BAB I

\*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.

* 1. 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 menperjelas apa itu \*args. Jadi sekarang mari kita bahas apa itu \*\*kwargs.

* 1. 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.

* 1. Menggunakan \*args dan \*\*kwargs pada sebuah 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**)**

**(**'arg1:'**,** 'dua'**)**

**(**'arg2:'**,** 5**)**

**(**'arg3:'**,** 8**)**

# dengan \*\*kwargs

>>> kwargs **=** **{**"arg3"**:** 3**,** "arg2"**:** "two"**,** "arg1"**:** 5**}**

>>> test\_args\_kwargs**(\*\***kwargs**)**

**(**'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**)**

BAB II

GENERATOR

Sebelum kita membicarakan tentang generator, mari kita mulai dengan iterator. Berdasarkan wikipedia, 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

2.1. 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?

2.2. 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?

2.3. Iteration

Dalam bahasa sederhana, iteration 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.

2.4. 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 elemen-nya 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 interator

**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**)**

Hasilnya:

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**)**

Hasilnya:

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: Traceback (most recent call last):

# File "args\_kwargs.py", line 12, in <module>

# print(next(gen))

# 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: Traceback (most recent call last):

# File "args\_kwargs.py", line 2, in <module>

# next(my\_string)

# 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: Traceback (most recent call last):

# File "args\_kwargs.py", line 2, in <module>

# int\_iter = iter(int\_var)

# 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 III

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.

3.1. 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: 25

3.2. Map

Map merupakan sebuah fungsi 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]

3.3. 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. Berikut contoh sederhanannya, umumnya kita melakukan penyaringan nilai sebagi 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 dapa 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**)**

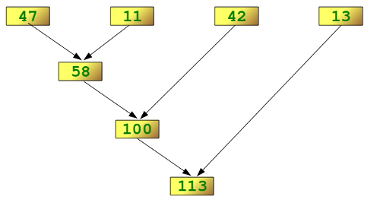
positive\_numbers **=** list**(**filter**(lambda** x**:** x **>=** 0**,** number\_list**))**

**print(**positive\_numbers**)**

# Output: [0, 1, 2, 3, 4]

3.4. 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

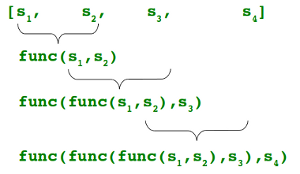
input\_list **=** **[**47**,**11**,**42**,**13**]**

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:



BAB IV

STRUKTUR DATA set

Set merupakan sebuah struktur data yang sangat penting. Set pada dasarnya sama seperti list dengan elemen unique, sehingga tidak bisa memiliki nilai duplikat. Set dapat didefinikan 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:

4.1. 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.

4.2. 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 V

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 sama dengan ekpresi if … else normal.

condition\_is\_true **if** condition **else** condition\_is\_false

**if** condition**:**

condition\_is\_true

**else:**

condition\_is\_false

Contoh penggunaan operator ternary:

is\_fat **=** **True**

state **=** "fat" **if** is\_fat **else** "not fat"

Operator ternary memungkinkan kita menguji suatu kondisi secara cepat daripada harus menggunakan statement if secara multiline. Seringkali operator ternary sangat bermanfaat dan dapat membuat kode kita lebih kompak namun tetap dapat di maintain 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 membingungkan dan 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 melalui 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 VI

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.

6.1. 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.

6.2. 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.

6.3. 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.

6.4. 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 pembahan tentang pembuatan decorator.

6.5. 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 6.3). 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 6.1. sampai dengan 6.4. Inilah yang dimaksud decorator pada Python! Mereka membungkus sebuah fungsi, dan mengubah perilaku dari fungsi tersebut. Sekarang mungkin kita heran bahwa 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.

6.5.1. 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 Djanggo. 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

6.5.2. 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.

6.6. 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 additional\_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>

6.7. 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 6.6. 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**):**

# Send an email to self.email

# Will not be implemented here

**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 VII

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 taidak 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.

7.1. 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 VIII

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 berubah, dan immutable berarti 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 kalian merasa 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 IX

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.

9.1. 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

Catatan: Abaikan simbol $. Simbol $ hanya sebagai tanda bahwa perintah tersebut dijalankan di terminal atau command prompt.

Sederhana bukan. Sekarang pada komputer kita telah terinstall virtualenv.

9.2. 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)

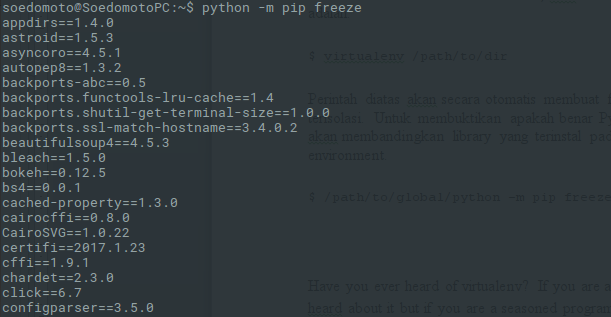
....

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

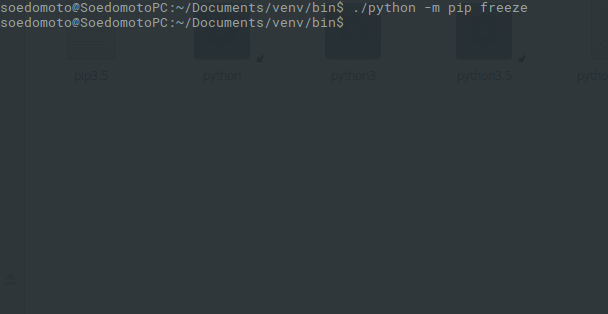
$ virtualenv /path/to/dir

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



$ /path/to/venv/python –m pip freeze



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

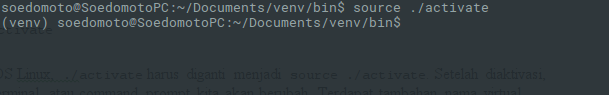
9.3. Menggunakan Virtual Environment

Setelah virtual environment yang berisi Python terisolasi telah berhasil dibentuk, virtual environment perlu diaktivasi agar Python yang digunakan dapat merujuk kepada Python yang terdapat pada virtual environment dan bukan Python yag terinstall pada sistem. Untuk melakukannya kita dapat menggunakan perintah ini:

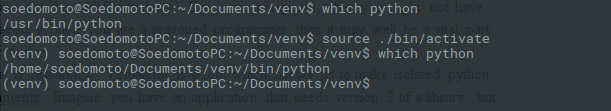
$ cd /path/to/venv

$ ./activate

Pada OS Linux, ./activate harus diganti menjadi source ./activate. Setelah diaktivasi, pada terminal atau command prompt kita akan berubah. Terdapat tambahan nama virtual environment yang kita buat di awal setiah baris perintah.



Setelah virtual environment berhasil diaktivasi, maka setiap perintah yang menggunakan interpreter Python akan merujuk pada Python pada virtual environment.



Untuk kembali ke Python yang terinstal pada sistem, maka virtual environment perlu di deaktivasi. Untuk melakukannya, kita cukup menutup terminal atau command prompt, atau dapat menggunakan command:

$ ./deactivate

Kita telah mempelajari beberapa hal terkait virtualenv dan dasar-dasar cara menggunakannya. Lebih detailnya dapat dipelajari pada dokumentasi yang tersedia pada [tautan ini](http://docs.python-guide.org/en/latest/dev/virtualenvs/).

BAB X

COLLECTIONS

Python telah dilengkapi dengan sebuah modul yang mengandung sejumlah tipe data yang disebut Collections. Tipe-tipe tersebut antara lain:

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

10.1. defaultdict

Pada dasarnya, defaultdict mempunyai karakteristik yang sama dengan dict. Bedanya, dengan 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 menkonversikannya dengan library json:

**import** json

**print(**json**.**dumps**(**some\_dict**))**

# Output:

# {"colours": {"favourite": "yellow"}}

10.2. OrderedDict

Tipe data dict yang urutannya tersusun secara acak. Urutan pada saat kita melakukan insert dan pada saat melakukan retrieve terkadang bisa berbeda, 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 dihapus.

**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

Terlihat bahwa urutan dari entri terjaga dengan menggunakan OrderedDict.

10.3. 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, 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**)**

10.4. deque

Deque pada dasarnya mirip dengan list. Perbedaannya, pada list kita hanya dapat menambahkan dan menghapus isian dari arah kanan, sementara pada deque kita dapat melakukannya pada dua arah, kanan dan kiri. 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

Kita dapat menambahkan data 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'])

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'])

Dan 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 item dalam sebuah deque. Jika kita membatasi jumlah, maka ketika jumlah item telah melebihi batas, item yang berada pada arah yang berlawanan dari item 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)

10.5. 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 tuple yang memungkinkan untuk mengakses setiap elemen-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 elemennya dengan menggunakan nama, sangat mirip dengan kita mengakses class variable. Jika kita perhatikan, sebuah named tuple membutuhkan 2 (dua) argumen pada saat pendefinisiannya. 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 nametuple 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')])

10.6. 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 10.5. 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. 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.

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

Species**(**1**)**

Species**[**'cat'**]**

Species**.**cat

This is much less error-prone. We have to be specific, and we should use only the

enumeration to name types.

There are three ways to access enumeration members. For example, all three methods

will get you the value for cat: