7. Membuat Function

Pengenalan Function Tanpa “return”

Pada saat membuat program terkadang kita menyalin blok kode yang sama di baris berikutnya, misal untuk mencetak sebuah biodata kita salin kembali kemudian ganti nilai – nilainya untuk menampilkan biodata tersebut. Apakah harus menyalin blok kode di setiap bagian kode yang membutuhkan kode tersebut ???. Waktu dulu kuliah Algoritma dan Pemrograman I, ketika mempelajari pemrograman pertama kalinya, baris kode yang sama dipakai berulang – ulang untuk mengeluarkan hasil tertentu. Tapi jadi tidak efisien, karena ukuran file program yang ditulis cukup besar. Oleh karena itu hampir di setiap bahasa pemrograman terdapat fitur yang dinamakan *function*. Nama lainnya *method, sub routine*, atau fungsi. *Function* ini berguna untuk penggunaan kembali blok kode yang akan digunakan di baris kode lain. Sekali tulis tinggal panggil.

Function ini jelas beda dengan *built-in function* yang ada di Python. *Built-in function* sendiri adalah *function* yang telah dibuatkan oleh pengembang bahasa pemrograman Python. Sedangkan *function* dibuat oleh programmer yang menggunakan bahasa pemrograman Python, atau istilah lainnya *user-defined function*.

Di Python untuk membuat *function* digunakan *keyword* **def**. Kemudian diikuti nama *function* yang diinginkan lalu parameter yang dibutuhkan dengan diawali tanda “(“ dan “)”.Untuk membuka *function* dimulai dengan tanda “:”. Tidak seperti di C, Java, atau PHP yang diawali dengan tanda “{“ dan diakhiri “}” untuk membuka sebuah *function*. Lalu tutupnya ? Seperti dijelaskan diawal di Python sendiri digunakan indentasi untuk menentukan apakah baris kode tersebut milik sebuah *function*, **if**, atau pengulangan. Jadi jika Anda ingin menuliskan kode untuk *function* yang Anda buat. Harus ada jarak satu indentasi agar kode tersebut dianggap sebagai kode dari *function* yang Anda tulis. Kemudian Anda dapat menambahkan *keyword* **return** jika *function* yang Anda tulis ingin mengembalikan nilai keluaran.

Berikut ada contoh tentang pembuatan *function* yang memiliki parameter dan tidak memiliki parameter. Penggunaan **return** digunakan pada contoh berikutnya :

listing : fungsi_1.py

```
def fungsi_tanpa_parameter():  for i in range(1, 5):
        print "looping ke - ", i

def fungsi_berparameter(batas_akhir):  for i in range(1, batas_akhir):
        print "looping ke - ", i
```

```
print " contoh penggunaan fungsi tanpa parameter : "  print "hasil : ", fungsi_tanpa_parameter()

print "\n\n"

print " contoh penggunaan fungsi berparameter : "  print "hasil : ", fungsi_berparameter(10)
```

Sebuah *function* jika dipanggil langsung nilai keluarannya tidak akan dicetak. Tetapi jika dipanggil melalui sebuah *function* seperti **print** nilai keluarannya akan ditampilkan. Kenapa nilainya "None" ? Karena di *function* yang tadi ditulis tidak disertakan *keyword* **return**. Jika sebuah *function* tidak diberikan **return** maka dapat dibilang *function* tersebut dianggap *procedure*. Sebuah *function* yang tidak memiliki nilai keluaran.

<< gambar 7.1 hasil eksekusi fungsi_1.py >>

Function yang Menggunakan "return"

Bagaimana kalau ditambahkan **return** ? Jika ditambahkan **return,** *function* yang Anda tulis akan menghasilkan nilai keluaran. Jika dipanggil langsung maka program tidak akan menampilkan nilai keluaran dari *function* tersebut. Jika *function* tersebut dipanggil melalui *function* atau *keyword* misalnya **print,** maka nilai keluarannya akan ditampilkan. Berikut terdapat *function* yang menghasilkan nilai keluaran yang memiliki parameter dan tidak berparameter :

listing : fungsi_2.py

```
def fungsi_tanpa_parameter():  temp = 0
    for i in range(1, 5):  temp = temp + i
    return temp


def fungsi_berparameter(batas_akhir):  temp = 0
    for i in range(1, batas_akhir):  temp = temp + i
    return temp


print " contoh penggunaan fungsi tanpa parameter : "  print "hasil : ", fungsi_tanpa_parameter()
print "hasil : ", fungsi_tanpa_parameter()  print "hasil : ", fungsi_tanpa_parameter()

print "\n\n"

print " contoh penggunaan fungsi berparameter : "  print "hasil : ", fungsi_berparameter(15)
print "hasil : ", fungsi_berparameter(20)  print "hasil : ", fungsi_berparameter(25)
```

Anda sendiri dapat melihat perbedannya antara *function* yang berparameter dengan tidak berparameter. Pada *function* yang tidak berparameter. Ketika dipanggil berulang – ulang nilai keluarannya tetap sama. Berbeda dengan *function* yang memiliki parameter, nilai keluaranya berbeda – beda ketika dipanggil. Tergantung nilai masukan yang diberikan.

<< gambar 7.2 hasil eksekusi fungsi_2.py >>

Default Argument pada Python

Sekarang Anda sudah mengenal *function* yang berparameter dan tidak berparameter. Umumnya saat akan memberikan nilai pada sebuah *function*, nilai tersebut akan diberikan saat *function* tersebut dipanggil. Apakah saat memasukkan nilai boleh tidak diisi atau dilewat ? Lalu apakah akan ada nilainya ?. Di Python terdapat sebuah fitur yang dinamakan *default argument* saat menulis sebuah *function*. *Default argument* sendiri adalah sebuah argumen yang sudah diisi nilai terlebih dahulu jika argumen tersebut tidak diberikan saat memanggil *function*. Jadi sekalipun dilewat nilai dari argument tersebut akan dipenuhi dengan nilai *default* nya. Berikut dibawah ini terdapat contoh pemanggilan *function* yang melewatkan semua argumen yang dibutuhkan *function*, dan yang tidak melewatkan semua argumen yang akan ditangani oleh *default argument* :

listing : fungsi_3.py

```
def cetak_biodata( nama, kota, umur=18):  print "Nama : ", nama;
   print "Umur : ", umur;  print "Kota : ", kota;  return;

# kalau parameter diisi semua
print "Tanpa memakai default argument : "
cetak_biodata( nama="miki", umur=50, kota="bandung" )

print "\n"

# kalau parameter tidak diisi semua  print "Memakai default argument : "
cetak_biodata(kota="bandung", nama="miki")
```

Kode diatas jika dieksekusi akan tampil seperti berikut :

<< gambar 7.3 hasil eksekusi fungsi_3.py >>

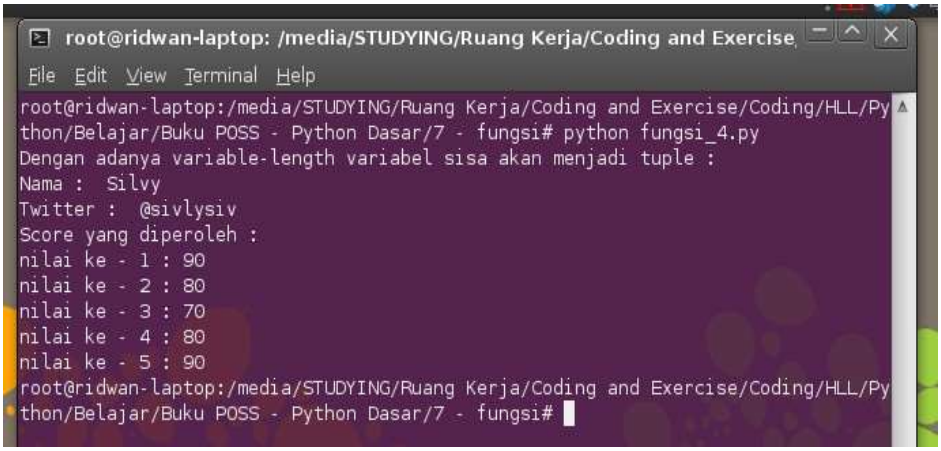
Variable-length Argument pada Python

Sekarang kita akan mengenal fitur yang dinamakan dengan *variable-length argument*. Fitur ini digunakan ketika ingin membuat sebuah *function* yang memiliki argumen yang dinamis. Argumen ini dapat disebut sebagai argumen yang tidak utama. Misal dalam sebuah fungsi dibutuhkan tiga argumen, maka argumen ke – 4 sampai ke – n argumen, tidak akan ditampung oleh argumen utama. Tapi ditampung oleh argumen terakhir yang menampung seluruh argumen yang diberikan setelah argumen utama. Di Python untuk menandakan bahwa argumen tersebut *variable-length argument*, diberikan tanda "*" pada argumen terakhir. *Variable-length argument* ini harus disimpan di akhir setelah argumen biasa dan *default argument*. Apabila disimpan di urutan awal, maka Python akan mengeluarkan *error* : **"SyntaxError: invalid syntax"**. Sebagai contoh Anda dapat perhatikan kode berikut ini :

listing : fungsi_4.py

```
def cetak_perolehan_nilai( nama, twitter, *scores):
  print "Nama : ", nama;  print "Twitter : ", twitter;
  print "Score yang diperoleh : "  i = 1
  for score in scores:
     print "nilai ke - %d : %d" % (i, score)  i= i + 1

  return;

# kalau parameter diisi semua
print "Dengan adanya variable-length variabel sisa akan menjadi tuple : "
cetak_perolehan_nilai("Silvy", "@sivlysiv", 90, 80, 70, 80, 90)
```

Seperti yang Anda lihat pada contoh diatas, argumen utama adalah nama dan twitter. Apabila kita memasukkan argumen setelahnya, maka argumen tersebut akan dikumpulkan dalam satu wadah yaitu *scores. Berapapun kita masukkan argumen, akan ditampung menjadi sebuah **list** yang berisi argumen – argumen yang dimasukkan setelah nama dan twitter.



<< gambar 7.4 hasil eksekusi fungsi_4.py >>

Keyword Argument pada Function

Dalam penggunaan *function* Anda mesti melewatkan argumen sesuai urutan yang ditulis pada parameter yang dibutuhkan oleh *function* yang Anda tulis. Apakah mungkin jika ditulis tanpa urutan argumen sudah baku pada *function* tersebut. Dalam *function* terdapat sebuah fitur yang dinamakan *keyword argument*. *Keyword argument* ini dapat melewatkan argumen tanpa harus sesuai urutan. *Keyword argument* diberikan saat memanggil sebuah *function* dengan mengambil nama argumen yang terdapat di *function* disambung dengan tanda "=" dan nilai dari argumen tersebut. Jika kita memberikan argumen yang tidak sesuai urutan tanpa menggunakan *keyword argument*, maka argumen yang diterima *function* tersebut tidak akan sesuai.

listing : fungsi_5.py

```
def cetak_biodata( nama, umur, kota):
   print "Nama : ", nama;  print "Umur : ", umur;  print "Kota : ", kota;  return;

# kalau pakai keyword argument : mau urutannya gimanapun input akan sesuai  print "Tanpa
memakai keyword argument : "
```

```
cetak_biodata( kota="bandung", nama="miki", umur=50 )

print "\n"

# kalau tidak memakai keyword argument : mau urutannya gimanapun input tidak akan sesuai  print
"Memakai keyword argument : "
cetak_biodata( "bandung", "miki", 50)

print "\n"

# kalau tidak memakai keyword argument : tapi urutannya sesuai maka input akan sesuai  print
"Memakai keyword argument : tapi urutannya sesuai "
cetak_biodata( "miki", 50, "bandung")
```

Pada contoh diatas, Anda dapat melihat perbedaan antara *function* yang melewatkan *keyword argument* dengan yang tidak menggunakan *keyword argument*. Contoh yang tidak menggunakan *keyword argument* tidak akan menerima masukan sesuai yang dibutuhkan *function* ketika urutan argumennya diacak.

<< gambar 7.5 hasil eksekusi fungsi_5.py >>

Keyword-length Argument pada Function

Keyword-length argument mempunyai cara penggunaan yang sama hanya saja, *keyword-length* ini menampung *keyword argument* yang berlebih ketika diberikan kepada *function* yang dipanggil. *Keyword argument* yang berlebih akan diterima dalam bentuk **dictionary**.

listing: fungsi_6.py

```
def cetak_perolehan_nilai( nama, twitter, **data_tambahan):  print "Nama : ", nama;
   print "Twitter : ", twitter;  print "\n"
   print "Data Lainnya : "  i = 1
   for data in data_tambahan:
      print "%s : %s" % (data, data_tambahan[data])


   return;

# kalau parameter diisi semua
print "Dengan adanya keyword argument, argumen tersisa akan menjadi dictionary : "
cetak_perolehan_nilai("Silvy","@sivlysiv",email="silvysilvy@gmail.com",
facebook="www.facebook.com/silvysil", telp="0838-1234-5678")
```

Pada contoh diatas, *keyword argument* yang berlebih ditampung kedalam *argument* data_tambahan dan argumen berlebih tersebut disimpan dalam **dictionary**.

<< gambar 7.6 hasil eksekusi fungsi_6.py >>

Pass by Reference dan Pass by Value pada Python

Berikutnya terdapat masalah *pass by value* atau *pass by reference*. Di Python semua nilai akan dilewatkan secara *by reference*. Artinya jika kita mengubah argumen di dalam fungsi maka nilai argumen yang direferensi tersebut akan ikut berubah juga. Misalkan dibawah contoh berikut terdapat sebuah **list** yang akan diganti dengan nilai baru, dan ada juga yang ditambahkan nilai baru.

listing : fungsi_7.py

```
def sebuah_fungsi(sebuah_list):  sebuah_list = [1, 2, 3, 4, 5]  print sebuah_list

def sebuah_fungsi_lainnya(sebuah_list):  sebuah_list.append([10, 20, 30])
    print sebuah_list

ini_list = [10, 20, 30]
sebuah_list = [100, 200, 300]

print "apakah ini_list berubah ? "  print ini_list  sebuah_fungsi(ini_list)
print ini_list  print ini_list
sebuah_fungsi_lainnya(ini_list)  print ini_list

print "apakah sebuah_list berubah ? "  print sebuah_list  sebuah_fungsi(sebuah_list)
print sebuah_list  print sebuah_list
sebuah_fungsi_lainnya(sebuah_list)  print sebuah_list
```

Pada kode diatas, Anda akan melihat sebuah perbedaan yang cukup penting. Ketika sebuah **list** diganti nilainya maka **list** yang ada di luar *function* tidak akan terpengaruh. Tapi ketika kita menambahkan data baru dengan menggunakan *method* pada **list** tersebut. Nilai diluar ikut berubah,. Hal ini terjadi karena pada *function* sebuah_fungsi_lainnya(), **list** sebuah_list masih menunjuk atau merujuk ke sebuah_list yang berada diluar. Atau dalam kata lain masih menunjuk ke “address” yang sama di memori utama.

<< gambar 7.7 hasil eksekusi fungsi_7.py >>

Variable Scope pada Python

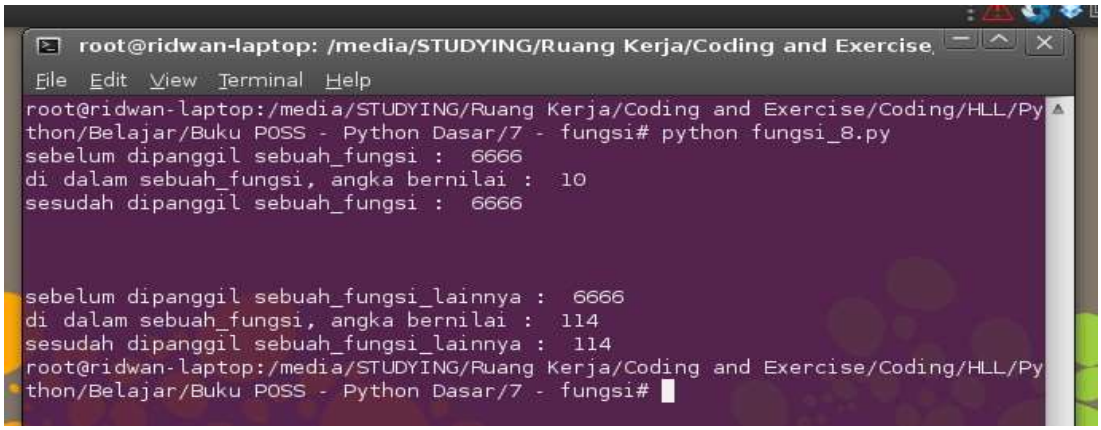
Variable scope adalah sebuah kondisi dimana variabel diakses secara lokal pada blok kode tertentu atau bersifat universal yang menyebabkan variabel tersebut dapat diakses dari blok kode manapun. Misal ada sebuah variabel di dalam *function*. Variabel tersebut bersifat lokal dan hanya dapat diakses didalam *function* tersebut. Lalu bagaimanakah kita menjadikan sebuah variabel agar bersifat global ?. Di Python terdapat sebuah *keyword* yang bernama **global**. *Keyword* ini digunakan untuk merujuk sebuah variabel di luar blok kode, misalnya sebuah variabel di dalam *function*, dengan nama yang sama.

listing : fungsi_8.py

```
def sebuah_fungsi():  angka =  10
    print "di dalam sebuah_fungsi, angka bernilai : ", angka

def sebuah_fungsi_lainnya():  global angka
    angka = 114
    print "di dalam sebuah_fungsi, angka bernilai : ", angka


angka = 6666

print "sebelum dipanggil sebuah_fungsi : ", angka  sebuah_fungsi()
print "sesudah dipanggil sebuah_fungsi : ", angka

print "\n\n"
```

```
print "sebelum dipanggil sebuah_fungsi_lainnya : ", angka  sebuah_fungsi_lainnya()
print "sesudah dipanggil sebuah_fungsi_lainnya : ", angka
```

Pada kode diatas variabel yang bernama angka dibubuhi *keyword* **global** pada *function* sebuah_fungsi_lainnya(). Hasilnya saat angka diubah nilainya. Maka nilai di variabel angka yang berada di luar blok *function* sebuah_fungsi_lainnya() ikut berubah.



<< gambar hasil eksekusi fungsi_8.py >>