsi, dengan baris terakhirnya adalah nilai sentinen (baris dari elemen-elemen null).

```
static PyMethodDef addList_funcs[] = {
    {"add", (PyCFunction)addList_add,
    METH_VARARGS, addList_docs},
    {NULL, NULL, 0, NULL}
};
```

- Terakhir adalah blok inisialisasi modul, yaitu signature PyMODINIT_FUNC init{module-name}.

```
PyMODINIT_FUNC inittestaddList(void) {
    Py_InitModule3("addList", addList_funcs,
    "Add all ze lists");
}
```

Setelah kode C selesai kita buat dan kita simpan dengan nama adder.c, selanjutnya kita membuat file setup.py yang akan digunakan untuk mengkompilasi dan menginstal modul yang kita buat diatas agar dapat digunakan pada kode Python.

```
from distutils.core import setup, Extension

setup(name='addList', version='1.0',
    ext_modules=[Extension('addList', ['adder.c'])])
```

Kemudian jalankan kode diatas melalui terminal atau command prompt dengan perintah:

```
$ python setup.py install
```

Setelah proses selesai, kita dapat mengujinya apakah modul diatas bekerja dengan baik:

```
import addList
```

```
l = [1,2,3,4,5]
print("Sum of List - " + str(l) + " = " +
      str(addList.add(l)))
# Output:
# Sum of List - [1, 2, 3, 4, 5] = 15
```

Seperti terlihat, kita telah berhasil membangun sebuah ekstensi C/Python dengan menggunakan API Python.h. Metode ini kelihatannya kompleks, akan tetapi ketika telah menguasainya, metode ini akan sangat bermanfaat.

Sebenarnya, terdapat cara lain untuk membuat ekstensi Python dalam kode C, yaitu dengan menggunakan Cython. Cython lebih cepat untuk membangun ekstensi dalam kode C, tetapi karena Cython sedikit berbeda jika dibandingkan dengan Python, maka cara ini tidak akan dibahas.

BAB XVIII

VIRTUAL ENVIRONMENT

Apakah kalian pernah mendengar virtualenv? Jika kalian adalah seorang programmer pemula, mungkin belum pernah mendengarnya. Akan tetapi, jika kalian adalah programmer yang telah akrab dengan Python, virtualenv bisa jadi merupakan bagian yang sangat penting.

Jadi, apakah itu virtualenv? Virtualenv adalah sebuah tool atau library Python yang memungkinkan kita membuat environment Python yang terisolasi. Terisolasi disini maksudnya adalah Python yang terdapat pada virtualenv tidak terpengaruh oleh Python yang terinstal pada sistem anda, termasuk didalamnya library. Misalnya pada sistem telah terinstal library python numpy, maka pada virtualenv library tersebut tidak tersedia.

Lantas, kenapa kita harus menggunakan virtualenv? Coba bayangkan pada sistem anda terinstal sebuah library python versi 3.x, dan digunakan oleh sebuah aplikasi. Kemudian kita membuat aplikasi lain yang memerlukan library python versi 2.x dan dengan library yang terinstal (3.x), aplikasi baru ini tidak dapat dijalankan. Pada kondisi ini kita dihadapkan pada sebuah dilema. Jika kita pertahankan library Python versi 3.x, aplikasi baru tidak dapat dijalankan. Sementara jika kita install library versi 2.x, aplikasi lama yang tidak dapat dijalankan. Bagaimana agar kedua aplikasi tetap dapat dijalankan? Pada kondisi seperti inilah virtualenv diperlukan.

42. Instalasi

Instalasi virtualenv relatif mudah dan tidak membutuhkan library lain. Tinggal buka terminal atau command prompt (tergantung apa OS yang digunakan), dan ketikkan:

```
$ pip install virtualenv
```

Sederhana bukan. Sekarang pada komputer kita telah terinstall virtualenv.

43. Membuat Virtual Environment

Sebelum membuat virtual environment, kita akan coba eksplorasi terlebih dahulu apa opsi yang tersedia ketika akan membuat virtual environment.

```
$ virtualenv -help
Usage: virtualenv [OPTIONS] DEST_DIR

Options:
  --version                show program's version
                           number and exit
  -h, --help              show this help message and
                           exit
  -v, --verbose            Increase verbosity.
  -q, --quiet              Decrease verbosity.
  -p PYTHON_EXE, --python=PYTHON_EXE
                           The Python interpreter to
                           use, e.g.,
                           --python=python2.5 will use
                           the python2.5 interpreter
                           to create the new
                           environment. The default is the
                           interpreter that virtualenv
                           was installed with
                           (/usr/bin/python3)
  ....
```

Berdasarkan halaman bantuan diatas, cara sederhana untuk membuat virtual environment adalah:

```
$ virtualenv /path/to/dir
```


Perintah diatas akan secara otomatis membuat folder yang berisi environment Python yang terisolasi. Untuk membuktikan apakah benar Python pada virtualenvironment terisolasi, kita akan membandingkan library yang terinstal pada Python global dan Python virtual environment.

```
$ /path/to/global/python -m pip freeze
```

```
soedomoto@SoedomotoPC:~$ python -m pip freeze
appdirs==1.4.0
astroid==1.5.3
asyncoro==4.5.1
autopep8==1.3.2
backports-abc==0.5
backports.functools-lru-cache==1.4
backports.shutil-get-terminal-size==1.0.0
backports.ssl-match-hostname==3.4.0.2
beautifulsoup4==4.5.3
```

```
$ /path/to/venv/python -m pip freeze
```

```
soedomoto@SoedomotoPC:~/Documents/venv/bin$ ./python -m pip freeze
soedomoto@SoedomotoPC:~/Documents/venv/bin$
```

Terlihat pada kedua gambar diatas, library yang terinstal pada Python global dan Python virtual environment berbeda. Terbukti bahwa virtualenv dapat membuat Python yang terisolasi.

Catatan: Jika pada komputer kita terinstall lebih dari satu versi Python, misalnya Python 2.7 dan Python 3.6, maka kita dapat menggunakan option `-p` untuk menentukan binary Python yang akan digunakan. Cara menggunakannya:

```
$ virtualenv -p /path/to/python/exe /path/to/dir
```



```
$ ./deactivate
```

Kita telah mempelajari beberapa hal terkait virtualenv dan dasar-dasar cara menggunakannya. Lebih detailnya dapat dipelajari pada dokumentasi yang tersedia pada [tautan ini](#).

BAB XIX

TARGET PYTHON 2 DAN 3

Pada sebagian besar kasus, kita ingin membangun sebuah program yang dapat dijalankan pada Python 2.x dan Python 3.x sekaligus. Bayangkan jika kita punya modul yang akan digunakan oleh ratusan orang, sebagian menggunakan Python 2.x dan sebagian lain menggunakan Python 3.x saja. Pada kondisi seperti ini kita punya 2 (dua) opsi: pertama, kita mendistribusikan modul dalam 2 (dua) versi, yaitu Python 2.x dan Python 3.x, dan kedua yaitu membuat kode kita kompatibel baik untuk Python 2.x maupun Python 3.x. Pada bagian ini, kita akan sedikit mempelajari bagaimana kita dapat membuat script yang kompatibel baik untuk Python 2.x maupun Python 3.x.

45. Import future

Penggunaan `__future__` memungkinkan untuk mengimport fungsionalitas dari Python 3.x dari Python 2.x. Sebagai contoh, context manager adalah fitur yang baru di Python 2.6+. Untuk menggunakannya pada Python 2.5, kita dapat menggunakan import seperti berikut:

```
from __future__ import with_statement
```

Fungsi print telah mengalami perubahan dari Python 2.x ke Python 3.x. Jika kita ingin menggunakan fungsi print yang kompatibel dengan Python 3.x dari Python 2.x, kita dapat meng-import-nya dari `__future__`.

```
print(print)
# Output:
# SyntaxError: invalid syntax
```

Kode diatas menghasilkan error pada Python 2.x, karena print tidak dikenali sebagai sebuah object. Untuk menjadikan print sebagai sebuah object, kita harus menambahkan fungsionalitas print yang dimiliki Python 3.x, yaitu dengan menggunakan `__future__`.

```
from __future__ import print_function
print(print)
# Output:
# <built-in function print>
```

46. Module Alias

Bagaimana kita biasanya meng-import package pada script? Sebagian besar dari kita akan melakukannya dengan cara ini:

```
import foo
```

atau

```
from foo import bar
```

Tahukan kalian, jika kita juga bisa melakukan penamaan atau alias pada modul dan fungsi? Cara menggunakannya sebagai berikut:

```
import foo as bar
```

Beberapa modul dan fungsi pada Python 2.x dan Python 3.x memiliki fungsionalitas sama, namun memiliki nama yang berbeda. Perhatikan contoh berikut:

```
try:
    # Import urllib in Python 3
    import urllib.request as urllib_request
except ImportError:
    # Import urllib in Python 2
    import urllib2 as urllib_request
```

Pada kode diatas, kita meletakkan import pada try-except. Hal ini dikarenakan tidak ada modul urllib.request pada Python 2.x, sehingga proses import ini akan menghasilkan ImportError. Pada Python 2.x fungsionalitas urllib.import disediakan oleh modul urllib2. Sehingga ketika kita eksekusi dengan Python 2.x dan import urllib.request dieksekusi, kita akan mendapatkan ImportError, sehingga blok except yang terdapat kode import urllib2 akan dieksekusi.

Satu lagi yang menarik dari contoh diatas adalah keyword 'as'. Keyword 'as' memberi nama alias kepada sebuah modul atau fungsi. Dengan keyword 'as', urllib.request dan urllib2, kedua-duanya dapat diberi dengan nama alias yang sama, yaitu urllib_request.

47. Obsolete Python 2 Builtin

Terdapat 12 Python 2.x builtin yang tidak terdapat pada Python 3.x. Agar kode kita compatible dengan Python 3.x, maka pastikan kita tidak menggunakannya. Caranya, yaitu dengan memblokir ke-12 builtin tersebut pada Python 2.x:

```
from future.builtins.disabled import *
```

Sebelum kita dapat menggunakannya, kita terlebih dahulu perlu menginstall modul future pada Python 2.x dengan menggunakan pip.

```
$ pip install future
```

Sekarang, ketika kita menggunakan modul builtin yang telah diblokir akan muncul NameError.

```
from future.builtins.disabled import *  
  
apply()
```

```
# NameError: obsolete Python 2 builtin apply is disabled
```