Chapter 0.1:
Dasar Bahasa Python

February 20, 2021

1 Dasar-Dasar Python

Buku ini ditulis dengan menggunakan **Jupyter Notebook**. Pada **Chapter 0** buku ini, akan dijelaskan perintah-perintah dasar Python, dengan menggunakan *interface* `IPython`. Meskipun menggunakan `IPtyhon`, tapi sebagian besar perintah dapat dieksekusi juga dari *command prompt* "Python".

Pada python, komentar menggunakan tanda #

[1]:
```
# ini komentar
x = 12  # bukan komentar
print(x)  # menampilkan x
```

```
12
```

1.1 Menggunakan Python seperti Calculator

`Python` layaknya `Matlab`, dapat dieksekusi secara interaktif.

1.1.1 Bilangan

[2]:
```
1 + 4
```

[2]: 5

[3]:
```
(12-3*2)
```

[3]: 6

[4]:
```
(12-3*2)/7
```

[4]: 0.8571428571428571

[5]:
```
7/8
```

[5]: 0.875

[6]:
```
8/4  # hasil pembagian selalu floating point, meskipun seharusnya hasilnya␣
↪integer (bulat)
```

[6]:
```
2.0
```

[7]:
```
21//4 # floor division, mengabaikan angka di belakang koma
```

[7]:
```
5
```

[8]:
```
21 % 4 # sisa pembagian
```

[8]:
```
1
```

[9]:
```
5 * 4 + 1 # hasil*pembagi + sisa pembagian
```

[9]:
```
21
```

Menggunakan ** sebagai pangkat

[10]:
```
7 ** 2 # 7 kuadrat
```

[10]:
```
49
```

[11]:
```
5 ** 3 # 5 pangkat 3
```

[11]:
```
125
```

Tanda sama dengan = digunakan untuk memberi nilai kepada sebuah variabel

[12]:
```
x = 10 # Nilai tidak akan ditampilkan
```

Ekspresi yang muncul di luaran dapat disimpan pada variabel _

[13]:
```
Buku = 100000/2
```

[14]:
```
harga = 50*5
```

[15]:
```
Buku * harga
```

[15]:
```
12500000.0
```

[16]:
```
harga + _
```

[16]:
```
12500250.0
```

Variabel _ akan berisi hasil perkalian buku dengan harga yaitu sehingga `harga + _ = 12500750.0`

Selain `float` dan `int`, Python juga mendukung jenis bilangan lain seperti `Decimal` dan `Fraction`. Python juga mempunyai *built-in function* untuk bilangan kompleks, menggunakan `j` atau `J`.

[17]:
```
Z1 = 2 + 5J
```

[18]:
```
Z2 = 3 + 4.0j
```

[19]:
```
type(Z1)
```

[19]:
```
complex
```

[20]:
```
type(Z2)
```

[20]:
```
complex
```

1.1.2 String

Selain bilangan, Python juga dapat digunakan untuk memanipulasi tipe `string`. Bisa digunakan *single quote* `'...'` atau *double quote* `"..."`, dengan hasil yang sama. \ dapat digunakan untuk *escape character* dari quote.

[21]:
```
'Learning Python'
```

[21]:
```
'Learning Python'
```

[22]:
```
"Learning Python"
```

[22]:
```
'Learning Python'
```

[23]:
```
'doesn\'t'
```

[23]:
```
"doesn't"
```

Menggunakan \ untuk escape *single quote*, atau dapat dilakukan juga dengan `double quote` tanpa *escape character* \.

[24]:
```
"doesn't"
```

[24]:
```
"doesn't"
```

Fungsi `print()` menghasilkan output yang lebih dapat terbaca dengan menghilangkan *enclosing quotes* dan mencetak *escaped and special character*:

[25]:
```
s = 'First line.\nSecond line.' # \n berarti garis baru
```

[26]:
```
s
```

[26]:
```
'First line.\nSecond line.'
```

[3]:
```
print(s)
```

```
First line.
Second line.
```

Jika sebuah karakter yang diberikan prefix \ tidak ingin diinterpretasikan sebagai karakter spesial maka bisa digunakan *raw string* dengan menggunakan awalan `r` pada quote pertama.

[4]:
```
print('C:\some\name')
```

```
C:\some
ame
```

[5]:
```
print(r'C:\some\name')
```

```
C:\some\name
```

String bisa span banyak baris dengan menggunakan """..."""" atau '''...'''. Untuk menghindari *end of line* di-insert secara otomatis maka gunakan \ diakhir baris.

[2]:
```
print("""\
Usage: thingy [OPTIONS]
-h Display this usage message
-H hostname Hostname to connect to """)
```

```
Usage: thingy [OPTIONS]
-h Display this usage message
-H hostname Hostname to connect to
```

Perhatikan tidak ada baris di-insert di baris pertama.

String bisa dirangkai dengan + dan diulang dengan *.

[29]:
```
'Fiky' + ' Yosef' + ' Suratman'
```

[29]:
```
'Fiky Yosef Suratman'
```

[30]:
```
3*'he' + '....'
```

[30]:
```
'hehehe...'
```

Satu atau lebih *string literals* yang ditempatkan berdekatan satu sama lain otomatis akan di-rangkaikan. Hal ini berguna untuk memotong string yang panjang.

[31]:
```
'Fi' 'Ky'
```

[31]:
```
'FiKy'
```

[32]:
```
text = ('Put several strings within parentheses '
        'to have them joined together.')
```

[33]:
```
text
```

[33]:
```
'Put several strings within parentheses to have them joined together.'
```

[34]:
```
prefix = 'Fiky'
```

[35]:
```
prefix + ' Yosef'
```

[35]:
```
'Fiky Yosef'
```

String bisa diakses per huruf dengan indeks, huruf pertama pada index 0.

[36]:
```
prefix[0]
```

[36]:
```
'F'
```

[37]:
```
prefix[3]
```

[37]:
```
'y'
```

Bisa diakses dengan index negatif, dimulai dari belakang

[38]:
```
prefix[-1]
```

[38]:
```
'y'
```

[39]:
```
prefix[-4]
```

[39]:
```
'F'
```

Selain *indexing*, pada Python disupport juga *slicing*, dimana awal selalu dimasukan dan akhir tidak.

[40]:
```
Nama = 'Fiky Yosef Suratman'
```

[41]:
```
Nama[0:6] # Luaran tidak akan memasukan Nama[6], tapi Nama[0] akan masuk
```

[41]:
```
'Fiky Y'
```

Indeks dari *Slicing* mempunyai default yang sangat berguna. Indeks pertama yang dihilangkan default-nya menjadi 0. Sedangkan indeks kedua yang dihilangkan default-nya menjadi ukuran string yang sedang di *slicing*.

[42]:
```
Nama[:6] # Sama dengan hasil sebelumnya dengan menggunakan Nama[0:6]
```

[42]:
```
'Fiky Y'
```

[43]:
```
Nama[6:19]
```

[43]:
```
'osef Suratman'
```

[44]:
```
Nama[6:len(Nama)]
```

[44]:
```
'osef Suratman'
```

[45]:
```
Nama[6:]
```

[45]:
```
'osef Suratman'
```

Tiga perintah terakhir menghasilkan luaran yang sama.

[46]:
```
Nama[7:]
```

[46]:
```
'sef Suratman'
```

[47]:
```
Nama[7:100] # Tidak akan menghasilkan error, meski 100 > len(Nama)
```

[47]:
```
'sef Suratman'
```

[48]:
```
Nama[20:100] # Tidak akan menghasilkan error, meski slicing out of range
```

[48]:
```
''
```

String termasuk tipe data yang ***Immutable***, tidak akan berubah dengan assignment.

[49]:
```
Nama[0]='J'
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-49-6a4c6c4435a5> in <module>
----> 1 Nama[0]='J'

TypeError: 'str' object does not support item assignment
```

[50]:
```
Nama[17:19]='in'
```

```
␣
↪---------------------------------------------------------------------------

TypeError                                 Traceback (most recent call␣
↪last)

<ipython-input-50-9ed217e9dd1b> in <module>
----> 1 Nama[17:19]='in'

TypeError: 'str' object does not support item assignment
```

Tetapi dapat digunakan sandingan string:

[51]:
```
Nama1 = 'J' + Nama[1:]
```

[52]:
```
Nama1
```

[52]:
```
'Jiky Yosef Suratman'
```

1.1.3 List

Python mempunyai sejumlah tipe data `compound`, digunakan untuk *grouping*. Salah satunya adalah list, yang dapat dituliskan harga-harga yang dipisahkan dengan koma diantara dua

square brackets.

[53]:
```
listVar = [1, 4, 9, 16, 25]
```

[54]:
```
listVar
```

[54]:
```
[1, 4, 9, 16, 25]
```

Elemen-elemen di dalam *list* boleh tidak sejenis:

[4]:
```
Kombinasi =[1, 2, 3, 'hei','huy']
```

[7]:
```
Kombinasi[3]
```

[7]:
```
'hei'
```

[9]:
```
Kombinasi[4][0:2]
```

[9]:
```
'hu'
```

Seperti strings (dan semua tipe berurut *built-in*), lists juga dapat diindeks dan *slicing*:

[165]:
```
listVar[0] # indexing mengembalikan sebuah item
```

[165]:
```
1
```

[166]:
```
listVar[:3] # slicing mengembalikan list yang baru
```

[166]:
```
[1, 4, 9]
```

[167]:
```
listVar[3:]
```

[167]:
```
[16, 25]
```

[45]:
```
listVar[-2:]
```

[45]:
```
[16, 25]
```

[46]:
```
listVar[:]
```

[46]:
```
[1, 4, 9, 16, 25]
```

List mendukung *concatenation*

[47]:
```
listVar + [36, 49]
```

[47]:
```
[1, 4, 9, 16, 25, 36, 49]
```

Tidak seperti string, list adalah *mutable*:

[48]:
```
listVar3 = [1, 8, 27, 65, 5**3]
```

```
[49]: listVar3

[49]: [1, 8, 27, 65, 125]

[53]: listVar3[3]=64

[56]: listVar3

[56]: [1, 8, 27, 64, 125]

[58]: listVar3.append(6**3)

[59]: listVar3

[59]: [1, 8, 27, 64, 125, 216]
```

Pemberian nilai menggunakan *slicing* memungkinkan juga, ukuran list bisa berubah atau menghilangkannya isinya sama sekali:

```
[1]: huruf =['a','b','c','d','e']

[2]: huruf

[2]: ['a', 'b', 'c', 'd', 'e']

[3]: huruf[2:4]=['C','D'] # Mengganti

[4]: huruf

[4]: ['a', 'b', 'C', 'D', 'e']

[5]: huruf[2:4]=[] # Menghilangkan

[6]: huruf

[6]: ['a', 'b', 'e']

[10]: huruf[:]=[] # Menghapus semua isi

[11]: huruf

[11]: []
```

Memungkinkan juga untuk membuat *nested list*, contoh:

```
[61]: x =['a','b','c','d']; y =[1,2,3]; c =[x,y]

[62]: c

[62]: [['a', 'b', 'c', 'd'], [1, 2, 3]]
```

[63]:
```
c[0]
```

[63]:
```
['a', 'b', 'c', 'd']
```

[64]:
```
c[0][3]
```

[64]:
```
'd'
```

[66]:
```
c[1][0]
```

[66]:
```
1
```

2 Flow Control

2.1 While

Loop dengan menggunakan *Flow control* `while` akan dieksekusi selama kondisi `True` pernyataan yang menyertainya. Setiap harga *non-zero Integer* adalah `True`, sedangkan nol adalah `False`. Operator pembanding diantaranya `< (less than)`, `> (greater than)`, `== (equal to)`, `<= (less than or equal to)`, `>= (greater than or equal to)` and `!= (not equal to)`

[2]:
```
a,b = 0,1
while b<10:
    print(b, end=',')
    a, b = b, a+b
```

```
1,1,2,3,5,8,
```

Argumen `end` digunakan untuk menghindari output *newline* pada perintah `print()` atau menggunakan akhiran lain (dalam contoh menggunakan ,).

2.2 If

Pernyataan yang paling terkenal untuk *flow control* adalah menggunakan `if`. Mungkin tidak sama sekali atau terdapat bagian `elif`, dan bagian `else` adalah opsional

[57]:
```
x = int(input("Please enter an integer: "))
```

```
Please enter an integer:  1
```

[58]:
```
if x < 0:
        x=0
        print('Negative changed to zero')
elif x==0:
        print('Zero')
elif x==1:
        print('Single')
else:
        print('More')
```

```
Single
```

2.3 For

Iterasi dengan `For` pada Python berbeda dengan bahasa pemrograman lain. Pernyataan `For` pada Python mengiterasi berdasarkan item-item yang ada pada *list* atau *string*.

[11]:
```
Kata =['Kucing','Anjing','Ayam']
for k in Kata:
    print(k,len(k))
```

```
Kucing 6
Anjing 6
Ayam 4
```

2.4 Fungsi `range()`

Jika diinginkan untuk melakukan iterasi pada urutan angka, maka bisa digunakan *built-in function* `range()`, yang menghasilkan *arithmetic progressions*:

[12]:
```
for i in range(5):
    print(i)
```

```
0
1
2
3
4
```

Perhatikan bahwa *end-point* 5 tidak masuk ke dalam hitungan. Sangat mungkin range dimulai pada angka lain atau *step* yang berbeda:

```
range(5,10)
   5 sampai 9

range(0,10,3)
   0, 3, 6, 9

range(-10, -100, -30)
   -10, -40, -70
```

Untuk melakukan iterasi terhadap indeks dari *sequence*, bisa dikombinasikan `range()` dan `len()`:

[13]:
```
a = ['fiky','punya','mobil','karimun','wagon']
```

[22]:
```
for i in range(len(a)):
    print(i, a[i])
```

```
0 fiky
1 punya
2 mobil
3 karimun
4 wagon
```

[23]:
```
print(range(10))
```

```
range(0, 10)
```

Terlihat pada output diatas, tidak diprint luaran `range(10)`, karena fungsi tersebut seolah berlaku sebagai list, tapi sebetulnya bukan list. Objek tersebut disebut *iterable*, sangat cocok untuk sebuah target untuk fungsi atau konstruksi yang diharapkan darinya item-item secara berurutan sampai habis. Lihat contoh dengan `for` *statement* di atas. Fungsi `list()` akan menghasilkan lists dari objek *iterable* seperti `range()`.

[28]:
```
list(range(5))
```

[28]:
```
[0, 1, 2, 3, 4]
```

2.5 Pernyataan `break` dan `continue`, dan `else` pada loops

Pernyataan `break` adalah keluar dari loop `for` atau `while` yang terdalam. Sedangkan pernyataan `else` dengan pernyataan-pernyataan setelahnya akan dieksekusi bila loop `for` sudah *exhaustive* atau kondisi menjadi *false* pada loop `while`, tetapi tidak dieksekusi jika loop selesai dengan pernyataan `break`.

[29]:
```
for n in range(2,10):
    for x in range(2,n):
        if n % x == 0:
            print(n,'equals',x, '*', n//x)
            break
    else:
        # loop for seluruhnya habis tanpa menghasilkan faktor
        print(n,'is a prime number')
```

```
2 is a prime number
3 is a prime number
4 equals 2 * 2
5 is a prime number
6 equals 2 * 3
7 is a prime number
8 equals 2 * 4
9 equals 3 * 3
```

Perhatikan pada kode di atas, bahwa `else` milik dari pernyataan `for` bukan `if`.

Pernyataan `continue` memerintahkan untuk melanjutkan ke iterasi selanjutnya.

[32]:
```
for bilangan in range(2,10):
    if bilangan % 2 == 0:
        print('Ditemukan bilangan genap', bilangan)
        continue
    print('Ditemukan sebuah bilangan', bilangan)
```

```
Ditemukan bilangan genap 2
Ditemukan sebuah bilangan 3
Ditemukan bilangan genap 4
Ditemukan sebuah bilangan 5
Ditemukan bilangan genap 6
```

```
Ditemukan sebuah bilangan 7
Ditemukan bilangan genap 8
Ditemukan sebuah bilangan 9
```

2.6 Pernyataan `pass`

Pernyataan `pass` digunakan saat tidak diinginkan untuk mengerjakan apapun (menunggu), meskipun program memerlukannya.

```
>>> while true:
...     pass  # Sibuk-menunggu untuk keyboard interrupt ditekan (CTRL+c)
...
```

Pernyataan `pass` juga dapat digunakan sebagai *place-holder* untuk sebuah fungsi atau *conditional body* ketika kita sedang membuat kode baru, sehingga kita akan ingat untuk diimplementasikan selanjutnya. Pada kode berikut `pass` akan diabaikan.

```
>>> def initlog(*args)
...     pass # Ingat untuk implementasi fungsi ini nanti!
...
```

2.6.1 Mendefinisikan Fungsi

Untuk mendefinisikan fungsi digunakan *keyword* `def`, yang diikuti dengan nama fungsi dan parameter-paremeter yang akan digunakan (di dalam kurung). Pernyataan lain mengikuti dan didahului dengan *indent*.

[48]:
```
def fib(n):     # Menulis deret Fibonacci sampai dengan n
    """Print a Fibonacci series up to n."""    # ini menyatakan docstring
    a,b = 0,1
    while a < n:
        print(a,end=' ')
        a, b = b, a+b
    print()
```

[49]:
```
fib(2000) # panggil fungsi fib()
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

Docstring akan muncul jika dipanggil fungsi dengan tanda `?`.

```
>>> fib?
Signature: fib(n)
Docstring: Print a Fibonacci series up to n.
File:      ~/Documents/E-book/Python/PythonDataScienceHandbook-master/notebooks/<ipython-inpu
Type:      function
```

Pada sebuah fungsi *global variable* tidak bisa di-*assign* value (kecuali dinamai dengan pernyataan global), meskipun bisa diakses (direfer).

Definisi fungsi menghasilkan nama fungsi pada tabel simbol. Tipe data dari nama fungsi akan dikenali oleh interpreter sebagai *user-defined function*. Data ini bisa diberikan ke nama lain yang bisa digunakan sebagai fungsi juga. Hal ini berlaku umum sebagai mekanisme penamaan:

[50]:
```
fib
```

[50]:
```
<function __main__.fib(n)>
```

[58]:
```
f = fib # nama fungsi bisa diassign ke sebuah variable, kemudian bisa
↪dieksekusi seperti di bawah.
```

[59]:
```
f(1000)
```

```
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
```

Meskipun fungsi tidak mempunyai pernyataan `return`, tetapi fungsi menghasilkan sebuah harga berupa *built-in name* `None`.

[60]:
```
fib(0)
```

[61]:
```
print(fib(0))
```

```
None
```

Program Fibonacci yang mengembalikan nilai berupa list:

[63]:
```
def fib2(n): # return Fibonacci series up to n
    """Return a list containing the Fibonacci series up to n."""
    result = []
    a,b=0,1
    while a<n:
     result.append(a)
     a, b = b, a+b
    return result
```

[64]:
```
x = fib2(100)
```

[66]:
```
x
```

[66]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

[67]:
```
fib2(2000)
```

[67]:
```
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597]
```

Pernyataan `result.append(a)` memanggil sebuah *method* dari sebuah objek *list*. *Method* adalah sebuah fungsi yang dimiliki oleh sebuah objek, dengan penamaan `obj.methodname`, dimana `obj` adalah sebuah objek, dan `methodname` adalah nama dari metode. Cotoh di atas sama dengan pernyataan `result + [a]` tetapi eksekusinya lebih efisien.

Membuat *default* harga argumen:

[77]:
```
def ask_ok(prompt, retries=4, reminder='Please try again!'):
    while True:
        ok = input(prompt)
        if ok in ('y', 'ye', 'yes'):
```

```
            return True
        if ok in ('n', 'no', 'nop', 'nope'):
            return False
        retries = retries - 1
        if retries < 0:
            raise ValueError('invalid user response')

        print(reminder)
```

Contoh di atas juga memakai *keyword* `in`, untuk testing apakah input pada variabel `ok` mengandung harga tertentu.

[79]:
```
ask_ok('coba? ')
```

```
coba?  dk
Please try again!
coba?  kdfj
Please try again!
coba?  y
```

[79]: True

Memanggil fungsi bisa dilakukan beberapa cara:

[76]:
```
ask_ok('Do you want to quit') # memberikan argumen yang mandatory
```

```
Do you want to quit y
```

[76]: True

[80]:
```
ask_ok('Ok to overwrite the file', 2) # memberikan argumen opsional
```

```
Ok to overwrite the file n
```

[80]: False

[82]:
```
ask_ok('Ok to overwrite the file', 2, 'Come on, only yes or no!') #␣
 ↪memeberikan semua argumen
```

```
Ok to overwrite the file y
```

[82]: True

Harga default dievaluasi pada saat definisi fungsi di *defining scope*, sehingga:

[138]:
```
i = 5

def f(arg=i):
    print(arg)
```

[139]:
```
i = 6
f()
```

```
5
```

Assignment terakhir tidak akan merubah harga `i`.

Important Harga default hanya dievaluasi satu kali. Sehingga hasilnya akan berbeda ketika default adalah *mutable object* seperti *list*, *dictionary* atau *instances* kebanyakan *class*. Contoh:

[110]:
```
def f(a, L=[]):
    L. append(a)
    return L
```

[111]:
```
K = [10]
```

[112]:
```
print(f(12,K))
```

```
[10, 12]
```

[113]:
```
print(f(1))
```

```
[1]
```

[114]:
```
print(f(2))
```

```
[1, 2]
```

Jika harga default tidak ingin dishare antara satu panggilan dengan panggilan yang lain, maka fungsi bisa ditulis sebagai berikut:

[116]:
```
def f(a, L=None): # perhatikan None (N besar)
    if L is None:
        L = []
    L.append(a)
    return L
```

[117]:
```
print(f(1))
```

```
[1]
```

[118]:
```
print(f(2))
```

```
[2]
```

Fungsi juga dapat dipanggil dengan menggunakan argumen *keyword* dengan bentuk `Key_Arg = value`. Sebagai contoh:

[119]:
```
def parrot(voltage, state='a stiff', action='voom', type='Norwegian Blue'):
    print("-- This parrot wouldn't", action, end=' ')
    print("if you put", voltage, "volts through it.")
    print("-- Lovely plumage, the", type)
    print("-- It's", state, "!")
```

Fungsi di atas memerlukan satu argumen wajib (`voltage`) dan tiga argumen opsional (`state`, `action`, dan `type`).

[121]:
```
parrot(1000) # one positional argument
```

```
-- This parrot wouldn't voom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
```

[122]:
```
parrot(voltage=1000) # one keyword argument
```

```
-- This parrot wouldn't voom if you put 1000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
```

[123]:
```
parrot(voltage=1000000, action='VOOOOOM') # 2 keywords argument
```

```
-- This parrot wouldn't VOOOOOM if you put 1000000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
```

[125]:
```
parrot(action='VOOOM', voltage=100000)# 2 keywords argument
```

```
-- This parrot wouldn't VOOOM if you put 100000 volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's a stiff !
```

[126]:
```
parrot('a million','bereft of life','jump') # Three positional arguments
```

```
-- This parrot wouldn't jump if you put a million volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's bereft of life !
```

[127]:
```
parrot('a thousand', state = 'pushing up the daisies') # 1 positional, 1␣
↪keyword
```

```
-- This parrot wouldn't voom if you put a thousand volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's pushing up the daisies !
```

Pada sebuah pemanggilan fungsi maka *keyword argument* harus mengikuti *positional argument*. Semua *keyword arguments* yang dilewatkan pada sebuah fungsi harus *match* dengan argumen-argumen yang diterima oleh sebuah fungsi, dan urutan tidak penting. Berikut *function calls* yang tidak valid (silahkan dicek luaran masing-masing):

```
>>> parrot() # required argument missing
>>> parrot(voltage=5.0, 'dead') # non-keyword argument after a keyword argument
>>> parrot(110, voltage=220) # duplicate value for the same argument
>>> parrot(actor='John Cleese') # unknown keyword argument
```

Ketika sebuah formal parameter muncul sebagai argumen dalam bentuk `**name` dengan tipe *dictionary* berisi semua argumen-argumen keyword, kecuali parameter formal lain. Bisa juga dikombinasikan dengan parameter formal `*name` dengan tipe *tuple*, yang berisi *positional arguments* di luar parameter list. Catatan `*name` harus terjadi sebelum `**name`.

[131]:
```
def cheeseshop(kind, *arguments, **keywords):
    print("-- Do you have any", kind, "?")
    print("-- I'm sorry, we're all out of", kind)
    for arg in arguments:
        print(arg)
    print("-" * 40)
    for kw in keywords:
            print(kw, ":", keywords[kw])
```

[132]:
```
cheeseshop("Limburger","It's very runny, sir.", "It's really very, Very␣
↪runny, Sir.",
          shopkeeper ="Michael Palin",
          client = "John Cleese",
          sketch ="Cheese Shop Sketch")
```

```
-- Do you have any Limburger ?
-- I'm sorry, we're all out of Limburger
It's very runny, sir.
It's really very, Very runny, Sir.
----------------------------------------
shopkeeper : Michael Palin
client : John Cleese
sketch : Cheese Shop Sketch
```

Urutan *keyword* yang ditampilkan sesuai dengan yang dimasukan saat *function call*.

List argumen yang sembarang akan disimpan pada tipe data `tuple`. Sebelum argumen sembarang bisa dibuat juga argumen normal.

[133]:
```
def write_multiple_items(file, separator, *args):
    file.write(separator.join(args))
```

Argumen-argumen `Variadic` seharusnya diletakan terakhir setelah formal parameter lain. Jika terdapat formal parameter yang terjadi setelah parameter `*args` hanya *keyword-only*, artinya hanya bisa digunakan sebagai keywords bukan *positional argument*

[141]:
```
def concat(*args, sep="/"):
    return sep.join(args)
```

[142]:
```
concat("earth","mars","Venus")
```

[142]:
```
'earth/mars/Venus'
```

[143]:
```
concat("earth","mars","Venus", sep=".")
```

[143]:
```
'earth.mars.Venus'
```

Kadang-kadang diperlukan untuk *unpack* argumen jika argumen tersebut sudah dalam bentuk `list` atau `tuple`. Sebagai contoh, fungsi *built-in* `range()` biasanya memerlukan argumen terpisah *start* dan *stop*. Jika sudah dalam bentuk list maka bisa digunakan operator `*`:

[144]:
```
list(range(3,6)) #pemanggilan biasa
```

[144]:
```
[3, 4, 5]
```

[145]:
```
args = [3,6]
```

[146]:
```
list(range(*args))
```

[146]:
```
[3, 4, 5]
```

Dengan cara yang sama, *dictionary* dapat menyatakan keyword arguments dengan operator **:

[148]:
```
d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
```

[149]:
```
parrot(**d)
```

```
-- This parrot wouldn't VOOM if you put four million volts through it.
-- Lovely plumage, the Norwegian Blue
-- It's bleedin' demised !
```

[150]:
```
type(d)
```

[150]:
```
dict
```

Fungsi yang kecil bisa dibuat dengan menggunakan `lambda` keyword, tapi terbatas pada satu ekspresi saja.

[151]:
```
def make_incrementor(n):
    return lambda x: x + n
```

[152]:
```
f = make_incrementor(42)
```

[156]:
```
f(0)
```

[156]:
```
42
```

3 Struktur Data

Bisa dilihat beberapa penggunaan *method* pada list:

[171]:
```
Kendaraan =['bemo','motor','mobil','kereta␣
 ↪api','kapal','pesawat','delman','bemo','bemo','pesawat']
```

[172]:
```
Kendaraan.count('bemo')
```

[172]:
```
3
```

[173]:
```
Kendaraan.index('bemo')
```

[173]:
```
0
```

[176]:
```
Kendaraan.index('bemo',2) # mencari 'bemo' dimulai pada posisi 2
```

[176]: 7

[177]:
```
Kendaraan.reverse()
```

[178]:
```
Kendaraan
```

[178]:
```
['pesawat',
 'bemo',
 'bemo',
 'delman',
 'pesawat',
 'kapal',
 'kereta api',
 'mobil',
 'motor',
 'bemo']
```

[179]:
```
Kendaraan.index('bemo')
```

[179]: 1

[180]:
```
Kendaraan.sort()
```

[181]:
```
Kendaraan
```

[181]:
```
['bemo',
 'bemo',
 'bemo',
 'delman',
 'kapal',
 'kereta api',
 'mobil',
 'motor',
 'pesawat',
 'pesawat']
```

[183]:
```
Kendaraan.pop(0)
```

[183]: 'bemo'

[184]:
```
Kendaraan
```

[184]:
```
['bemo',
 'bemo',
 'delman',
 'kapal',
 'kereta api',
 'mobil',
 'motor',
 'pesawat',
 'pesawat']
```

[188]:
```
Kendaraan.remove('bemo')
```

[189]:
```
Kendaraan
```

[189]:
```
['bemo',
 'delman',
 'kapal',
 'kereta api',
 'mobil',
 'motor',
 'pesawat',
 'pesawat']
```

[190]:
```
Kendaraan.remove('pesawat')
```

[191]:
```
Kendaraan
```

[191]:
```
['bemo', 'delman', 'kapal', 'kereta api', 'mobil', 'motor', 'pesawat']
```

3.1 Menggunakan List Sebagai Stacks

Metode-metode pada list membuat mudah untuk dibuat sebagai stack. Untuk menambahkan elemen gunakan `append()` dan untuk mengambil item gunakan `pop()`. Prinsipnya elemen yang ditambahkan terakhir adalah elemen pertama yang akan diperoleh (*last-in, first-out*).

[3]:
```
stack = [3,4,5]
```

[4]:
```
stack.append(6), stack.append(7), stack
```

[4]:
```
(None, None, [3, 4, 5, 6, 7])
```

[5]:
```
stack.pop()
```

[5]:
```
7
```

[6]:
```
stack.pop()
```

[6]:
```
6
```

[7]:
```
stack
```

[7]:
```
[3, 4, 5]
```

3.2 Menggunakan Lists sebagai Queue

Meskipun list bisa digunakan sebagai *queue* (*first-in, first-out*), tetapi list tidak efisien untuk operasi ini (Karena setiap elemen lain harus digeser satu). Untuk implementasi queue, maka gunakan `collections.deque` yang didesain untuk *fast append* dan *pop* dari kedua sisi. Contoh:

[10]:
```
from collections import deque
queue = deque(["erick","john","Michael"])
```

```
queue.append("Terry")
queue.append("Graham")
queue.popleft()
queue.popleft()
queue
```

[10]:
```
deque(['Michael', 'Terry', 'Graham'])
```

3.3 *List Comprehension*

List comprehension menyediakan cara yang singkat untuk membuat sebuah list. Biasa digunakan untuk membuat list baru dimana setiap elemen dihasilkan dari operasi-opreasi yang diaplikasikan pada setiap anggota sekuens lain, atau membuat subsekuens dari eleme-elemen yang memenuhi kondisi tertentu.

[13]:
```
Pangkat2 =[];
```

[14]:
```
for x in range(10):
    Pangkat2.append(x**2)
```

[17]:
```
Pangkat2
```

[17]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Atau bisa juga dilakukan sbb:

[20]:
```
Pangkat2 = [x**2 for x in range(10)] # Disebut dengan list comprehension
```

[21]:
```
Pangkat2
```

[21]:
```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

List comprehension terdiri dari *square brackets* yang berisi sebuah ekspresi diikuti oleh klausul `for`, kemudian diikuti lagi oleh `for` selanjutnya atau `if`. Hasilnya berupa list baru dengan elemen-elemen hasil evaluasi pada `for` atau `if`:

[30]:
```
[(x,y) for x in [1,2,3] for y in [3,1,4] if x!=y]
```

[30]:
```
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Sebagai alternatif dari pernyataan berikut:

[31]:
```
combs =[]
for x in [1,2,3]:
    for y in [3,1,4]:
        combs.append((x,y))
```

[32]:
```
combs
```

[32]:
```
[(1, 3), (1, 1), (1, 4), (2, 3), (2, 1), (2, 4), (3, 3), (3, 1), (3, 4)]
```

[33]:
```
type(combs)
```

[33]:
```
list
```

[34]:
```
combs[0]
```

[34]:
```
(1, 3)
```

[35]:
```
type(combs[0])
```

[35]:
```
tuple
```

Kalau ekspresi berupa *tuple* maka harus di dalam kurung (seperti (x,y) di atas)

[41]:
```
Vektor =[-5,-2.5,0,2.5,5]
```

[42]:
```
[x**2 for x in Vektor]
```

[42]:
```
[25, 6.25, 0, 6.25, 25]
```

[44]:
```
[x**2 for x in Vektor if x >= 0] # Dengan filtering
```

[44]:
```
[0, 6.25, 25]
```

[46]:
```
[abs(x) for x in Vektor] # Apply fungsi terhadap semua elemen.
```

[46]:
```
[5, 2.5, 0, 2.5, 5]
```

[47]:
```
BuahSegar = ['   pisang','  strawberry ','Buah nikmat ']
```

[52]:
```
[buah.strip() for buah in BuahSegar] # memanggil method untuk semua elemen
```

[52]:
```
['pisang', 'strawberry', 'Buah nikmat']
```

[53]:
```
buah
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-53-a59a51894b6f> in <module>
----> 1 buah


NameError: name 'buah' is not defined
```

[54]:
```
buahVar = [buah.strip() for buah in BuahSegar]
```

[55]:
```
buahVar
```

[55]:
```
['pisang', 'strawberry', 'Buah nikmat']
```

[58]:
```
[(x,x**2,x**3) for x in range(6)] # Membuat tuple
```

[58]:
```
[(0, 0, 0), (1, 1, 1), (2, 4, 8), (3, 9, 27), (4, 16, 64), (5, 25, 125)]
```

[59]:
```
Vek = [[1,2,3],[5,6,7],[8,9,10]]
```

[61]:
```
[flatVek for x in Vek for flatVek in x] # Flatten sebuah list
```

[61]:
```
[1, 2, 3, 5, 6, 7, 8, 9, 10]
```

Ekivalen dengan berikut:

[62]:
```
flatVek = []
for x in Vek:
    for y in x:
        flatVek.append(y)
```

[63]:
```
flatVek
```

[63]:
```
[1, 2, 3, 5, 6, 7, 8, 9, 10]
```

List comprehension juga dapat berupa ekspresi kompleks dan *nested functions*

[64]:
```
from math import pi
[str(round(pi,i)) for i in range(1,6)]
```

[64]:
```
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

3.4 Nested *List Comprehension*

Ekspresi awal pada *list comprehension* bisa ekspresi sembarang, termasuk *list comprehension* lain:

[67]:
```
matrix = [[1,2,3,4],
          [5,6,7,8],
          [9,10,11,12]]
```

[68]:
```
matrix
```

[68]:
```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

[69]:
```
[[row[i] for row in matrix] for i in range(4)]
```

[69]:
```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Pernyataan *nested list comprehension* di atas apabila dikerjakan dengan `for` loop akan menjadi:

[72]:
```
transposed =[]
for i in range(4):
    transposed.append([row[i] for row in matrix])
```

[73]:
```
transposed
```

[73]:
```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

yang sama juga dengan 2 `for` statement:

[75]:
```
transposed =[]
for i in range(4):
    # Tiga pernyataan berikut adalah implementasi dari nested
    transposed_row = []
    for row in matrix:
        transposed_row.append(row[i])
    transposed.append(transposed_row)
```

[76]:
```
transposed
```

[76]:
```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Dalam kenyataannya, jikalau ada pilihan dengan *built-in function* maka gunakanlah *built-in function*. Cotoh di atas bisa digunakan *built-in function* `zip()`.

[77]:
```
list(zip(*matrix)) # *matrix --> unpacking argument list
```

[77]:
```
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

3.5 Pernyataan `del`

Untuk *remove* item dari list dengan diberikan indeks-nya bukan harganya, maka bisa digunakan `del`. Pernyataan `del` bisa juga digunakan untuk menghapus *slice* atau menghapus semua.

[100]:
```
a = [-1, 1, 66.25, 333, 333, 1234.5]
```

[101]:
```
del a[0]
a
```

[101]:
```
[1, 66.25, 333, 333, 1234.5]
```

[102]:
```
del a[2:4] # remove slice
```

[103]:
```
a
```

[103]:
```
[1, 66.25, 1234.5]
```

[104]:
```
del a[:] # remove semua
```

[105]:
```
a
```

[105]:
```
[]
```

[106]:
```
del a # del bisa digunakan untuk menghapus variabel juga
```

[107]:
```
a
```

```
␣
↪---------------------------------------------------------------------------

NameError                                 Traceback (most recent call␣
↪last)

<ipython-input-107-3f786850e387> in <module>
----> 1 a


NameError: name 'a' is not defined
```

3.5.1 C.5. Tuple dan Sequence

Data sekuens lain yang bisa dipergunakan selain *list* dan *string* adalah *tuple*. Sebuah *tuple* terdiri dari sejumlah harga-harga yang dipisahkan oleh koma. Contoh:

[113]:
```
t = 12345, 54321, 'hello!'
```

[114]:
```
t[0]
```

[114]:
```
12345
```

[115]:
```
t
```

[115]:
```
(12345, 54321, 'hello!')
```

Tuple bisa juga nested:

[116]:
```
u = t, 1, 2, 4, 7,10
```

[117]:
```
u
```

[117]:
```
((12345, 54321, 'hello!'), 1, 2, 4, 7, 10)
```

[118]:
```
u[0]
```

[118]:
```
(12345, 54321, 'hello!')
```

[119]:
```
u[1]
```

[119]:
```
1
```

```
[120]: u[0][0]
```

```
[120]: 12345
```

```
[122]: u[4]
```

```
[122]: 7
```

```
[123]: u[0][2][0]
```

```
[123]: 'h'
```

```
[124]: t
```

```
[124]: (12345, 54321, 'hello!')
```

```
[125]: t[0]=4567
```

```
␣
↪---------------------------------------------------------------------------

       TypeError                                 Traceback (most recent call␣
↪last)

       <ipython-input-125-3696dc458987> in <module>
   ----> 1 t[0]=4567


       TypeError: 'tuple' object does not support item assignment
```

Tuple adalah *imutable,* tapi dapat berisi objek *mutable* seperti *list*:

```
[134]: v = ([1, 2, 3], [3, 2, 1])
```

```
[135]: type(v)
```

```
[135]: tuple
```

```
[136]: v[0]
```

```
[136]: [1, 2, 3]
```

```
[137]: v[0][1]
```

```
[137]: 2
```

```
[140]: v[0][1]=100  # karena tuple v berisi list yang mutable maka bisa di assign.
```

```
[141]: v
```

```
[141]: ([1, 100, 3], [3, 2, 1])
```