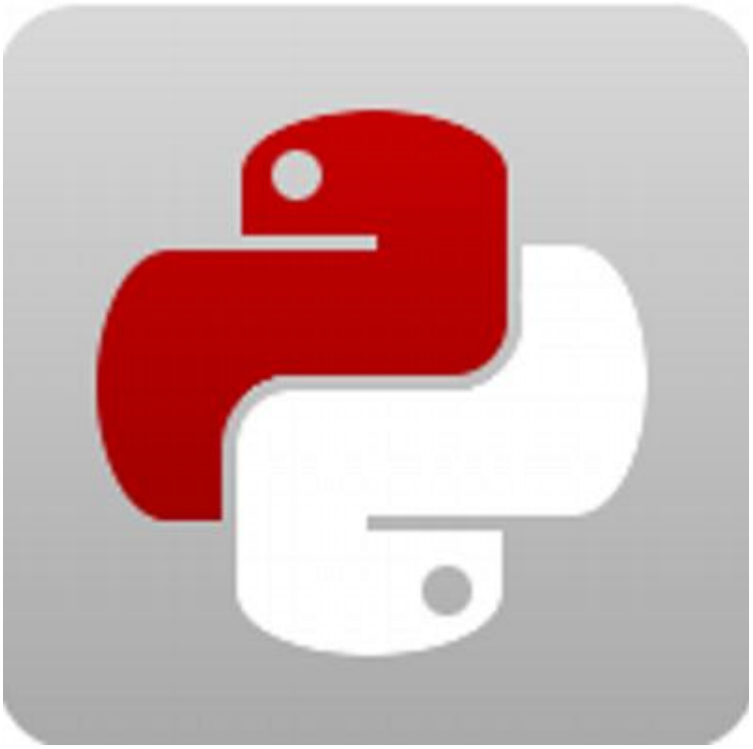
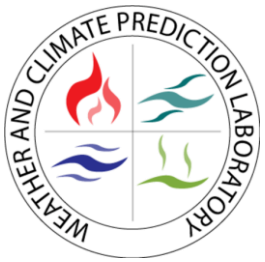
Seri Komputasi



Tutorial Pemrograman Python 2

Untuk Pemula

Sandy H.S. Herho



Tutorial Pemrograman Python 2
Untuk Pemula

Sandy H.S. Herho

WCPL
Weather and Climate Prediction Laboratory

Judul : **Tutorial Pemrograman Python 2 Untuk Pemula**

Penyusun:

Sandy H.S. Herho

Program Studi Meteorologi, Fakultas Ilmu dan Teknologi Kebumian,

Institut Teknologi Bandung

Untuk Mami dan Evelyn,

Semoga tetap tabah menjalani kehidupan

Kata Pengantar

Pada pertengahan tahun 2016, penyusun menyampaikan kepada saya bahwa ia berencana untuk membukukan tutorial tentang Python. Nantinya tutorial ini hendak digunakan sebagai penunjang kuliah dasar pemrograman di Program Studi Meteorologi dan Teknik Geologi. Seketika setelah mendengar rencananya saya takjub serta kagum untuk rencana yang diutarakannya. Terlebih jika melihat pada historis penulisannya dalam beberapa buku filsafat, sejarah, dan paleoklimatologi yang telah ditulisnya, kesemuanya itu disebar bebas melalui akun **academia.edu** dan **researchgate.net** yang dimilikinya. Walau disebar bebas, bukan berarti buku-bukunya tidak bermutu dan murahan, beberapa tulisannya pernah dipakai sebagai bahan ajar tak resmi dalam sebuah institusi pendidikan, selain itu penyusun juga menjadi **top 0.5%** pengguna **academia.edu**. Selain itu menulis buku atau modul pemrograman yang bagus sama sulitnya dengan menelurkan sebuah ide dan pemikiran.

Tak berapa lama kemudian, di awal bulan September 2017 saya menerima *draft* buku tutorial tersebut, lalu kemudian jadilah buku ini, buku yang ada dihadapan anda dan anda pegang sekarang. Buku yang lebih menitik-beratkan kepada teori tentang pemrograman Python jauh dari bayangan saya sebelumnya. Jika diawal dulu, saya membayangkan buku ini akan seperti buku panduan pemrograman yang menjelaskan *how to programming* secara terperinci beserta contoh kode yang tinggal dikopi dan paste atau seperti modul praktikum dengan penjelasan singkat dan lebih banyak latihannya, tidak buku ini tidak seperti itu, buku ini lebih dari itu. Buku ini akan membawa anda menuju level pemahaman yang berbeda setelah anda membacanya. Maka dari itu lanjutkan dan teruskan hingga selesai untuk membacanya.

Mengapa Python?

Menurut data yang dirilis **IEEE Spectrum** pada pertengahan tahun ini, Python menempati posisi pertama dalam **The 2017 Top Programming Languages**, sebuah pemeringkatan yang dilaksanakan oleh **IEEE Spectrum** dipertengahan tahun yang sedang berjalan. Pemeringkatan yang dilakukan oleh **IEEE Spectrum** ini dilakukan dengan mengumpulkan 12 buah data dari 10 *sources* di internet termasuk dari pencarian **Google**, **Twitter**, **Github** hingga dari **IEEE Xplore Digital Library** yang berisi lebih dari 3,6 juta *paper* dan *conference*. Tidak hanya populer di internet, Python juga begitu populer dikalangan limuwan dan *engineer*.

Berbicara dalam dunia kampus, Python juga dalam tahun-tahun terakhir ini telah banyak digunakan untuk menjadi bahasa pemrograman yang digunakan pada kuliah pengantar pemrograman atau pada pengantar ilmu komputer seperti pada kuliah **CS50: Introduction to Computer Science** dari Harvard yang begitu terkenal di Internet serta **6.0001 Introduction to Computer Science and Programming in Python** dari MIT. Pada kuliah yang lebih tinggi atau pada level *research*, banyak persoalan numerik dan komputasi hingga persoalan-persoalan matematikan dan fisika yang telah menggunakan Python dalam penyelesaiannya. Didukung dengan *library* yang beragam dan adanya komunitas yang sangat besar dan saling membantu ketika menghadapi persoalan dengan Python membuat bahasa pemrograman ini terus berkembang.

Sehingga tak salah ketika ***Weather and Climate Prediction Laboratory*** **ITB** mulai berpindah dalam penggunaan bahasa pemrograman yang akan mereka gunakan untuk pembelajaran di kelas dan laboratorium serta untuk penelitian. Sebuah perkembangan yang telah sesuai dengan perkembangan zaman dan tren yang sedang muncul saat ini.

Python ibarat paket lengkap dari sebuah bahasa pemrograman dengan dukungan komunitas dan library. Pengguna tidak hanya dapat menggunakan Python sebagai bahasa pemrograman untuk menyelesaikan persoalan numerik dan analisis data, namun juga dapat mengembangkan program yang terintegrasi. Setelah analisis data dan persoalan numerik diselesaikan dengan Python, pengguna juga dapat membuat visualisasi data dengan berbagai tampilan dari *library-library* yang beragam seperti **matplotlib, vpython** dan lain sebagainya. Lalu mengkoneksikannya ke web dan menjadikan Python sebagai *back-end* servernya. Sebelum itu bahkan dengan berbagai *library* yang ada seperti **scrapy**, **socket** kita dapat berbagi data atau menambang data dari berbagai *source* di internet.

Langkah selanjutnya...

Setelah menyelesaikan membaca buku ini atau setelah anda pembaca selesai mempelajari Python melalui buku ini anda telah sampai pada level penguasaan pemrograman dengan menggunakan Python (dasar-dasar pemrograman Python). Ke depannya anda diharapkan dapat terus mempelajari secara mandiri dan memperdalam lagi kemampuan *programming* anda menggunakan Python dan mengeksplor penggunaan Python di segala bidang sebanyak-banyaknya. Belajar lebih dalam mengenai algoritma dan struktur data pada Python, belajar berbagai metode numerik, belajar bermacam-macam metoda statistik, *machine learning* dan *artificial intelligence* dan lain

sebagainya. Di dalam komunitas Python yang sangat besar di internet semua telah tersedia baik media dan materi pembelajaran, contoh kasus, potongan kode dan panduan-panduan singkat lainnya. Akhir kata saya mengucapkan selamat belajar dan semoga sukses!

Bandung, 12 September 2017

Rio Harapan Ps

***Computational Physicist*, Program Studi Fisika ITB**

http://rhps.github.io

Selayang Pandang

Python merupakan bahasa pemrograman tingkat tinggi yang dewasa ini telah menjadi standar dalam dunia komputasi ilmiah. Python merupakan bahasa pemrograman *open source* multi-*platform* yang dapat digunakan pada berbagai macam sistem operasi (Windows, Linux, dan MacOS). Selain itu, Python juga merupakan bahasa pemrograman yang fleksibel dan mudah untuk dipelajari. Program yang ditulis dalam Python umumnya lebih mudah dibaca dan jauh lebih ringkas dibandingkan penulisan program dalam bahasa C atau Fortran. Python juga memiliki modul standar yang menyediakan sejumlah besar fungsi dan algoritma, untuk menyelesaikan pekerjaan seperti mengurai data teks, memanipulasi dan menemukan file dalam *disk*, membaca / menuliskan file terkompresi, dan mengunduh data dari server web. Dengan menggunakan Python, para *programmer* juga dapat dengan mudah menerapkan teknik komputasi tingkat lanjut, seperti pemrograman berorientasi objek.

Bahasa pemrograman Python dalam banyak hal berbeda dengan bahasa pemrograman prosedural, seperti C; C++; dan Fortran. Dalam Fortran, C, dan C++, file *source code* harus dikompilasi ke dalam bentuk *executable file* sebelum dijalankan. Pada Python, tidak terdapat langkah kompilasi, sebagai gantinya *source code* ditafsirkan secara langsung baris demi baris. Keunggulan utama dari suatu bahasa pemrograman terinterpretasi seperti Python adalah tidak membutuhkan pendeklarasian variabel, sehingga lebih fleksibel dalam penggunaannya. Namun, terdapat kelemahan yang mencolok, yaitu program – program numerik yang dijalankan pada Python lebih lambat ketimbang dijalankan menggunakan bahasa pemrograman terkompilasi. Kelemahan ini tentu membuat kita berpikir apakah Python cocok untuk digunakan dalam komputasi ilmiah? Meskipun bekerja dengan agak lambat, Python memiliki banyak fungsi – fungsi sederhana yang dapat menjalankan hal – hal yang umumnya dikerjakan dengan *subroutine* rumit dalam C dan/atau Fortran. Sehingga Python merupakan pilihan tepat dalam komputasi ilmiah dewasa ini.

Untungnya, banyak *routine* numerik dan matematis umum yang telah disusun sebelumnya, yang dikelompokan ke dalam dua buah paket (NumPy dan SciPy) yang dapat diimpor secara mudah ke dalam Python. Paket NumPy (*Numerical Python*) menyediakan banyak *routine* dasar guna memanipulasi *array* dan matriks numerik dalam skala besar. Paket SciPy (*Scientific Python*) memperluas kegunaan NumPy dengan kumpulan algoritma ilmiah yang sangat berguna, seperti minimalisasi; transformasi Fourier; regresi; dan banyak teknik – teknik aplikasi matematis lainnya.

Karena bersifat *open source*, kedua paket ini sangat populer di kalangan ilmuwan. Dengan adanya paket NumPy dan SciPy, Python dapat berdiri sejajar (bahkan di atas) bersama program komputasi ilmiah Matlab dalam penggunaannya sebagai alat bantu sains dewasa ini.

Tutorial ini menggunakan bahasa Python 2.7. Kendati penggunaan Python 3 sudah mulai meluas, hingga kini dunia komputasi ilmiah masih menggunakan bahasa Python 2 sebagai bahasa standar mereka. Python 2.7 merupakan bahasa Python 2 versi terakhir dan rencananya akan tetap dipelihara hingga tahun 2020.

Instalasi

Untuk mempelajari tutorial ini, hal yang pertama harus kalian lakukan adalah melakukan instalasi interpreter dasar. Selain itu, kini juga terdapat berbagai macam aplikasi editor yang bersifat *GUI-driven* untuk menuliskan program Python secara lebih mudah. Pada platform Windows misalnya, saya cenderung menggunakan Anaconda yang tersedia secara gratis dan menyertakan editor Spyder didalamnya. Paket tunggal ini menyertakan *tools* yang dibutuhkan dalam komputasi ilmiah. Anaconda dapat kalian unduh di https://www.continuum.io. Kemudian ikuti langkah instalasi hingga selesai. Namun, jika kalian ingin menghemat memori, kalian tidak perlu menginstal seluruh paket *add-on* pada saat melakukan instalasi. Paket – paket utama yang wajib diinstal untuk komputasi ilmiah antara lain adalah Python, NumPy, SciPy, Spyder, IPython, dan matplotlib.

Jika kalian tidak ingin menggunakan Anaconda, kalian harus menginstal perangkat lunak sebagai berikut:

- Python di http://www.python.org/. Pastikan bahwa kalian meninstal Python 2.7.
- NumPy untuk Python 2.7 di http://www.scipy.org/.
- SciPy untuk Python 2.7 di http://www.scipy.org/.
- matplotlib untuk Python 2.7 di http://www.matplotlib.org/.

Sumber – Sumber Pembelajaran

Terdapat banyak sekali dokumentasi Python yang dapat digunakan sebagai bahan pembelajaran. Pada editor Spyder, kalian dapat menggunakan perintah **`Help (?) > Python Documentation`** untuk mengakses dokumentasi. Di samping itu, terdapat juga manual lengkap beserta dokumentasi – dokumentasi yang sangat bermanfaat untuk membantu pembelajaran kalian yang dapat di akses di http://docs.python.org .

Komunitas *developer* Python juga memelihara wiki mereka sendiri. Khususnya bagi kalian para *programmer* pemula, terdapat beberapa halaman di wiki tersebut yang sekiranya dapat kalian manfaatkan di http://wiki.python.org/moin/BeginnersGuide .

Bagi kalian yang sudah terbiasa dengan Python, dan ingin mendalaminya, kalian dapat mengakses situs Dive into Python yang menyediakan tutorial penggunaan Python tingkat lanjut. Dive into Python dapat kalian akses di http://www.diveintopython.org/ .

Interpreter Interaktif

Untuk memulai penggunaan Python, kalian dapat menuliskan perintah "python" pada *command prompt* atau terminal kalian. Kemudian kalian akan menjumpai tampilan sebagai berikut (khusus bagi pengguna Anaconda):

```
Python 2.7.13 |Anaconda 4.4.0 (64-bit)| (default, May 11 2017, 13:17:26) [MSC v.
1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
Anaconda is brought to you by Continuum Analytics.
Please check out: http://continuum.io/thanks and https://anaconda.org
>>>
```

Tanda `>>>` mengindikasikan bahwa Python menunggu *input* perintah yang hendak kalian masukkan. Tidak dibutuhkan proses kompilasi saat kalian menuliskan program pada interpreter interaktif. Python dapat membaca perintah dan langsung memberikan respon, contohnya:

```
>>> 1
1
```

Seperti yang akan kita bahas pada bagian – bagian berikutnya, Python juga dapat membaca *script* (file yang berisi daftar perintah yang telah dituliskan sebelumnya dan dieksekusi dalam suatu sikuen). Dengan pengecualian pada penyembunyian *output* pada saat membaca file, pada dasarnya tidak terdapat perbedaan cara Python memperlakukan suatu perintah dalam penulisan interaktif maupun dalam *script*. Kelebihan penulisan secara interaktif adalah kita dapat menguji perintah – perintah dalam program yang kita buat sudah benar.

Pada Python, teks yang dituliskan sesudah tanda "#" akan dianggap sebagai komentar program. Python akan mengabaikan teks sesudahnya hingga akhir baris tersebut Contoh penggunaannya:

```
>>> 1 # tekan nomor 1
1
```

Perintah panjang pada Python dapat dibagi ke dalam beberapa baris menggunakan perintah "\". Ketika kita menggunakan perintah ini, baris berikutnya harus memuat jumlah spasi yang sama dengan baris sebelumnya. Spasi dalam Python bersifat sintaktis, kita akan membicarakan hal ini secara lebih lanjut pada bagian – bagian berikutnya.

```
>>> 1.243 + (3.42839 – 4.394834) * 2.1 \
... + 4.587 – 9.293 + 34.234 \
... – 6.2 + 3.4
```

Kalian juga dapat memanfaatkan tanda kurung untuk melanjutkan perintah pada baris selanjutnya tanpa direpotkan oleh tanda "\", sebagai berikut:

```
>>> (1.243 + (3.42839 – 4.394834) * 2.1
... + 4.587 – 9.293 + 34.234
... – 6.2 + 3.4)
```

Umumnya penggunaan tanda kurung ini lebih disukai oleh para *programmer* Python ketimbang tanda ”\”.

Meskipun jarang digunakan, berikut diberikan contoh penggunaan tanda “;” untuk menjalankan perintah – perintah yang berbeda dalam satu baris.

```
>>> 1 + 4 ; 6 – 2
5
4
```

Sebisa mungkin hindari penggunaan tanda “;” dalam program kalian karena dapat mengurangi reputasi keterbacaan program kalian oleh orang lain.

Terdapat fungsi **`help`** dalam Python yang dapat membantu kalian dalam hampir segala hal. Berikut ini diberikan contoh andaikan kita ingin mencari tahu penggunaan dari fungsi **`sun`**.

```
>>> help(sum)
Help on built-in function sum in module __builtin__:
sum(...)
sum(sequence, start=0) -> value
Returns the sum of a sequence of numbers (NOT strings) plus the value
of parameter 'start'. When the sequence is empty, returns start.
```

Fungsi **`help`** bahkan dapat bekerja pada fungsi dan variabel yang kalian buat sendiri. Python menyediakan cara yang sangat mudah guna menambahkan teks deskriptif yang dapat digunakan oleh fungsi **`help`**. Untuk lebih jelasnya akan kita bahas nanti.

Python merupakan bahasa pemrograman *case sensitive*. Maka, kita harus mendefinisikan variabel dan fungsi secara benar agar dapat dikenali. Untuk lebih jelasnya, perhatikan contoh pendeklarasian variabel di bawah ini:

```
>>> Var = 1
>>> var = 2
>>> Var
1
>>> var
2
```

Untuk keluar dari interpreter interaktif, kita dapat menggunakan fungsi `exit()`.

```
>>> exit()
c:\>
```

Segalanya Merupakan Objek

Dalam Python, nilai, *list*, kelas, dan fungsi merupakan objek Sebagai sebuah objek, maka terdapat fitur – fitur dan fungsi – fungsi khas yang dapat diakses dengan menggunakan notasi titik, contohnya:

```
>>> s = "halo"
>>> s.capitalize()
'Halo'
>>> s.replace("lo", "i")
'hai'
```

Kita juga dapat menggunakan notasi titik pada *string* itu sendiri secara langsung.

```
>>> "halo".capitalize()
'Halo'
```

Fakta bahwa segala sesuatu dalam Python merupakan objek menjadikan Python sebagai bahasa pemrograman dengan fleksibilitas tinggi. Dalam pemrograman berorientasi objek, kode program akan dipecah ke dalam bagian – bagian yang disebut kelas. Melalui pemrograman berorientasi objek, kode program yang kita buat akan lebih mudah untuk dikembangkan dan dirawat.

Tipe – Tipe Numerik

Bilangan tanpa koma didefinisikan sebagai tipe *integer*.

```
>>> type(13)
<type 'int'>
```

Fungsi **type** berguna untuk mengetahui tipe argumen yang kita gunakan. Melalui *return value* dalam perintah ini, kita mengetahui bahwa "13" merupakan tipe Python "int" yang berarti suatu *integer*. Setiap *integer* biasa membutuhkan ruang memori sebesar 4 bit, dan dapat bervariasi pada rentang -2147483648 hingga 2147483647. Di sisi lain, *integer* dalam jumlah besar dalam Python diklasifikasikan sebagai tipe *integer* tersendiri yang disebut sebagai 'long'.

```
>>> type(10000000000)
<type 'long'>
```

Operasi numerik yang melibatkan *integer* dalam jumlah besar membutuhkan lebih banyak memori dibandingkan operasi pada *integer* biasa.

Pada operasi numerik umumnya kita menggunakan angka normal dengan titik desimal dalam Python dikenal sebagai tipe 'float'.

```
>>> type(1.)
<type 'float'>
```

Tipe 'float' dalam Python merupakan bilangan riil berpresisi ganda. Rentangnya bergantung pada perangkat keras yang digunakan, namun umumnya berkisar antara 10^{-308} hingga 10^{308} atau 14 buah bilangan desimal dibelakangnya.

Python juga dapat menangani operasi bilangan kompleks. Notasi "j" menandai bagian bilangan imajiner.

```
>>> type(1+2j)
<type 'complex'>
```

Contoh operasi menggunakan bilangan kompleks pada Python antara lain berikut ini:

```
>>> (1+2j)*(1-2j)
(5+0j)
```

Perlu kalian catat, bilangan kompleks pada Python wajib dituliskan dalam tanda kurung.

Pada setiap tipe dalam Python, terdapat fungsi ekivalen yang dapat mengubah nilainya ke dalam tipe lain. Untuk lebih jelas, perhatikan contoh berikut:

```
>>> int(3.2)
3
>>> float(2)
2.0
>>> complex(1)
(1+0j)
```

Selain itu, terdapat juga fungsi **round** yang dapat digunakan untuk pembulatan ke nilai *integer* terdekat. Perhatikan contoh berikut ini:

```
>>> int(0.8)
0
>>> round(0.8)
1.0
>>> int(round(0.8))
1
```

Menggunakan Python Sebagai Kalkulator

Penjumlahan dua buah bilangan:

```
>>> 1+1
2
```

Pembagian bilangan *integer*:

```
>>> 8/3
2
```

Pembagian bilangan desimal akan menghasilkan nilai desimal pula, meskipun salah satu bilangannya merupakan *integer*. Pada saat melakukan operasi matematika, Python melakukan konversi nilai *output* sesuai dengan tipe nilai *input* yang paling presisi. Berikut ini contohnya:

```
>>> 8./3
2.6666666666666665
```

Operasi pangkat menggunakan operator "**":

```
>>> 8**2
64
```

Perhatikan contoh perhitungan berikut ini, di mana nilai menjadi 1 akibat pangkatnya berupa pembagian *integer*:

```
>>> 8**(1/2)
1
```

Operasi modulo juga dapat dilakukan pada Python dengan menggunakan operator "%" yang digunakan untuk mendapatkan sisa pembagian:

```
>>> 8 % 3
2
>>> 4 % 3.
1.0
>>> 8**0.5
2.8284271247461903
```

Tipe *Boolean* dan Operasinya

Operator standar yang digunakan dalam operasi tipe *boolean* adalah perbandingan antara dua buah nilai. Hasilnya adalah konstanta "True" atau "False":

```
>>> 1 > 6
False
>>> 2 <= 2
True
```

Untuk operasi kesamaan antara dua buah nilai, Python menggunakan operator "==". Operator untuk operasi 'bukan' adalah "not".

```
>>> 1 == 2
False
```

Untuk operasi ketidaksamaan digunakan operator "!=":

```
>>> 2 != 5
True
```

Atau sebagai alternatif kita dapat menggunakan operator “not” seperti yang tadi telah dijelaskan:

```
>>> not 2 == 5
True
```

Sebagaimana telah kita ketahui, konstanta “True” dan “False” dalam tipe *boolean* memiliki nilai numerik masing – masing 1 dan 0. Operator – operator logika juga dapat digunakan pada operasi seperti contoh berikut ini:

```
>>> (2 > 1) and (5 < 8)
True
>>> (2 > 1) or (10 < 8)
True
>>> (not 5==5) or (1 > 2)
False
>>> ((not 3 > 2) and (8 < 9)) or (9 > 2)
True
```

Pengubahan Nilai Variabel

Dalam Python, variabel dapat diubah nilainya. Tidak seperti pada kebanyakan bahasa pemrograman lainnya, tipe variabel dalam Python tidak perlu dideklarasikan terlebih dahulu. Dalam Python tipe bersifat dinamis, yang berarti tipe suatu variabel dapat diubah sepanjang program:

```
>>> a = 1
>>> a
1
>>> b = 2
>>> b == a
False
```

Nilai suatu variabel numerik dapat ditambahkan dan dikurangi menggunakan operator “+=” dan “-=”:

```
>>> a = 1
>>> a = a + 1
>>> a
2
>>> a += 1
>>> a
3
>>> a -= 3
>>> a
0
```

Operator yang mirip juga dapat digunakan pada operasi pengalian dan pembagian:

```
>>> a = 2
>>> a *= 4
>>> a
8
>>> a /= 3.
>>> a
2.6666666666666665
```

Perhatikanlah, pada baris terakhir terjadi perubahan tipe variabel dari “int” ke “float”, akibat pembagian terhadap bilangan desimal.

Tipe *String*

Salah satu keunggulan Python adalah kemampuannya bekerja pada tipe *string*. Dalam Python, *string* direpresentasikan oleh tipe ‘str’. Tipe data *string* dituliskan di antara dua tanda petik tunggal atau ganda atau kutip ganda tiga kali. Berikut contoh pemakaiannya:

```
>>> s = "adimanusia"
>>> print s
adimanusia
>>> s = ‘adimanusia’
>>> print s
adimanusia
>>> s = “””adimanusia”””
>>> print s
adimanusia
```

Penulisan dengan menggunakan tanda kutip dua terkadang berguna jika kita hendak menuliskan kata – kata seperti pada contoh berikut ini:

```
>>> s = "Sandy's climbing shoes"
```

Kita juga dapat melakukan penggabungan antara dua buah kata pada data *string* ini:

```
>>> "Sandy " + 'Herho'
'Sandy Herho'
```

Operator perkalian juga dapat digunakan untuk melakukan pengulangan data *string*:

```
>>> s = "Komando! "*3
>>> s
'Komando! Komando! Komando! '
```

Fungsi `len` dapat digunakan untuk menghitung panjang suatu data *string*, atau dalam konteks ini merupakan jumlah karakter yang ditulis. Contohnya:

```
>>> len("Iban Aso")
8
```

Seperti yang telah dijelaskan sebelumnya, tanda kutip dua tiga kali juga dapat digunakan untuk menuliskan data *string*. Tanda ini juga berguna ketika kalian hendak menuliskan kalimat panjang yang terbagi dalam beberapa baris. Contohnya:

```
>>> s = """Waktuku belum tiba, segelintir orang terlahir ketika ia mati.
... Mungkin suatu hari nanti, akan berdiri sebuah lembaga terhormat tempat para
... Guru Besar hidup dan meneliti sebagaimana aku hidup dan meneliti, bahkan
... mungkin suatu saat nanti akan diadakan berbagai diskusi ilmiah tentang
... Paleoklimatologi di Prodi Meteorologi ITB.”””
>>> print s
Waktuku belum tiba, segelintir orang terlahir ketika ia mati.
Mungkin suatu hari nanti, akan berdiri sebuah lembaga terhormat tempat para
Guru Besar hidup dan meneliti sebagaimana aku hidup dan meneliti, bahkan
mungkin suatu saat nanti akan diadakan berbagai diskusi ilmiah tentang
Paleoklimatologi di Prodi Meteorologi ITB.
```

Terdapat juga metode untuk melakukan pemeriksaan *substring* yang ada dalam suatu string, berikut contohnya:

```
>>> "Atmosphaira" in "HMME ‘Atmosphaira’"
True
>>> "PKS dan HTI" in "NKRI"
False
```

Karakter – Karakter Khusus Dalam *String*

Format – format khusus pada tipe data string, seperti jeda antar baris (*line break*), *tab*, dan masih banyak lagi dapat dikerjakan dengan menuliskan karakter “\”. Untuk membuat sebuah jeda antar baris kita dapat menyisipkan karakter “\n”. Berikut contohnya:

```
>>> print "Kristo berambut\nKribo"
Kristo berambut
Kribo
```

Untuk format *tab* kita dapat menuliskan karakter “\t”:

```
>>> print "Kristo berambut\tKribo"
Kristo berambut       Kribo
```

Ketika kita ingin menuliskan kalimat dengan tanda petik, gunakan karakter “\’”, dan “\””. Lebih jelasnya, perhatikan contoh berikut ini:

```
>>> print "Herho\'s student said, \"I like this course.\""
Herho's student said, "I like this course."
```

Karena tanda garis miring merupakan karakter khusus yang penggunaannya kita lihat di atas, maka untuk memasukkan karakter garis miring dalam suatu karakter dalam teks *string*, pergunakanlah garis miring dua kali. Berikut contohnya:

```
>>> print "Penggunaan karakter garis miring \\."
Penggunaan karakter garis miring \.
```

Ada kalanya kita membutuhkan penulisan karakter – karakter khusus ini untuk penulisan teks *string* biasa. Untuk mengabaikan fungsi dari karakter khusus tersebut, pergunakanlah karakter "r":

```
>>> print r"String ini tidak memngenali format \t dan \n."
String ini tidak mengenali format \t dan \n.
```

Format Dalam *String*

Data bertipe numerik dapat diubah menjadi tipe *string* dengan menggunakan fungsi **`str`**.

```
>>> str(1)
'1'
>>> str(1.0)
'1.0'
>>> str(1+2j)
'(1+2j)'
```

Python juga memiliki sintaks "%" untuk memberikan kita kontrol yang lebih pada tipe data *string*. Berikut ini contohnya:

```
>>> s = "Nilai pi adalah%8.3f" %3.141591
>>> print s
Nilai pi adalah    3.142
```

Angka 8 merupakan jumlah karakter *string* yang disediakan untuk hasil konversi *float* tersebut, sedangkan angka 3 menunjukkan jumlah bilangan sesudah koma yang dikonversi dalam *string* tersebut. Kita juga dapat menghilangkan angka didepan tanda "%" guna menghilangkan spasi. Python tetap memberikan ruang kosong minimum guna penulisan angka yang dikehendaki. Untuk lebih jelasnya coba kalian lihat contoh berikut ini:

```
print "Nilai pi adalah %.3f."% 3.141591

Nilai pi adalah 3.142
```

Python juga dapat membuat nilai desimal pada *string* jika spesifikasi panjang telah kita jabarkan dalam perintahnya. Berikut ini contohnya:

```
>>> print "%8.3f" % 10. + "\n" + "%8.3f" % 100.

       10.000

      100.000
```

Untuk membuat angka – angka tersebut menjadi rata kiri, gunakan tanda "-" sesudah operator "%".

```
>>> print "%-8.3f" % 10. + "\n" + "%-8.3f" % 100.
10.000
100.000
```

Kita juga dapat menempatkan angka 0 didepan bilangan desimal yang diubah menjadi *string*, melalui cara sebagai berikut:

```
>>> print "%08.3f" % 100.
0100.000
```

Python menyediakan banyak cara untuk melakukan pengaturan format tipe *float* yang diubah ke dalam bentuk *string*. Selain menggunakan operator "f", kita juga dapat menggunakan operator "e" untuk mengekspresikan notasi eksponensial. Lihat contoh berikut ini:

```
>>> print "%10.3e" % 1024.
1.0240e+003
```

Kita juga dapat melakukan pengaturan *integer* dalam *string* dengan menggunakan operator "i" atau "d". Secara *default*, Python cenderung melakukan pembulatan ke angka di bawahnya ketimbang melakukan pembulatan secara benar saat mengerjakan operasi ini. Berikut ini contohnya:

```
>>> print "%i" % 3.7
3
```

Format eksponensial dapat digunakan secara sama baik pada *integer*, maupun pada *float*:

```
>>> print "%.4e" % 238482
2.3848e+005
```

Nilai numerik yang berlainan tipe dapat kita format ke dalam satu data *string* yang sama. Untuk melakukannya kalian tinggal mengikuti format yang telah dijabarkan sebelumnya, kemudian mengakhirinya dengan tanda dalam kurung yang memuat nilai format pertama yang diikuti oleh nilai format selanjutnya. Untuk lebih jelas perhatikan contoh berikut ini:

```
>>> print "Saya sudah tingkat %i dan IP saya masih %.3f." % (4, 1.83)
Saya sudah tingkat 4 dan IP saya masih 1.830.
```

Nilai sesudah operator "%" sesungguhnya merupakan tipe *tuple* yang tidak akan kita bahas di sini. Pada pembahasan nanti, kita akan membahas tipe *tuple* secara mendetail.

Kita dapat melakukan pemformatan pada nilai *string* sendiri, dengan menggunakan operator "s", seperti berikut ini:

```
>>> print "Pemilik rumah %.1f kali %.1f ini adalah Widji %s." % (2, 3, "Thukul")
Pemilik rumah 2.0 kali 3.0 ini adalah Widji Thukul.
```

Jika kita ingin menentukan lebar spesifikasi format dengan menggunakan nilai variabel, maka operator "*" dapat digunakan untuk menggantikan nilai lebar dan *integer* tambahan, mendahului nilai yang akan diformat dalam *tuple* berikutnya. Berikut contohnya:

```
>>> a = 10
>>> print "%0*i" % (a, 12345)
0000012345
```

Ada kalanya kita membutuhkan tanda "%" untuk dimasukkan dalam data *string* yang hendak kita tuliskan bersama format tertentu. Pada kasus ini, guna mematikan karakter khusus "%", kita gunakan tanda "%%". Untuk lebih jelas perhatikan contoh di bawah ini:

```
>>> print "Pada %i bungkus nasi padang, terkandung 8%% ganja." %1
Pada 1 bungkus nasi padang, terkandung 8% ganja.
```

Tipe *List*

Kemampuan Python untuk memanipulasi variabel dan objek dalam *list* merupakan salah satu keunggulan bahasa pemrograman ini. Terdapat dua macam objek *list* dalam Python, yaitu *tuple* dan *list*. Perbedaannya adalah, pada *tuple* daftar yang dituliskan bersifat tetap dan tidak dapat diubah, sedangkan pada *list* kita dapat menambah dan menghapus daftar; melakukan penyusunan; dan beberapa jenis modifikasi lainnya. Pemrosesan pada *tuple* umumnya lebih cepat dibandingkan pemrosesan *list*, namun karena kemudahan pada proses modifikasi, aturan praktis yang umumnya dianut oleh para *programmer* Python adalah selalu gunakan *list*.

List dapat kita buat dengan menggunakan tanda "[]". Perhatikan contoh berikut:

```
>>> l = [1,2,3,4,5]
>>> print l
[1, 2, 3, 4, 5]
```

List dalam jumlah banyak dapat dibagi menjadi beberapa baris tanpa harus menggunakan operator "\", namun jangan lupa untuk memperhatikan indentasi elemen *list* kalian. Berikut contohnya:

```
>>> l = [1, 2, 3,
... 4, 5, 6,
... 7, 8, 9]
...
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Kita juga dapat merangkai *list* yang berbeda dengan menggunakan operator penjumlahan:

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
```

Perhatikanlah bahwa penjumlahan di atas bukanlah penjumlahan vektor, melainkan hanya penambahan elemen pada *list* pertama. Untuk melakukan penjumlahan vektor, kalian dapat melihatnya pada penggunaan *array* dalam tutorial NumPy.

Untuk melakukan pengulangan elemen – elemen dalam *list*, kita dapat menggunakan operator perkalian:

```
>>> [1,2]*3

[1, 2, 1, 2, 1, 2]
```

Operator penambahan seperti yang telah kita bahas pada tipe data numerik. Contohnya adalah sebagai berikut:

```
>>> l1 = [1,2,3]

>>> l2 = [4,5,6]
>>> l1 += l2
>>> l1
[1, 2, 3, 4, 5, 6]
```

Fungsi `range` dapat kita manfaatkan untuk menuliskan *list* yang berbentuk deret bilangan. Bentuk umumnya adalah `range(start, stop, step)`, di mana argumen pertama dan terakhir bersifat

opsional. Kalian harus mengetahui jika fungsi `range` selalu dimulai dari angka nol jika tidak didefinisikan titik awalnya, dan titik akhirnya tidak termasuk dalam `range` tersebut. Berikut contohnya:

```
>>> range(4)
[0, 1, 2, 3]
>>> range(1, 4)
[1, 2, 3]
>>> range(0, 8, 2)
[0, 2, 4, 6]
```

Panjang dari suatu *list* dapat kita ketahui dengan menggunakan fungsi `len`:

```
>>> len([1, 3, 5, 7])
4
```

Akses Terhadap Elemen – Elemen *List*

Elemen – elemen *list* dapat kita akses menggunakan operator "[]", sebagaimana contoh berikut ini:

```
>>> l = [1,4,7]
>>> l[0]
1
>>> l[2]
7
```

Perhatikanlah, bahwa elemen pertama dalam *list* berindeks 0, dan indeks elemen akhir pada *list* merupakan jumlah elemen dalam *list* dikurangi 1. Hal ini berbeda dengan pemrosesan pada Fortran, tapi mirip dengan pemrosesan pada C dan C++. Indeks pada seluruh objek – objek dalam sikuen pada Python (*list*, *tuple*, dan *array*), dimulai dari indeks 0.

Jika kita menuliskan indeks di luar jangkauan data, kita akan mendapati:

```
>>> l[3]
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

Kita juga dapat mengubah elemen pada *list* pada indeks tertentu dengan memasukan nilai yang kita kehendaki pada indeks tertentu. Berikut contohnya:

```
>>> l = [1,4,7]
>>> l[0] = 5
>>> l
[5, 4, 7]
```

Indeks – indeks negatif dapat kita gunakan jika kita ingin mengindeks dari bagian akhir *list* (dimulai dari indeks -1). Berikut contohnya:

```
>>> l = [1,4,7]
>>> l[-1]
7
>>> l[-3]
1
```

List dapat dibagi menggunakan menggunakan fungsi sebagai berikut `l[lower:upper:step]`, di mana `lower` merupakan indeks objek pertama yang akan kita gunakan, `upper` menunjukkan sampai indeks paling akhir yang hendak kita potong (di mana indeks terakhir tidak termasuk, jadi dikurangi satu), dan `step` menunjukkan rentang yang dilewati dan bersifat opsional. Berikut contohnya:

```
>>> l = [1,2,3,4,5]
>>> l[0:4]
[1, 2, 3, 4]
>>> l[0:4:2]
[1, 3]
```

Jika kita tidak menuliskan bagian `lower`, maka *default* –nya adalah indeks 0, sedangkan jika kita tidak menuliskan bagian `upper`, maka *default* –nya adalah indeks terakhi dalam *list* tersebut. Contohnya adalah:

```
>>> l = [1,2,3,4,5]
>>> l[:4]
[1, 2, 3, 4]
>>> l[2:]
[3, 4, 5]
>>> l[::2]
[1, 3, 5]
```

Kita juga dapat menggunakan indeks negatif untuk memotong *list*, contohnya:

```
>>> l=[1,2,3,4,5,6,7,8,9,10]
>>> l[-3:]
[8, 9, 10]

>>> l[:-2]
[1,2,3,4,5,6,7,8]
```

Dalam pemotongan ini, besar indeks akhir yang melebihi *list* yang hendak dipotong tidak dikenali sebagai *error*, melainkan akan disesuaikan dengan ukuran akhir *list* tersebut. Berikut contohnya:

```
>>> l = [4,2,8,5,2]
>>> l[2:10]
[8, 5, 2]
>>> l[-10:3]
[4, 2, 8]
```

Penggunaan *List Comprehension*

Python menyediakan sintaks – sintaks yang dapat kita akses dengan mudah guna membuat *list* baru dari *list*, atau *tuple* yang sudah ada, serta membuat objek – objek iterasi lainnya. Metode ini dikenal sebagai *list comprehension* yang mempunyai bentuk umum sebagai berikut:

```
>>> [expression for object in iterable]
```

Sebagai contoh, kita buat *list* kuadrat *integer* berikut ini:

```
>>> [i*i for i in range(5)]
[0, 1, 4, 9, 16]
```

Pada ekspresi di atas, elemen – elemen dalam *list* dibuat melalui fungsi `range` yang diterapkan pada objek `i`. Perlu kalian ingat, bahwa ketika Python akan selalu membuat *list* baru dengan metode *list comprehension*, dan *list* sebelumnya tidak akan mengalami perubahan.

Berikut ini adalah contoh – contoh *list comprehension* lainnya:

```
>>> [k*5 for k in [4,8,9]]
[20, 40, 45]
>>> [q**(0.5) for q in (4,9,16)]
[2.0, 3.0, 4.0]
>>> [k % 2 == 0 for k in range(5)]
[True, False, True, False, True]
>>> [character for character in "Python"]
['P', 'y', 't', 'h', 'o', 'n']
```

List dapat membuat lebih dari satu komponen *iterable*. Python akan menjalankan *iterable* yang paling kanan terlebih dahulu, karena lebih cepat penyelesaiannya. Contohnya adalah sebagai berikut:

```
>>> [[j,k] for j in range(4) for k in range(j+1,4)]
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

Kita juga dapat menggunakan fungsi **`if`** guna memfilter elemen pada *list comprehension*. Bentuk umumnya adalah:

```
>>> [expression for object in iterable if condition]
```

Sebagai contoh, di sini ditampilkan bagaimana kita membuat *list* dari *sublist* sebagaimana contoh ini:

```
>>> [[j,k] for j in range(4) for k in range(4) if j < k]
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
```

Berikut contoh lainnya, bagaimana kita memfilter kata dengan huruf "i" pada *list*:

```
>>> [s for s in ["iban", "anu", "yari", "jeki", "fikri", "aufa"] if "i" in s]
['iban', 'yari', 'jekri', 'fikri']
```

Contoh di atas juga dapat diterapkan dengan mengambil huruf pertama pada masing – masing elemen dengan huruf "i" di bawah ini:

```
>>> [s[0] for s in ["iban", "anu", "yari", "jeki", "fikri", "aufa"] if "i" in s]
['i', 'y', 'j', 'f']
```

Operasi dan Fungsi Pada Tipe *List*

List dapat memuat berbagai macam tipe objek dalam Python, bahkan dapat memuat *list* lainnya dalam suatu *list*. Berikut ini contohnya:

```
>>> l = [1., 2, "tiga", [4, 5, 6]]
>>> l[2]
'tiga'
>>> l[3]
[4, 5, 6]
```

Indeks ganda juga dapat kita terapkan untuk mengakses elemen *list* dalam *list*:

```
>>> 3 in [1, 2, 3]
True
>>> 4 in range(4)
False
```

Kita juga dapat memeriksa nilai atau objek dalam *list*:

```
>>> l = [1., 2, "tiga", [4, 5, 6]]
>>> l[3][1]
5
```

Kita juga dapat menghapus elemen dalam *list*:

```
>>> l = [1,2,3,4]
>>> del l[1]
>>> l
[1, 3, 4]
```

Penghapusan elemen dalam *list* dapat menggunakan notasi pemotongan.

```
>>> l = range(5)
>>> del l[1:3]
>>> l
[0, 3, 4]
```

Elemen pertama dengan nilai tertentu dalam *list* dapat kita hapus dengan menggunakan fungsi **`remove`**. Berikut contohnya:

```
>>> l = [1, 2, 3, 2, 1]
>>> l.remove(2)
>>> l
[1, 3, 2, 1]
```

Kita juga dapat menambahkan elemen pada lokasi tertentu dalam *list* menggunakan fungsi **`insert(index, value)`**, atau dengan menggunakan notasi pemotongan. Berikut contohnya:

```
>>> l = [1, 2, 3, 4]
>>> l.insert(2, 0)
>>> l
[1, 2, 0, 3, 4]
>>> l = l[:4] + [100] + l[4:]
>>> l
[1, 2, 0, 3, 100, 4]
```

Kita juga dapat membuat *list* kosong dan mengisinya dengan elemen – elemen yang kita inginkan, seperti contoh berikut ini:

```
>>> l = []
>>> l.append(4)
>>> l
[4]
>>> l.extend([5,6])
>>> l
[4, 5, 6]
>>> l.append([5,6])
>>> l
[4, 5, 6, [5, 6]]
```

Perbedaan antara kedua metode ini adalah, jika **`append`** akan menambahkan argumen sebagai elemen baru dalam *list*, sedangkan **`extend`** hanya akan menambahkan argumen tersebut sebagai elemen terakhir dalam *list.*

Elemen dalam *list* dapat kita hitung menggunakan metode **`count`**.

```
>>> [1, 2, 6, 2, 3, 1, 1].count(1)
3
```

Kita juga dapat menemukan indeks elemen yang pertama kali muncul dalam *list*. Jika elemen tersebut tidak terdapat dalam *list*, maka akan keluar *output error*.

```
>>> l = [1, 5, 2, 7, 2]
>>> l.index(2)
2
>>> l.index(8)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

Jika suatu *list* hanya memuat data numerik, maka kita dapat menjumlahkannya dengan menggunakan fungsi **`sum`**.

```
>>> sum([0, 1, 2, 3])
6
```

Elemen – elemen dalam *list* juga dapat kita urutkan menggunakan fungsi **`sorted`** dan metode **`sort()`**.

```
>>> l = [4, 2, 7, 4, 9, 1]
>>> sorted(l)
[1, 2, 4, 4, 7, 9]
>>> l

[4, 2, 7, 4, 9, 1]
>>> l.sort()
>>> l
[1, 2, 4, 4, 7, 9]
```

Terdapat perbedaan antara penggunaan fungsi dan metode ini. Jika pada fungsi **`sorted`**, Python akan membentuk *list* baru sebagai *output* dan tidak mengubah *list* lamanya, pada metode **`sort()`**., Python akan mengubah *list* yang belum tersusun menjadi *list* yang tersusun.

Di samping fungsi **`sorted`**, khusus untuk pengurutan dua buah bilangan Python mempunyai fungsi **`cmp(x,y)`**, di mana berlaku *output* -1 jika x < y, 0 jika x = y, 1 jika x > y. Berikut contohnya:

```
>>> cmp(1,2)
-1

>>> cmp(1,1)
0

>>> cmp(2,1)
-1
```

Fungsi **`sorted`** juga berlaku untuk *list* yang memuat elemen – elemen *string*. Perhatikan contoh berikut ini:

```
>>> sorted(['menwa', 'psik', 'skygers', 'hmme'])
['hmme', 'menwa', 'psik', 'skygers']
```

Jika elemen – elemen dalam *list* terdiri dari data numerik dan *string* seperti contoh di bawah ini, maka elemen diurutkan sesuai karakter yang pertama muncul dengan fungsi **`sorted`**.

```
>>> l = [[5, 'menwa'], [3, 'psik'], [7, 'skygers'], [3, 'hmme']]
>>> sorted(l)
[[3, 'hmme'], [3, 'psik'], [5, 'menwa'], [7, 'skygers']]
```

Kita juga dapat mengurutkan *list* secara terbalik dengan menggunakan metode **`sort()`**.

```
>>> l = [1, 2, 3]
>>> l.reverse()
>>> l
[3, 2, 1]
```

Tipe *Tuple*

Tuple pada dasarnya mirip dengan *list*, namun bersifat tetap. Jika kalian akrab dengan bahasa pemrograman seperti C dan C++, *tuple* mirip dengan *array* konstan Ketika kita sudah membuatnya, kita tidak dapat melakukan perubahan pada elemen – elemen di dalamnya. *Tuple* ditulis dengan menggunakan notasi “()”. Berikut adalah contoh *tuple*:

```
>>> t = (1, 2, 3)
>>> t[1] = 0
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Sama seperti *list*, *tuple* juga dapat memuat elemen – elemen dengan berbagai macam tipe yang dapat meliputi *list* dan *tuple* itu sendiri.

```
>>> t = (0., 1, 'dua', [3, 4], (5,6) )
```

Keuntungan kita menggunakan *tuple* adalah pemrosesannya yang jauh lebih cepat dibandingkan menggunakan *list*. Para *programmer* Python umumnya menggunakan *tuple* untuk efisiensi proses pemindahan data dan/atau argumen fungsi yang bersifat tetap. Ketika kita menuliskan daftar beberapa elemen yang dipisahkan oleh koma, tapi tidak diberi kurung, maka Python akan membacanya sebagai *tuple*. Contohnya sebagai berikut:

```
>>> 1, 2, 3
(1, 2, 3)
>>> "hmme", 7., [1, 2, 3]
('hmme', 7.0, [1, 2, 3])
```

Tuple bukan merupakan satu – satunya objek yang tidak dapat berubah dalam Python. *String* juga tidak dapat kita ubah dalam Python, perhatikanlah contoh berikut ini:

```
>>> s = "Mr Death memiliki 7 buah motor."
>>> s[17] = "6"
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

Untuk mengubah nilai *string* tersebut, kita membutuhkan metode pemotongan secara manual seperti berikut ini:

```
>>> s = s[:18] + "6" + s[19:]
>>> s
'Mr Death memiliki 6 buah motor.'
```

Tipe data *float*, *integer*, dan bilangan kompleks juga bersifat tidak dapat diubah, namun hal ini tidak begitu mengganggu para *programmer* Python. Pada tipe – tipe tersebut yang dimaksud sebagai data yang tidak dapat diubah berarti nilai numerik yang baru selalu menempati lokasi

memori yang baru pula, alih – alih memodifikasi alokasi memori yang telah digunakan untuk menyimpan nilai terdahulu.

Penugasan Variabel dan *Name Binding*

Python memperlakukan penugasan variabel secara berbeda dibandingkan bahasa pemrograman lainnya, di mana variabel harus dideklarasikan sebelumnya agar tersedia alokasi memori tertentu yang memungkinkan kita untuk melakukan manipulasi. Perhatikanlah penugasan variabel berikut ini:

```
>>> a = 1
```

Dalam bahasa pemrograman lainnya, perintah ini mungkin dibaca sebagai,"Menempatkan nilai 1 pada lokasi memori yang berkorespondensi terhadap variabel a." Namun, dalam Python perintah ini mungkin bermakna sedikit berbeda, yaitu "Menyediakan sebuah tempat pada memori untuk memuat sebuah variabel *integer* yang diberi nilai 1, kemudian mengarahkan variabel a ke situ." Proses ini dikenal sebagai *name binding* dalam pemrograman Python. Maka variabel dalam Python lebih nampak sebagai 'peta' yang mengarahkan pada lokasi memori tertentu, bukannya variabel itu sendiri yang menempati lokasi memori tertentu.

Perhatikan contoh berikut ini:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a[1] = 0
>>> a
[1, 0, 3]
>>> b
[1, 0, 3]
```

Pada baris kedua, Python mengikat variabel b ke lokasi memori yang sama dengan variabel a. Patut kalian ketahui, bahwa proses ini bukanlah penyalinan variabel a ke b, maka dari itu modifikasi pada a juga mempengaruhi variabel b. Terkadang hal ini justru merupakan keuntungan bagi para *programmer* karena mempermudah dan mempercepat eksekusi suatu program.

Jika kalian ingin melakukan penyalinan objek secara eksplisit, Python menyediakan modul **`copy`**. Berikut contoh penggunaannya:

```
>>> import copy
>>> a = [1, 2, 3]
>>> b = copy.copy(a)
>>> a[1] = 0
>>> a
[1, 0, 3]
>>> b
[1, 2, 3]
```

Di sini, fungsi **`copy.copy`** bertugas untuk membuat lokasi baru pada memori dan menyalin isi variabel a pada lokasi tersebut, kemudian variabel b diarahkan ke sana. Oleh karena a dan b berada pada lokasi memori yang terpisah, perubahan pada salah satunya tidak berdampak pada yang lain.

Sebenarnya fungsi **`copy.copy`** hanya menyalin bagian terluar dari struktur dalam *list* tersebut. Jika suatu *list* memuat *list* lain, atau objek – objek dengan tingkat variabel yang lebih dalam, maka kita harus menggunakan fungsi **`copy.deepcopy`** untuk melakukan penyalinan secara lengkap. Berikut contohnya:

```
>>> import copy
>>> a = [1, 2, [3, 4]]
>>> b = copy.copy(a)
>>> c = copy.deepcopy(a)
>>> a[2][1] = 5
>>> a
[1, 2, [3, 5]]
>>> b
[1, 2, [3, 5]]
>>> c
[1, 2, [3, 4]]
```

Kita disarankan untuk tidak terlalu sering menggunakan modul **`copy`** (oleh karena itulah **`copy`** ditempatkan dalam modul bukan pada perintah standar dalam Python). Sebagian besar program dalam Python tidak membutuhkan fungsi dalam modul ini. Namun, hal tersebut dapat dilakukan jika kalian terbiasa menulis program dalam gaya yang sesuai dengan *programmer* Python. Jika kalian terlalu sering menggunakan modul **`copy`**, maka program yang kalian tulis akan tidak efisien.

Contoh berikut ini mungkin dapat membingungkan kalian:

```
>>> a = 1
>>> b = a
>>> a = 2
>>> a
2
>>> b
1
```

Mengapa `b` tidak berubah? Alasannya tentu saja berhubungan dengan dengan objek – objek yang bersifat tetap. Pemanggilan ulang suatu nilai bersifat tetap (*immutable*), yang berarti tidak dapat berubah ketika sudah terekam dalam memori. Pada baris kedua, `b` diarahkan pada suatu lokasi dalam memori, di mana terdapat nilai 1 yang telah dibuat pada baris pertama. Pada baris ketiga, dibuat nilai baru, yaitu 2 di dalam memori, dan variabel `a` diarahkan pada nilai tersebut, namun nilai 1 tetap terekam dalam memori (tidak berubah). Hasilnya adalah variabel `a` dan `b` terarah pada bagian yang berbeda dalam memori. Kemudian yang menjadi pertanyaan adalah mengapa pada contoh *list* nilai `b` dapat berubah? Hal ini dikarenakan oleh sifat dari tipe *list* yang dapat berubah dalam memori (*mutable*). Berikut ini contoh lainnya yang mirip:

```
>>> a = 1
>>> b = a
>>> a = []
>>> a.append(1)
>>> a
[1]
>>> b
1
```

Pada baris ketiga contoh tersebut, variabel a mengarah pada suatu titik *list* kosong yang telah dibuat dalam memori.

Berikut ini merupakan aturan umum penugasan variabel dalam Python yang wajib kalian pahami:

- Penugasan dalam Python menggunakan tanda "=" yang berarti menunjuk suatu variabel pada ruas kiri ke suatu lokasi memori pada ruas kanan.
- Jika ruas kanan berupa variabel, maka arahkan ruas kiri pada lokasi yang sama dengan memori yang ditunjukkan oleh ruas kanan. Jika ruas kanan merupakan suatu objek atau nilai baru, buatlah suatu tempat baru dalam memori untuk memuat nilai atau objek tersebut dan arahkanlah pada variabel di sisi kirinya.
- Perubahan pada suatu objek yang dapat berubah (*mutable*) akan berdampak pada lokasi memori tempat penyimpanannya.dan juga pada setiap variabel yang mengarah ke sana. Objek yang tidak dapat berubah (*immutable*) tidak dapat dimodifikasi dan perubahan nilai akan menempati lokasi baru dalam memori.

Python memungkinkan kita untuk melakukan pemeriksaan apakah dua variabel yang berbeda mengarah pada suatu objek atau nilai yang sama dalam memori dengan menggunakan perintah **`is`** seperti pada contoh berikut ini:

```
>>> a = [1, 2, 3]
>>> b = a
>>> a is b
True
>>> b = [1, 2, 3]
>>> a is b
False
```

Pada baris ke lima, dibuat lokasi baru dalam memori yang memuat *list* dengan isi yang sama dengan *list* sebelumnya, namun memiliki alamat memori yang berbeda. Variabel **`b`** kemudian diarahkan pada lokasi penyimpanan *list* yang baru di memori, oleh karena itu ketika dilakukan komparasi antara **`a`** dan **`b`** hasilnya adalah **`False`**, meskipun datanya identik.

Kalian tentu menduga bahwa pemrograman dengan menggunakan Python tidaklah efisien karena Python harus membuat lokasi baru dalam memori untuk setiap objek dan/atau nilai baru. Namun, untungnya Python merupakan bahas pemrograman unggul yang mampu mengelola memori secara transparan dan cerdas. Python memiliki teknik pengelolaan memori yang dikenal sebagai *garbage collection*. Melalui teknik ini, Python melacak berapa banyak variabel yang mengarah pada setiap lokasi memori yang menyimpan objek dan/atau nilai tertentu. Ketika sudah tidak ada lagi variabel yang mengarah pada objek dan/atau nilai pada suatu lokasi memori, maka Python akan menghapus objek dan/atau nilai dari lokasi tersebut, sehingga lokasi memori itu dapat digunakan untuk penggunaan berikutnya. Perhatikanlah contoh berikut ini:

```
>>> a = [1, 2, 3, 4] #a mengarah ke list 1
>>> b = [2, 3, 4, 5] #b mengarah ke list 2
>>> c = a #c mengarah ke list 1
>>> a = b #a mengarah ke list 2
>>> c = b[1] #c mengarah ke angka '3'; list 1 dihapus dari memori
```

Pada baris terakhir, tidak lagi terdapat variabel yang mengarah ke lokasi *list* pertama, karena itu Python secara otomatis menghapusnya dari memori. Mungkin banyak dari kalian yang berpikir, bahwa kita dapat melakukan penghapusan secara eksplisit menggunakan perintah **`del`** seperti contoh berikut ini:

```
>>> a = [1, 2, 3, 4]
>>> del a
```

Namun, perintah ini hanya akan menghapus variabel a, sedangkan lokasi memori tertentu masih memuat objek dari variabel tersebut, kecuali jika variabel a merupakan satu – satunya variabel yang mengarah pada objek tersebut. Untuk lebih jelasnya perhatikan contoh berikut ini:

```
>>> a = [1, 2, 3, 4]
>>> b = a
>>> del a
>>> b
[1, 2, 3, 4]
```

Penugasan Berganda

Memungkinkan bagi kita untuk melakukan penugasan banyak variabel pada *list* dan *tuple* dalam Python pada waktu yang sama. Berikut contoh penugasan banyak variabel dalam *list*:

```
>>> [a, b, c] = [1, 2, 3]
>>> a
1
>>> b
2
>>> c
3
```

Pada contoh tersebut, Python melakukan penugasan variabel dengan menyusun elemen – elemennya ke dalam *list* pada ruas kanan. Sehingga, *list* yang hendak disusun tersebut harus sama panjang, jika tidak maka terjadi *error*.

Dalam kasus penugasan berganda, *tuple* biasanya lebih sering digunakan dibandingkan *list* karena jauh lebih efisien. Berikut contohnya:

```
>>> (a, b, c) = (3, "herho", [1, 2])
>>> a
3
>>> b
'herho'
>>> c
[1, 2]
```

Berikut ini adalah contoh bagaimana *tuple* dapat digunakan sebagai fungsi yang memuat dua buah nilai:

```
>>> a, b = f(c)

>>> returned = f(c)
>>> a, b = returned
```

List comprehension juga dapat digunakan untuk melakukan penugasan banyak variabel seperti contoh berikut ini:

```
>>> l = [(1,2), (3,4), (5,6)]
>>> [a+b for (a,b) in l]
[3, 7, 11]
```

Dalam contoh tadi, variabel *tuple* (a,b) ditugaskan untuk setiap elemen dalam *list* l. Karena l memuat *tuple*, nilainya merupakan lokasi dalam memori tempat ditugaskannya variabel a dan b. Kita juga dapat melakukan hal yang sama dengan perintah sebagai berikut:

```
>> [t[0] + t[1] for t in l]
```

Dalam hal ini, variabel t ditugaskan untuk setiap *tuple* dalam *list* l, kita dapat mengaksesnya dengan menggunakan tanda pengindeksan "[]". Alternatif lainnya adalah kita dapat menggunakan fungsi **sum** untuk menyelesaikan permasalahan yang sama:

```
>>> [sum(t) for t in l]
```

Umumnya penugasan berganda ini digunakan untuk menukar nilai antar variabel, seperti contoh berikut ini:

```
>>> a = 1
>>> b = 3
>>> a, b = b, a
>>> a
3
>>> b
1
```

Fungsi dan Manipulasi Dalam *String*

Fungsi pemrosesan *string* merupakan salah satu keunggulan lain dari bahasa pemrograman Python karena sangat handal dan mudah digunakan untuk memproses data teks, khususnya ketika digabungkan dengan kegunaan *list*. Setiap data *string* dalam Python (sama juga seperti tipe data

lainnya) merupakan suatu objek. Fungsi – fungsi *string* merupakan bagian dari objek – objek ini yang dapat kita akses menggunakan notasi titik.

Berikut ini dua hal yang harus kalian perhatikan ketika menggunakan fungsi – fungsi *string*:

- *String* bersifat tidak dapat diubah (*immutable*), maka suatu fungsi yang memodifikasi data *string*, sesungguhnya membuat data *string* baru hasil modifikasi dari data *string* sebelumnya.
- Seluruh fungsi *string* bersifat *case sensitive*.

String dapat kita potong seperti layaknya *list*. Hal ini memudahkan kita untuk melakukan ekstraksi terhadap objek *substring*. Berikut contohnya:

```
>>> s = "Icha Vinaldi"
>>> s[:4]
'Icha'
>>> "Icha Vinaldi"[-7:]
'Vinaldi'
```

String dapat kita pecah ke dalam *list*. Fungsi **`split`** akan secara otomatis memecah *string* menjadi elemen – elemen dalam *list* begitu terdapat celah antar teks tersebut (baik spasi maupun baris baru). Berikut contohnya:

```
>>> "Halo\nGita Hardaningtyas".split()
['Halo', 'Gita', 'Hardaningtyas’]
```

Kita juga dapat menggunakan fungsi **`split`** untuk memecah *string* ke dalam *list* pada bagian *substring* tertentu:

```
>>> "Gita Hardaningtyas".split('a')

['Git', ' H', 'rd', 'ningty', 's']
```

Fungsi yang merupakan kebalikan dari **`split`** adalah **`join`** yang digunakan untuk menggabungkan elemen – elemen *list* yang berupa *string* menjadi *string* utuh. Berikut contoh – contohnya:

```
>>> l = ["Riggita", "Putri", "Damayanti"]
>>> " ".join(l)
'Riggita Putri Damayanti'
>>> ", ".join(["Parako", "Sandha", "Gultor"])
'Parako, Sandha, Gultor'
>>> "".join(["Para","komando"])
'Parakomando'
```

Kita dapat menggunakan fungsi **strip** untuk membuang celah teks di awal dan akhir suatu data *string*. Berikut contohnya:

```
>>> "  Iggi   ".strip()
'Iggi'
>>> "Iggi\n\n".strip()
'Iggi'
```

Fungsi **replace** dapat kita gunakan untuk membentuk *string* baru yang mana suatu komponen substring baru telah mengganti komponen *substring* dalam *string* sebelumnya. Berikut ini contohnya:

```
>>> "Pak Ridho menguasai Matlab".replace("Matlab","Python")
'Pak Ridho menguasai Python'
```

Kita juga dapat memeriksa apakah suatu *substring* terdapat dalam suatu *string* dengan menggunakan fungsi **in** dan memeriksa indeks *substring* tertentu yang pertama kali dituliskan dalam *string* tersebut. Untuk lebih jelasnya, perhatikanlah contoh berikut ini:

```
>>> s = "M.Ridho Syahputra,M.Si."
>>> "M.Si." in s
True
>>> s.index("a")
10
>>> s.index("M.T.")
Traceback (most recent call last):
  File "<pyshell#15>", line 1, in <module>
    s.index("M.T.")
ValueError: substring not found
```

Terkadang kita membutuhkan teks yang rata kiri dan/atau rata kanan dalam *string*. Berikut cara yang umumnya digunakan oleh para *programmer* Python:

```
>>> s = "apple".ljust(10) + "orange".rjust(10) + "\n" \
... + "grape".ljust(10) + "pear".rjust(10)
>>> print s
apple  orange
grape    pear
```

Terdapat beberapa fungsi yang dapat digunakan untuk memanipulasi penggunaan huruf besar dalam Python. Kalian dapat melihatnya melalui contoh berikut ini:

```
>>> s = "Buku ini ditulis oleh Wilks."
>>> s.lower()
'buku ini ditulis oleh wilks.'
>>> s.upper()
'BUKU INI DITULIS OLEH WILKS.'
>>> s.capitalize()
'Buku ini ditulis oleh wilks.'
>>> s.title()
'Buku Ini Ditulis Oleh Wilks.'
```

Beberapa fungsi berikut ini dapat digunakan untuk memeriksa akhiran dan awalan suatu *string*:

```
>>> s = "Annisa Indahvinaldi"
>>> s.startswith("An")
True
>>> s.startswith("a")
False
>>> s.endswith("i")
True
```

Kita juga dapat memeriksa jenis elemen suatu data *string*. Misalnya, ketika kita ingin memastikan bahwa elemen dari data *string* kita tersusun seluruhnya dari alfabet:

```
>>> "Icha".isalpha()
True
>>> "Icha!".isalpha()
False
```

Hal yang sama juga dapat kita lakukan untuk memeriksa elemen numerik dalam *string*:

```
>>> "12014081".isdigit()
True
>>> "65 juta tahun yang lalu.".isdigit()
False
```

Tipe *Dictionary*

Dictionary adalah salah satu tipe kumpulan objek yang mirip dengan *list* dan *tuple*. Namun, perbedaannya adalah *dictionary* memuat objek yang tidak tersusun. Sebagai gantinya, *dictionary* menghubungkan relasi satu ke satu (pemetaan) antara kunci (*key*) ke suatu nilai. Dalam bahasa pemrograman lainnya dikenal sebagai *array* asosiatif yang indeksnya berupa nama tertentu. Bentuk umum *dictionary* adalah sebagai berikut:

```
variabel = {key1 : nilai1, key2 : nilai2, ...}
```

Berikut ini adalah contoh *dictionary*:

```
>>> d = {"Nama" : "Annisa Indahvinaldi", "Prodi" : "Teknik Geologi",
... "NIM" : "12014081"}
```

Untuk mengakses nilai dari *key* tertentu maka kita dapat melakukan:

```
>>> d["Nama"]
'Annisa Indahvinaldi'
>>> d["NIM"]
'12014081'
>>> d["Alamat"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Alamat'
```

Perhatikanlah bahwa pemanggilan nilai *key* yang tidak terdapat dalam *dictionary* akan menghasilkan nilai *error*.

Key pada *dictionary* tidak harus berupa *string*. *Key* dapat berupa seluruh objek yang tidak dapat berubah (*immutable*), yaitu *integer*, *tuple*, dan/atau *string*. Sementara untuk nilai sendiri, kita bebas untuk menempatkan objek tipe apa saja, meliputi numerik; *list*; modul; fungsi; dll. Berikut contohnya:

```
>>> d = {"satu" : 80.0, 2 : [0, 1, 1], 3 : (-20,-30), (4, 5) : 60}
>>> d[(4,5)]
60
>>> d[2]
[0, 1, 1]
```

Kita juga dapat membuat suatu *dictionary* kosong:

```
>>> d = {}
```

Kita juga dapat mengubah dan menambahkan elemen pada *dictionary*. Berikut ini contohnya:

```
>>> d = {"Prodi":"Meteorologi", "Kampus":"ITB"}
>>> d["Prodi"] = "Earth Sciences"
>>> d["Kampus"] = "University of Cambridge"
>>> d["Alamat"] = "Downing Street, Cambridge, Cambridgeshire\
... , CB2 3EQ, UK"
>>> d
{'Prodi': 'Earth Sciences', 'Alamat': 'Downing Street, Cambridge, Cambridgeshire
, CB2 3EQ, UK', 'Kampus': 'University of Cambridge'}
```

Untuk menghapus elemen dari *dictionary* kita dapat menuliskan perintah **`del`**.

```
>>> del d["Alamat"]
>>> d
{'Prodi': 'Earth Sciences', 'Kampus': 'University of Cambridge'}
```

Terdapat dua cara untuk memeriksa apakah *key* tertentu berada dalam *dictionary* yang dapat kalian lihat berikut ini:

```
>>> d = {"Prodi":"Meteorologi", "Kampus":"ITB"}
>>> "Prodi" in d
True
>>> d.has_key("Alamat")
False
```

Ukuran suatu *dictionary* dapat kita ukur menggunakan fungsi **`len`**.

```
>>> len(d)
2
```

Untuk menghapus seluruh elemen dalam *dictionary*, kita dapat menggunakan fungsi **`clear`**.

```
>>> d = {"Prodi":"Meteorologi", "Kampus":"ITB"}
>>> d.clear()
>>> d
{}
```

Kita juga dapat mengubah *key* dan nilai dari *dictionary* menjadi *list*. Berikut ini contohnya:

```
>>> d = {"Prodi":"Meteorologi", "Kampus":"ITB"}
>>> d.keys()
['Prodi', 'Kampus']
>>> d.values()
['Meteorologi', 'ITB']
```

Di samping itu, kita juga dapat menjadikan *dictionary* menjadi elemen *tuple* dalam *list* dengan format ”(*key*, nilai)”, seperti berikut ini:

```
>>> d.items()
[('Prodi', 'Meteorologi'), ('Kampus', 'ITB')]
```

Sebaliknya, kita juga dapat membuat sebuah *dictionary* dari *list* yang memuat dua buah elemen *tuple*. Perhatikan contoh berikut ini:

```
>>> l = [("Alamat", "Jalan Ganesha 10"), ("Kampus", "ITB")]
>>> dict(l)
{'Alamat': 'Jalan Ganesha 10', 'Kampus': 'ITB'}
```

Dictionary juga mengenal fungsi **`get`** yang digunakan untuk memperoleh nilai dalam *dictionary* tersebut berdasarkan *key* yang dilewati. Jika *key* yang dicari tidak ada, maka fungsi **`get`** akan mengembalikan nilai *default* (**`None`**). Berikut ini contohnya:

```
>>> d = {"Prodi":"Meteorologi", "Kampus":"ITB"}
>>> d.get("Prodi", "Teknik Geologi")
'Meteorologi'
>>> d.get("NIM", 12014081)
12014081
```

Perintah If

Perintah **`if`** digunakan untuk menangani permasalahan bersyarat. Berikut ini contohnya:

```
>>> x = 12014081
>>> if x > 12014081:
...   print "NIM di atas Icha."
... elif x <12014081:
...   print "NIM di bawah Icha."
... else:
...   print "Halo, Icha!"
...
Halo, Icha!
```

Perhatikanlah bahwa baris pertama dalam rangkaian ini diawali dengan perintah **`if`** yang berarti jika, baris kedua dengan perintah el**`if`** yang berarti selain "if" (*else if*), rangkaian diakhiri dengan perintah **`else`** yang berarti selain keduanya. Masing – masing perintah ini diakhiri dengan tanda

titik dua yang kemudian diikuti subperintah yang hendak dieksekusi. Pada rangkaian perintah **`if`**, kedua perintah el**`if`** dan **`else`** bersifat opsional.

Satu lagi konsep penting dalam pemrograman Python yang harus kalian ingat adalah bahwa spasi dan indentasi mempunyai makna sintaktis. Spasi dan indentasi mendikte bagaimana kita mengeksekusi perintah. Tanda titik dua diperlukan untuk menandai subperintah sesudah perintah **`if`**, pada pengulangan (*looping*), atau sesudah pendefinisan fungsi. Dalam Python, seluruh subperintah yang satu kelompok sesudah tanda titik dua harus memilki indentasi yang sama, jika tidak maka akan menghasilkan *error*. Untuk lebih jelasnya perhatikan contoh berikut ini:

```
>>> if 1 < 3:
...     print "baris satu"
...             print "baris dua"
  File "<stdin>", line 3
    print "baris dua"
    ^
IndentationError: unexpected indent
```

Coba kalian perhatikan perbedaan contoh tadi dengan contoh berikut ini:

```
>>> if 1 < 3:
...     print "baris satu"
...     print "baris dua"
...
baris satu
baris dua
```

Para *programmer* Python umumnya menggunakan empat spasi untuk memulai penulisan subperintah sesudah tanda titik dua. Meskipun terkadang merepotkan bagi para *programmer* pemula, indentasi ini membantu kita untuk dapat membaca setiap maksud perintah dalam program menjadi lebih jelas.

Kita dapat menggunakan perintah **`if`** untuk mengeksekusi seluruh perintah atau fungsi yang menghasilkan nilai *boolean*: **`True`** atau **`False`**. Angka 0 juga dapat diinterpretasikan Python sebagai **`False`**, sedangkan angka bukan nol dapat diinterpretasikan sebagai **`True`**. *List* dan objek kosong

lainnya dapat dibaca sebagai **`False`**, sedangkan yang tidak kosong dibaca sebagai **`True`**. Berikut ini contohnya:

```
>>> d = {}
>>> if d:
...     print "Dictionary tidak kosong"
... else:
...     print "Dictionary kosong"
...
Dictionary kosong
```

Perintah **`if`** tunggal (tanpa rangkaian el**`if`** dan **`else`**) dapat kita eksekusi dalam satu baris perintah, sehingga tidak diperlukan indentasi. Berikut contohnya:

```
>>> if 1 < 3: print "satu lebih kecil dari tiga"
...
satu lebih kecil dari tiga
```

Kita juga dapat membuat rangkaian perintah **`if`** bersarang dengan memanfaatkan indentasi. Berikut ini contohnya:

```
>>> s = "Fisika Atmosfer"
>>> if "Rekayasa" in s:
...     print "Tidak ada matakuliah rekayasa di sini."
... elif "Fisika" in s:
...     if "Kuantum" in s:
...             print "Kami sudah tidak mengajarkan Fisika Kuantum."
...     elif "Atmosfer" in s:
...             print "Fisika Atmosfer merupakan matakuliah favorit di sini."
...
Fisika Atmosfer merupakan matakuliah favorit di sini.
```

Pengulangan Menggunakan For

Seperti juga bahasa pemrograman lainnya, Python juga menyediakan mekanisme pengulangan untuk menghasilkan nilai secara beturut – turut. Namun, tidak seperti bahasa pemrograman lainnya, *looping* dalam Python tidak melakukan iterasi pada *integer*, melainkan memperlakukannya seperti elemen dalam sikuen seperti layaknya *list* dan *tuple*. Bentuk umum *looping* dengan perintah **`for`** dalam Python adalah sebagai berikut:

```
>>> for elemen in sikuen
...     <perintah>
```

Perhatikanlah pada rangkaian pengulangan menggunakan **`for`**, juga berlaku indentasi yang sama seperti yang kita terapkan pada rangkaian perintah **`if`**. Berikut contoh iterasi dalam *list* dan *tuple*:

```
>>> for i in [1,2,8, "Atmosphaira"]:
...     print i
...
1
2
8
Atmosphaira
>>> for i in (120.14, [0,8,1], {"Nama" : "Annisa Indahvinaldi"}):
...     print i
...
120.14
[0, 8, 1]
{'Nama': 'Annisa Indahvinaldi'}
```

Perhatikanlah bahwa elemen yang hendak diiterasikan tidak harus bertipe sama. Dalam proses pengulangan, variabel i digunakan untuk memproses nilai dari elemen – elemen dalam *list* atau *tuple* untuk diproses dalam iterasi. Kita tidak wajib menuliskan variabel pengulangan ini sebagai **`i`**, tetapi jika kita menggunakan variabel yang telah digunakan sebelumnya, maka nilainya akan digantikan oleh proses *looping* ini.

Kita juga dapat mengerjakan proses pengulangan dengan melakukan pemotongan pada elemen *list* seperti pada contoh berikut ini:

```
>>> l=[1,2,3,4,5,6,7,8,9]
>>> for i in l[4:]:
...     print i
...
5
6
7
8
9
```

Iterasi pada *dictionary* bekerja pada *key* alih – alih pada nilai. Ingatlah, bahwa pada *dictionary* tidak dikenal urutan. Secara umum para *programmer* Python umumnya melakukan iterasi eksplisit pada *key* dan nilai pada *dictionary* sebagaimana contoh berikut ini:

```
>>> d = {"Nama": "Annisa Indahvinaldi", "NIM": "12014081"}
>>> for val in d:
...     print val
...
Nama
NIM
>>> for val in d.keys():
...     print val
...
Nama
NIM
>>> for val in d.values():
...     print val
...
Annisa Indahvinaldi
12014081
```

Dengan menggunakan metode penugasan berganda dalam Python, memungkinkan bagi kita untuk melakukan iterasi untuk lebih dari satu nilai pada waktu yang sama. Berikut ini contohnya:

```
>>> l = [(1,2), (3,4), (5,6), (7,8), (9,10)]
>>> for (a,b) in l:
...     print a + b
...
3
7
11
15
19
```

Dalam contoh tersebut, Python melakukan iterasi terhadap elemen – elemen dalam *list* dengan melakukan penugasan terhadap variabel (a,b) ke setiap nilai dalam elemen *tuple* di *list* 1. Metode penugasan berganda juga memudahkan kita dalam melakukan *looping key* dan nilai secara bersamaan pada *dictionary*. Berikut contohnya:

```
>>> d = {"Nama": "Annisa Indahvinaldi", "NIM": "12014081"}
>>> d.items()
[('Nama', 'Annisa Indahvinaldi'), ('NIM', '12014081')]
>>> for (key,val) in d.items():
...     print "Key: %s dan nilainya adalah %s" % (key,val)
...
Key: Nama dan nilainya adalah Annisa Indahvinaldi
Key: NIM dan nilainya adalah 12014081
```

Memungkinkan juga bagi kita untuk melakukan iterasi pada sikuen bilangan menggunakan fungsi **range** seperti contoh berikut ini:

```
>>> for i in range(8):
...     print i
...
0
1
2
3
4
5
6
7
```

Dalam bahasa pemrograman lainnya, seseorang mungkin akan melakukan gaya komputasi seperti berikut ini untuk melakukan iterasi dalam suatu *list*:

```
>>> l = [1,2,3]
>>> for i in range(len(l)):
...     print l[i]
...
1
2
3
```

Namun dalam Python, para *programmer* umumnya memakai cara berikut ini yang jauh lebih efisien dan sesuai dengan pemrograman yang berorientasi objek:

```
>>> l = [1,2,3]
>>> for i in l:
...     print i
...
1
2
3
```

Perintah pada baris kedua dan baris ketiga sesungguhnya dapat dijadikan datu baris karena alasan yang telah kita bahas sebelumnya, namun bagi para *programmer* Python lebih umum dipisah menjadi dua baris agar tidak membingungkan.

Kita dapat menggunakan perintah **`enumerate`** jika ingin mendapatkan keterangan indeks dalam proses iterasi terhadap suatu *list*. Berikut contohnya:

```
>>> l = [1,2,3]
>>> for (ind, val) in enumerate(l):
...     print "Elemen ke %i dalam list adalah %d" %(ind,val)
...
Elemen ke 0 dalam list adalah 1
Elemen ke 1 dalam list adalah 2
Elemen ke 2 dalam list adalah 3
```

Perhatikanlah jika indeks yang dihasilkan oleh fungsi **`enumerate`** selalu dimulai dari angka 0, meskipun kita melakukan pemotongan *list* terlebih dahulu sebelum diiterasikan. Berikut ini contohnya:

```
>>> l = [1,2,3,4,5,6,7,8,9]
>>> for (ind, val) in enumerate(l[3:]):
...     print "Elemen ke %i dalam list adalah %d" %(ind,val)
...
Elemen ke 0 dalam list adalah 4
Elemen ke 1 dalam list adalah 5
Elemen ke 2 dalam list adalah 6
Elemen ke 3 dalam list adalah 7
Elemen ke 4 dalam list adalah 8
Elemen ke 5 dalam list adalah 9
```

Python memungkinkan kita untuk melakukan iterasi dua *list* secara bersamaan menggunakan fungsi **`zip`** seperti contoh berikut ini:

```
>>> l1 = [1,2,3]
>>> l2 = [4,5,6]
>>> for (a,b) in zip (l1,l2):
...     print a, b, a + b
...
1 4 5
2 5 7
3 6 9
```

Fungsi **zip** sendiri dapat digunakan di luar proses pengulangan **for**. Cara bekerjanya sesederhana mengelompokan lebih dari satu *list*, serta membuat *tuple* dari elemen – elemen *list* yang telah dikelompokkan. Berikut ini contoh penggunaan fungsi **zip**:

```
>>> zip ([1,2,3], [4,5,6])
[(1, 4), (2, 5), (3, 6)]
>>> zip ([1,2,3], [4,5,6], [7,8,9])
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
```

Sebagaimana pada perintah **if**, kita juga dapat melakukan pengulangan bersarang dengan menggunakan fungsi **for**:

```
>>> for i in range (5):
...     for j in range (1,i):
...             print(i,j)
...
(2, 1)
(3, 1)
(3, 2)
(4, 1)
(4, 2)
(4, 3)
```

Memungkinkan bagi kita untuk melanjutkan proses iterasi kita secara langsung tanpa mengeksekusi perintah pada blok indentasi yang sama dengan menggunakan perintah **continue**. Berikut ini contoh penggunaan fungsi **continue** untuk menghasilkan hasil yang sama dengan contoh

sebelumnya (meskipun begitu, metode ini dianggap tidak efisien karena melakukan banyak proses iterasi):

```
>>> for i in range (5):
...     for j in range(5):
...             if i <= j: continue
...             print (i,j)
...
(1, 0)
(2, 0)
(2, 1)
(3, 0)
(3, 1)
(3, 2)
(4, 0)
(4, 1)
(4, 2)
(4, 3)
```

Kita juga dapat menghentikan proses iterasi yang paling dalam (*innermost loop*) dengan menggunakan perintah **`break`**. Sekali lagi, contoh berikut ini akan menghasilkan hasil yang sama, namun metode kerjanya seefisien contoh yang pertama karena iterasi paling dalam akan terhenti seketika pada waktu kemunculan perintah **`break`**. Berikut contohnya:

```
>>> for i in range (5):
...     for j in range(5):
...             if i <= j: break
...             print (i,j)
...
(1, 0)
(2, 0)
(2, 1)
(3, 0)
(3, 1)
(3, 2)
(4, 0)
(4, 1)
(4, 2)
(4, 3)
```

Pengulangan Menggunakan While

Pengulangan menggunakan perintah **`while`** berbeda dengan pengulangan menggunakan perintah **`for`**. Pengulangan menggunakan perintah **`while`** tidak melakukan iterasi elemen – elemen di sepanjang deret data, namun dapat terus melanjutkan iterasi asalkan beberapa kondisi uji terpenuhi. Pemakaian sintaks pada pengulangan **`while`** tetap mengikuti aturan indentasi seperti kasus – kasus yang telah kita jumpai sebelumnya. Berikut ini bentuk umum yang digunakan:

```
>>> while kondisi:
```

Berikut ini diberikan contoh penerapan iterasi menggunakan perintah **`while`** untuk menghitung deret Fibonacci:

```
>>> i1,i2 = 1,1
>>> while i1 < 30:
...     i1,i2 = i2, i1 + i2
...     print i1
...
1
2
3
5
8
13
21
34
```

Terkadang kita butuh memberhentikan iterasi pada suatu lokasi di tengah – tengah perintah yang mengikutinya. Untuk menyelesaikan permasalahan ini, Python mempunyai perintah **`break`** untuk menghentikan iterasi tak berhingga. Dalam contoh ini misalnya, kita ingin melihat bilangan Fibonacci dengan nilai di bawah atau sama dengan 30:

```
>>> i1,i2 = 1,1
>>> while True:
...     i1,i2 = i2, i1 + i2
...     if i1 > 30: break
...     print i1
...
1
2
3
5
8
13
21
```

Pada contoh tersebut, kita membuat iterasi tak berhingga dengan menggunakan perintah **`while True`**. Penting untuk kalian ingat jika kalian bekerja pada pengulangan bersarang, maka perintah **`break`** hanya menghentikan iterasi di bagian paling dalam.

Fungsi

Fungsi memainkan peranan penting dalam setiap program yang kita tuliskan dalam Python. Bahasa pemrograman lain umumnya membedakan antara fungsi (yang menghasilkan nilai) dengan *subroutine* yang tidak menghasilkan nilai, tetapi melakukan tugas yang mirip. Namun, Python hanya mengenal istilah fungsi, meskipun dapat menghasilkan nilai tunggal; ganda; maupun tidak menghasilkan nilai sama sekali. Seperti layaknya hal – hal lain, fungsi dalam Python merupakan objek. Dengan demikian berarti, fungsi dapat memuat *list*, *tuple*, *dictionary*, dan bahkan mampu mengirimnya ke fungsi lain. Hal inilah yang membuat Python merupakan bahasa pemrograman yang paling fleksibel.

Untuk membuat fungsi, kita dapat menggunakan perintah **`def`** seperti contoh fungsi penjumlahan berikut ini:

```
>>> def jum(arg1,arg2):
...     x = arg1 + arg2
...     return x
...
>>> jum(8,1)
9
```

Pada contoh tersebut perintah **`def`** bermakna pendefinisian fungsi yang diberi nama **`jum`**, yang mana membutuhkan dua buah variabel sebagi *input* (**`arg1`**, **`arg2`**). Seluruh subperintah yang dituliskan di bawah **`def`** harus mengikuti aturan indentasi. Perintah **`return`** memberitahukan Python untuk melakukan dua hal, yaitu untuk keluar dari fungsi yang telah kita buat, dan jika kita memberikan nilai pada fungsi tersebut maka **`return`** akan mengembalikan nilai ke fungsi tersebut.

Tidak seperti bahasa pemrograman lainnya, fungsi dalam Python tidak perlu mengetahui tipe variabel apa yang menjadi *input*annya. Python dapat membacanya sendiri setiap kali kita

memanggil fungsi tersebut. Dengan menggunakan fungsi penjumlahan yang telah kita buat sebelumnya, kita dapat menerapkan fungsi ini pada berbagai tipe data:

```
>>> jum("Iban","Aso")
'IbanAso'
>>> jum([1,2,3],[4,5,6])
[1, 2, 3, 4, 5, 6]
>>> jum(12014081,"Iban Aso")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in jum
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Pada penggunaan fungsi yang terakhir, terjadi *error* karena operator penjumlahan dalam Python tidak didesain untuk menjumlahkan *integer* dengan *string*. *Error* ini hanya dapat terjadi ketika kita memanggil fungsi dengan argumen yang tidak tepat.

Perintah `return` dapat dituliskan pada bagian mana saja dalam fungsi, berikut ini contohnya:

```
>>> def pangkat(x,y):
...     if x <= 0:
...             return o
...     else:
...             return x**y
...
>>> def pangkat(x,y):
...     if x <= 0:
...             return 0
...     else:
...             return x**y
...
>>> pangkat(8,1)
8
>>> pangkat(0,12)
0
```

Jika tidak terdapat perintah **`return`** pada suatu fungsi, atau jika perintah **`return`** telah digunakan tanpa memiliki nilai pengembaliannya (*return value*). Python akan mengeluarkan nilai khusus yaitu **`None`**. Berikut ini contohnya:

```
>>> def tes(x):
...     print"%11.4e" %x
...     return
...
>>> ret = tes(1)
 1.0000e+00
>>> ret == None
True
>>> ret is None
True
```

`None` merupakan objek khusus dalam Python yang mirip dengan **`True`** dan **`False`**. Secara harafiah berarti tidak ada nilai dan tidak akan tampak meskipun kita menjalankan perintah **`print`**. Meskipun begitu, kita dapat melakukan pengecekan terhadap nilai **`None`** dengan menggunakan kesamaan kondisional atau dengan menggunakan perintah **`is`**.

Jika seseorang ingin mengubah fungsi bergantung pada tipe argumen yang menjadi *input*an, kita dapat menguji tipe – tipe yang berbeda menggunakan fungsi **`type`**. Berikut ini contohnya:

```
>>> def jum(arg1, arg2):
... #tes untuk melihat apabila satu nilai adalah string dan yang lain bukan
...     if type(arg1) is str and not type(arg2) is str:
...             konverarg1 = type(arg2)(arg1)
...             return konverarg1 + arg2
...     elif not type(arg1) is str and type(arg2) is str:
...             konversiarg2 = type(arg1)(arg2)
...             return konversiarg2 + arg1
...     else:
...             return arg1 + arg2
...
>>> jum(12014,"081")
12095
>>> jum(12014.,"081")
12095.0
```

Perhatikanlah pada contoh tersebut, perintah **`type(arg2)`** juga digunakan untuk menghasilkan fungsi yang mengonversi objek awal ke dalam tipe arg2, misalnya **`int`**; **`float`**; atau **`complex`**. Perintah **`type (arg2) (arg1)`** sejatinya merupakan fungsi konversi untuk arg1 dari tipe arg1 menjadi tipe arg2.

Fungsi dapat mengeluarkan lebih dari satu nilai. Berikut ini contohnya:

```
>>> def tes (x,y):
...     a = x**y
...     b = x%y
...     return a,b
...
>>> tes (1,3)
(1, 1)
>>> c,d = tes(1,3)
>>> c
1
>>> d
1
```

Argumen Opsional Dalam Fungsi

Kita dapat melakukan penambahan argumen pada fungsi yang hendak kita definisikan. Fungsi kita tentunya memiliki nilai *default*, namun dengan adanya penambahan argumen, kita dapat melakukan perubahan pada nilai *default* yang dihasilkan oleh fungsi tersebut. Perhatikanlah contoh berikut ini:

```
>>> def fmtSatuan (x, format = "%.3f", Satuan = "meter"):
...     return x + " " + Satuan
...
>>> def fmtSatuan (x, format = "%.3f", Satuan = "meter"):
...     return format % x + " " + Satuan
...
>>> fmtSatuan(13)
'13.000 meter’
>>> fmtSatuan(13, "%.1f")
'13.0 meter'
>>> fmtSatuan(13, "%.1f", "cm")
'13.0 cm'
>>> fmtSatuan(13, "%.1f", Satuan = "cm")
'13.0 cm'
```

Perhatikanlah pada baris kedua dari belakang, kita perlu menuliskan secara eksplisit argumen opsional **`Satuan`** karena kita tidak menuliskan **`format`** sebelumnya. Secara umum alangkah lebih baik bagi kita untuk secara eksplisit senantiasa menuliskan argumen opsional untuk memperjelas bahwa argumen yang kita gunakan bersifat opsional. Berikut ini contoh penggunaannya:

```
>>> fmtSatuan(13, format = "%.1f", Satuan = "cm")
'13.0 cm'
```

Namespace Fungsi

Variabel – variabel argumen di dalam fungsi yang telah kita gunakan sebelumnya berada dalam *namespace* mereka sendiri. Hal ini berarti bahwa penugasan argumen ke nilai yang baru tidak berdampak pada nilai lamanya yang berada di luar fungsi. Perhatikan contoh berikut ini:

```
>>> def penambahan(a):
...     a = a + 1
...     return a
...
>>> a = 9
>>> penambahan(a)
10
>>> a
9
```

Mengapa nilai a tidak berubah? Karena a merupakan variabel argumen yang didefinisikan melalui perintah **`def`**. a diperlakukan sebagai variabel baru yang hanya ada di dalam fungsi. Ketika fungsi telah selesai didefinisikan dan kita keluar dari program tersebut, variabel a dalam fungsi akan dihapus dari memori dengan metode *garbage collection* oleh Python. Maka dari itu, variabel a yang kita simpan di luar fungsi tidak mengalami perubahan.

Kemudian, muncul pertanyaan bagaimana jika kita ingin mengubah variabel menggunakan fungsi? Dalam bahasa pemrograman lainnya, kalian mungkin dapat mengirimkan variabel ke dalam fungsi untuk mengubah nilainya secara langsung. Namun hal ini bukanlah cara yang ditempuh oleh para *programmer* Python. Pendekatan yang dilakukan dalam Python adalah menggunakan penugasan ke dalam suatu fungsi untuk mengembalikan suatu nilai. Pendekatan yang digunakan pada Python ini pada dasarnya jauh lebih mudah dipahami dibandingkan pada bahasa pemrograman lainnya karena secara eksplisit menunjukkan bahwa variabel telah diubah ketika kita menggunakan fungsi tersebut. Perhatikanlah contoh berikut ini untuk memperjelas pemahaman kalian:

```
>>> def penambahan(a):
...     return a + 1
...
>>> a = 5
>>> a = penambahan(a)
>>> a
6
```

Masih terdapat satu lagi kendala dalam hal ini. Objek yang dapat diubah (*mutable*) dapat diubah melalui fungsi jika kita menggunakan objek fungsi tersebut dan/atau mengakses elemen di dalamnya. Untuk memperjelas, coba kalian perhatikan contoh berikut ini:

```
>>> def mengubahlist(l):
...      l.append(7)
...      l[0] = 81
...
>>> l = [1,2,3]
>>> mengubahlist(l)
>>> l
[81, 2, 3, 7]
```

Perbedaan pada objek yang dapat diubah (*mutable*) ini berkaitan dengan pendekatan *name binding* dalam Python. Perhatikanlah konstruksi program berikut ini:

```
>>> def arg(x):
...      arg = nilai baru
>>> x = nilai
>>> fn(x)
```

Ketika kita memanggil fungsi **`fn(x)`**, Python membuat variabel baru **`arg`** di dalam *namespace* fungsi tersebut dan mengarahkannya ke lokasi data dalam memori yang ditunjuk oleh variabel x. Dengan melakukan pengaturan **`arg`** sama dengan nilai baru di dalam fungsi sekedar bermakna bahwa kita mengarahkan **`arg`** ke ke lokasi baru dalam memori sesuai dengan nilai baru tersebut, bukannya mengubah lokasi dalam memori yang berhubungan dengan x. Maka dari itu, x tidak akan mengalami perubahan.

Namun, di sisi lain coba kalian perhatikan contoh berikut ini:

```
>>> def fn(arg):
... arg[indeks] = nilai baru
>>> x = [nilai]
>>> fn(x)
```

Notasi kurung siku pada baris kedua merupakan perintah kepada Python untuk mencari lokasi elemen **`indeks`** ke n dari variabel **`arg`** dalam memori untuk menyimpan **`nilai baru`** ke dalamnya. Hal

ini terjadi karena dengan keberadaan notasi kurung siku sesudah `arg`, maka fungsi tersebut merupakan fungsi objek dari `arg`, dan oleh karena itu secara langsung juga merupakan fungsi dari memori dan data di mana `arg` terarah. Kasus yang sama juga dapat terjadi jika kita melakukan pemanggilan beberapa fungsi objek yang dapat mengubah isi fungsi tersebut, misalkan `arg.sort()`. Dalam kasus ini, nilai x akan mengalami perubahan di luar fungsi.

Memandang Fungsi Sebagai Objek

Seperti yang telah berulangkali diterangkan, dalam Python fungsi merupakan objek. Oleh karena itu, kita dapat mengirimkannya ke fungsi lain sebagai argumen. Perhatikan contoh berikut ini:

```
>>> def kwadrat(x):
...     return x*x
...
>>> def masukkankelist(l, fn):
...     return [fn(ele) for ele in l]
...
>>> l = [1,2,3,4,5,6,7,8,9,10]
>>> masukkankelist(l, kwadrat)
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Kita di sini memasukkan fungsi `kwadrat` ke fungsi `masukkankelist`. Perhatikanlah ketika kita mengirimkan suatu fungsi ke fungsi lain, kita tidak butuh menuliskan argumen lagi. Jika kita menuliskan argumen lagi, kita malah akan mengirim nilai kembalian (*return value*) dari fungsi tersebut, alih – alih fungsinya sendiri.

Untuk membuktikan bahwa fungsi dalam Python diperlakukan sebagai objek, coba kalian jalankan perintah berikut ini:

```
>>> kwadrat
<function kwadrat at 0x0000000001E0D978>
```

Nilai heksadesimal yang kita lihat tersebut merupakan lokasi memori yang menyimpan fungsi `kwadrat`. Selain itu, kita juga dapat melakukan pemeriksaan tipe:

```
>>> type(kwadrat)
<type 'function'>
```

Layaknya objek – objek lain dalam Python kita juga dapat melakukan penugasan dengan menggunakan fungsi. Berikut ini contohnya:

Pendokumentasian Fungsi

Kita dapat mendokumentasikan fungsi yang telah kita buat sendiri dalam Python dengan menuliskan *string* sebagai keterangan sesudah perintah **`def`** sebagai deskripsi bagaimana fungsi ini bekerja. Hal ini sangat bermanfaat untuk mendokumentasikan kode yang telah kalian tuliskan dan dapat memberikan penjelasan yang berguna bagi sebagian pengguna lain yang membutuhkannya. Dengan melakukan pendokumentasian, maka fungsi *built-in* **`help`** dapat mendeskripsikan fungsi yang kita tuliskan. Berikut ini contoh pendokumentasian fungsi:

```
>>> def pangkat(x,y):
...     """ Memangkatkan variabel x dan y untuk tipe numerik. Menghasilkan
...         nilai tunggal. """
...     return x**y
...
>>> help(pangkat)
Help on function pangkat in module __main__:


pangkat(x, y)
    Memangkatkan variabel x dan y untuk tipe numerik. Menghasilkan
    nilai tunggal.
```

Umumnya para *programmer* Python menggunakan tanda kutip dua tiga kali untuk menuliskan dokumentasi *string*, karena dalam fungsi – fungsi yang kompleks seringkali kita membutuhkan lebih dari satu baris untuk menuliskan dokumentasinya.Biasakanlah untuk mendokumentasikan fungsi yang telah kalian tuliskan, karena hal ini akan sangat membantu untuk pengerjaan kalian berikutnya. Dalam pendokumentasian fungsi terdapat tiga hal pokok yang wajib kalian tuliskan sebagai berikut ini:

1. Deskripsi umum tentang fungsi tersebut,
2. Penggunaan argumen dalam fungsi, dan
3. Nilai yang dihasilkan oleh fungsi tersebut beserta tipe variabelnya.

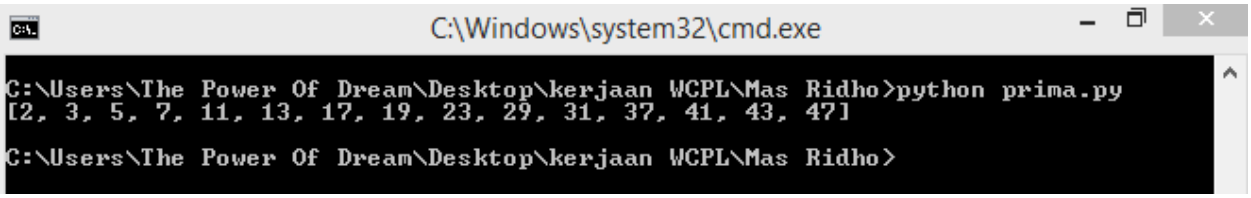
Penulisan *Script*

Sejauh ini contoh – contoh yang telah diberikan dikerjakan secara langsung dalam *prompt* interaktif. Seperti yang telah dibahas di awal tutorial ini, pemrograman dalam Python juga dapat dilakukan dengan metode *scripting* layaknya pada bahasa pemrograman lainnya. Sesungguhnya metode ini tidak ada bedanya dengan metode yang telah kita lakukan sebelumnya dengan *prompt* (hanya menyusunnya ke dalam satu file dengan ekstensi `.py`). Perhatikanlah contoh *script* `prima.py` berikut ini:

prima.py

```
1  def nextprime(primelist):
2      k = max(primelist)+1
3      while True:
4          FoundDivisor = False
5          for prime in primelist:
6              if k % prime == 0:
7                  FoundDivisor = True
8                  break
9          if FoundDivisor:
10             k +=1
11         else:
12             return k
13 upperlimit = 50
14 l = [2]
15 while l[-1] < upperlimit:
16     l.append(nextprime(l))
17 l = l[:-1]
18 print l
19
```

Kita dapat menjalankan program ini melalui *command line* dengan menuliskan `python` dan nama *script* yang hendak kita jalankan. Python akan menjalankan isi file tersebut secara berurutan sama seperti saat kita menjalankannya dalam *prompt* interaktif. Pada Windows hasilnya akan terlihat menjadi:



Modul

Kita juga dapat mengimpor *script* ke dalam lingkungan interpreter Python. Ketika file Python diimpor dengan cara demikian, maka file tersebut dinamakan modul. Penggunaan modul merupakan basis utama dalam pemrograman menggunakan Python. Dengan menggunakan modul kita dapat mengorganisasikan kode – kode yang telah kita tuliskan sebelumnya dan dapat kita impor kapanpun kita membutuhkannya. Mari kita mempelajari penggunaan modul melalui *script* bilangan prima yang telah kita buat pada contoh sebelumnya:

```
>>> import prima
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> type(prima)
<type 'module'>
>>> prima. nextprime
<function nextprime at 0x0000000001E9AD68>
>>> prima. l
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
```

Ada beberapa hal penting yang dapat kalian pelajari melalui contoh di atas:

- Kita dapat mengimpor *script* dengan menggunakan perintah **`import`**. Pasca memproses perintah tersebut, Python segera mengeksekusi file **`prima.py`**.
- Ketika Python mengeksekusi *script* yang diimpor, maka Python akan membuat objek bertipe modul.
- Kita tidak perlu menggunakan ekstensi **`.py`** ketika memberikan perintah **`import`**, Python akan mengenalinya asalkan kita membuka program Python pada direktori yang sama. Jika kita membukanya pada direktori yang berbeda, maka kita dapat menggunakan fitur PYTHONPATH yang akan kita bahas pada akhir bagian ini.

- Setiap objek yang dibuat ketika kita menjalankan file yang diimpor tidak dihapus, melainkan diubah menjadi anggota objek modul. Kita dapat mengakses fungsi dan variabel dalam modul program tersebut melalui notasi titik, seperti **`prima. l`** dan **`prima. nextprime`**.

Melalui pemanfaatan objek – objek dalam modul ini, kita dapat dengan mudah memanfaatkan bagian – bagian program yang telah kita jalankan sebelumnya. Kita juga dapat mengimpor satu modul ke modul lainnya, sehingga dapat membuat program dengan derajat variabel yang bervariasi.

Kita dapat membuat dan mengubah objek – objek modul sebagaimana yang dapat kita lakukan pada objek – objek Python lainnya. Perhatikan contoh berikut ini:

```
>>> prima .l = []
>>> prima .l
[]
>>> prima .k = 5
>>> prima .k
5
```

Mengimpor modul sebanyak dua kali bukan berarti mengeksekusikannya sebanyak dua kali juga:

```
>>> import prima
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> import prima
>>>
```

Python hanya akan mengimpor modul sebanyak satu kali karena alasan efisiensi. Contoh ketidakefisienan yang sering terjadi adalah ketika kita mengimpor modul sebanyak beberapa kali untuk mendapatkan submodul yang sama. Namun, masalah keterbatasan pengimporan modul ini dapat kita atasi dengan menggunakan fungsi **`reload`**. Berikut ini contohnya:

```
>>> import prima
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
>>> import prima
>>> reload(prima)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
<module 'prima' from 'prima.py'>
```

Terdapat perbedaan ketika kita mengeksekusi *script* secara langsung di *command line* jika dibandingkan dengan ketika kita mengimpornya ke program lain. Ketika kita menjalankan *script* dari *command line*, proses yang terjadi adalah pengeksekusian baris demi baris perintah dalam *script* tersebut, namun hal ini tidak berlaku ketika kita mengimpornya sebagai modul. Untuk lebih jelasnya, coba kalian perhatikan contoh berikut ini:

kali.py

```
1  def kali(x,y): #
2      return x*y
3
4  if __name__ == "__main__":
5      #hanya dieksekusi ketika kita menjalankannya secara langsung dari command line
6      print kali(5,10)
```

Variabel **__name__** merupakan variabel khusus dalam Python yang berfungsi untuk memberitahukan kepada kita jenis modul apa yang kita gunakan. Nilai **"__main__"** kita tuliskan, jika dan hanya jika merupakan program berjenis **main** (utama) dan dijalankan secara langsung di *command line*. Berikut ini hasil ketika kita menjalankan program tersebut secara langsung di *command line*:

```
C:\Users\The Power Of Dream\Desktop\kerjaan WCPL\Mas Ridho>python kali.py
50
```

Jika kita menjalankannya dengan perintah **import**:

```
>>> import kali
>>> kali. kali(7,9)
63
```

Perhatikanlah, Python tidak mengeksekusi perintah `kali(5,10)` ketika kita melakukan pengimporan, meskipun demikian kita masih dapat mengakses setiap objek atau fungsi yang didefinisikan dalam file tersebut.

Kita tidak dapat menuliskan *path* secara langsung dalam perintah `import`. Python dapat mengenali modul yang kita simpan pada ketiga lokasi berikut ini:

- Dalam direktori kerja kita saat ini.
- Pada direktori khusus yang dikenal sebagai PYTHONPATH.
- Pada instalasi standar Python.

PYTHONPATH merupakan variabel *system environment* yang berfungsi untuk mengarahkan Python ke folder tertentu. Untuk melakukan pengaturan PYTHONPATH dalam Windows kalian dapat klik kanan pada My Computer > Properties > Advanced > Environment Variables. Kemudian PYTHONPATH dapat kalian tambahkan ke kategori User Variables, dan kalian isi Value –nya dengan path lokasi penyimpanan *script* kalian.

Modul – Modul Standar

Python merupakan bahasa pemrograman yang memiliki *library* raksasa, berisi modul – modul siap pakai yang dapat memberikan solusi permasalahan komputasi kita. Kita tidak akan membahasnya satu per satu dalam bagian ini, karena cakupannya terlampau luas untuk tutorial Python bagi pemula. Berikut ini modul – modul standar yang sangat membantu kalian dalam komputasi ilmiah:

- `os` : berisi fungsi – fungsi untuk pengoperasian berbagai sistem operasi.
- `os.path` : berisi fungsi – fungsi untuk memanipulasi nama direktori atau *folder path*.
- `sys` : berisi fungsi – fungsi sistem program tertentu.
- `time` : memuat fungsi – fungsi untuk pengaturan waktu dalam berbagai macam format.
- `filecmp` : memuat fungsi – fungsi untuk membandingkan file dan direktori.
- `tempfile` : berisi fungsi – fungsi untuk pembuatan dan penghapusan otomatis pada file yang bersifat sementara.
- `glob` : memuat fungsi – fungsi untuk menangani ekspresi *wildcard* file tertentu (misalnya, "*.txt").

- `shutil` : memuat fungsi – fungsi untuk operasi file dalam tingkat tinggi (*high-level file operation*) (misalnya untuk penyalinan dan pemindahan file).
- `struct` : memuat fungsi – fungsi untuk keperluan penyimpanan data biner dalam bentuk *compact* dan *binary string*.
- `gzip`, `bz2`, `zipfile`, dan `tarfile` : memuat fungsi – fungsi untuk keperluan penulisan dan pembacaan dari berbagai file yang terkompresi.
- `pickle` : memuat fungsi – fungsi yang dapat mengubah berbagai objek Python memnjadi *string* sehingga dapat ditulis atau dibaca dalam suatu file.
- `hashlib` : memuat fungsi – fungsi untuk keperluan kriptografi.
- `socket` : berisi fungsi – fungsi untuk jaringan dalam tingkat rendah (*low-level networking*).
- `popen2` : berisi fungsi – fungsi yang digunakan untuk menjalankan program lain dan mendapatkan hasilnya.
- `urllib` : bersi fungsi – fungsi untuk pengumpulan data dari server internet.
- `ftplib` dan `telnetlib` : berisi fungsi – fungsi untuk *interfacing* dengan komputer – komputer lain melalui ftp dan protokol telnet.
- `audioop` dan `imageop` : berisi fungsi – fungsi yang digunakan untuk memanipulasi audio mentah dan data citra (misalnya, memotong; mengubah ukuran; dll.).

Modul – modul lainnya dapat kalian lihat secara lengkap dalam *Python Library Reference* di *Python Documentation*. Di samping modul – modul tersebut terdapat dua buah modul *add-on* yang sangat berguna dalam proses komputasi ilmiah, yaitu NumPy dan SciPy yang akan kita bahas secara khusus dalam bagian lain. Terdapat banyak sekali modul *add-on* untuk berbagai macam keperluan yang dapat kalian unduh secara gratis di internet.

Membaca Data Dari File

Hal yang paling penting dalam bahasa pemrograman ilmiah adalah kemampuannya untuk membaca dan menulis data dari dan ke suatu file. Permasalahan membaca dan menuliskan data pada file sangatlah mudah untuk dikerjakan dengan menggunakan Python. Pada bagian ini kita

akan menggunakan file `data.txt` yang merupakan data cuaca hasil pengamatan *Automatic Weather Station* (AWS) setiap lima menit di titik observasi Boscha, Lembang pada tanggal 13 Maret 2017 mulai pukul 00.00 WIB malam hingga pukul 01.00 WIB. Berikut ini isi `data.txt` :

```
--Date-Time---|-Press-|-Temp|-RHum|-WSpd|-WDir|-GSpd|-GDir|-Rain-|-Batt
    20170313 00:00, 886.79, 20.0, 85.9, 1.5,259.9, 2.7,315.0,0, 5.0
    20170313 00:05, 886.79, 20.0, 85.4, 1.5,259.9, 3.3,315.0,0, 5.0
    20170313 00:10, 886.81, 20.0, 85.2, 1.4,277.7, 2.7,270.0,0, 5.0
    20170313 00:15, 886.75, 20.0, 85.1, 1.2,280.8, 3.3,270.0,0, 5.0
    20170313 00:20, 886.76, 20.0, 85.0, 1.2,259.9, 2.7,315.0,0, 5.0
    20170313 00:25, 886.69, 20.0, 84.6, 1.3,254.4, 2.0,270.0,0, 5.0
    20170313 00:30, 886.60, 20.0, 84.9, 1.4,271.5, 2.7,270.0,0, 5.0
    20170313 00:35, 886.56, 20.0, 85.1, 1.2,263.0, 2.7,270.0,0, 5.0
    20170313 00:40, 886.54, 20.0, 85.0, 1.5,256.8, 3.3,270.0,0, 5.0
    20170313 00:45, 886.53, 20.0, 84.9, 1.5,269.2, 3.3,270.0,0, 5.0
    20170313 00:50, 886.47, 20.0, 85.1, 1.0,245.1, 2.7,315.0,0, 5.0
    20170313 00:55, 886.43, 20.0, 85.0, 1.4,281.6, 2.7,315.0,0, 5.0
    20170313 01:00, 886.42, 20.0, 85.0, 1.4,271.5, 2.7,315.0,0, 5.0
```

Berikut ini contoh cara membuka objek file `data.txt` dan membaca seluruh isinya sebagai *string*:

```
>>> f = file("data.txt", "r")
>>> s = f.read()
>>> f.close()
>>> s
'#--Date-Time---|-Press-|-Temp|-RHum|-WSpd|-WDir|-GSpd|-GDir|-Ra
13 00:00, 886.79, 20.0, 85.9, 1.5,259.9, 2.7,315.0,0, 5.0\n2017
9, 20.0, 85.4, 1.5,259.9, 3.3,315.0,0, 5.0\n20170313 00:10, 886
1.4,277.7, 2.7,270.0,0, 5.0\n20170313 00:15, 886.75, 20.0, 85.1
270.0,0, 5.0\n20170313 00:20, 886.76, 20.0, 85.0, 1.2,259.9, 2.
0170313 00:25, 886.69, 20.0, 84.6, 1.3,254.4, 2.0,270.0,0, 5.0\
886.60, 20.0, 84.9, 1.4,271.5, 2.7,270.0,0, 5.0\n20170313 00:35
5.1, 1.2,263.0, 2.7,270.0,0, 5.0\n20170313 00:40, 886.54, 20.0,
 3.3,270.0,0, 5.0\n20170313 00:45, 886.53, 20.0, 84.9, 1.5,269.
.0\n20170313 00:50, 886.47, 20.0, 85.1, 1.0,245.1, 2.7,315.0,0,
:55, 886.43, 20.0, 85.0, 1.4,281.6, 2.7,315.0,0, 5.0\n20170313
.0, 85.0, 1.4,271.5, 2.7,315.0,0, 5.0\n'
>>> print s
#--Date-Time---|-Press-|-Temp|-RHum|-WSpd|-WDir|-GSpd|-GDir|-Rai
20170313 00:00, 886.79, 20.0, 85.9, 1.5,259.9, 2.7,315.0,0, 5.0
20170313 00:05, 886.79, 20.0, 85.4, 1.5,259.9, 3.3,315.0,0, 5.0
20170313 00:10, 886.81, 20.0, 85.2, 1.4,277.7, 2.7,270.0,0, 5.0
20170313 00:15, 886.75, 20.0, 85.1, 1.2,280.8, 3.3,270.0,0, 5.0
20170313 00:20, 886.76, 20.0, 85.0, 1.2,259.9, 2.7,315.0,0, 5.0
20170313 00:25, 886.69, 20.0, 84.6, 1.3,254.4, 2.0,270.0,0, 5.0
20170313 00:30, 886.60, 20.0, 84.9, 1.4,271.5, 2.7,270.0,0, 5.0
20170313 00:35, 886.56, 20.0, 85.1, 1.2,263.0, 2.7,270.0,0, 5.0
20170313 00:40, 886.54, 20.0, 85.0, 1.5,256.8, 3.3,270.0,0, 5.0
20170313 00:45, 886.53, 20.0, 84.9, 1.5,269.2, 3.3,270.0,0, 5.0
20170313 00:50, 886.47, 20.0, 85.1, 1.0,245.1, 2.7,315.0,0, 5.0
20170313 00:55, 886.43, 20.0, 85.0, 1.4,281.6, 2.7,315.0,0, 5.0
20170313 01:00, 886.42, 20.0, 85.0, 1.4,271.5, 2.7,315.0,0, 5.0
```

Terdapat dua buah argumen dalam fungsi **`file`**, yaitu nama file dan yang diikuti oleh sebuah *string* **`"r"`** yang mengindikasikan tujuan kita membuka file tersebut adalah untuk membacanya. Kita juga dapat tidak menuliskan argumen kedua, karena Python akan secara otomatis memprosesnya sebagai **`"r"`** secara *default*, namun alangkah lebih baik jika kita tetap menuliskannya untuk memperjelas program yang kita tuliskan. Kita juga dapat menggunakan perintah **`open`** sebagai pengganti **`file`**.

Selanjutnya, kita dapat membaca seluruh isi file tersebut dalam bentuk *string* dengan menggunakan metode **`read()`**. Python akan membaca keseluruhan isi file, termasuk bagian kosong, seperti spasi dan perpindahan baris.

Ketika kita telah melakukan pembacaan isi file, ada baiknya kita menjalankan fungsi **`close`** yang akan menghentikan fungsi pembacaan pada file kita. Pada banyak kasus, Python akan secara otomatis menutup file yang telah dibuka dengan *garbage collection routine*, jika tidak lagi terdapat objek yang mengarah pada file tersebut. Perhatikanlah perintah berikut ini:

```
>>> s = file("data.txt", "r").read()
```

Perintah ini akan menghasilkan hasil yang sama dengan contoh terakhir, namun objek file yang dibuat tidak akan bertahan pasca eksekusi. Proses kerjanya adalah sebagai berikut, pertama kali fungsi **`file`** membuat objek, kemudian metode **`read()`** mengekstraksi konten file di dalamnya dan menempatkannya ke dalam variabel **`s`**. Sesudah operasi ini selesai, tidak terdapat lagi variabel yang mengarah ke objek file sama sekali, oleh karena itu file akan otomatis tertutup. Objek file yang dibuat di dalam fungsi juga akan secara otomatis tertutup ketika kita keluar dari fungsi tersebut karena variabel yang dibuat di dalam fungsi telah terhapus ketika kita keluar dari fungsi itu.

Kita juga dapat membaca file sebagai *list* untuk setiap barisnya, berikut ini contohnya:

```
>>> l = file("data.txt", "r").readlines()
>>> l
['#--Date-Time---|-Press-|-Temp|-RHum|-WSpd|-WDir|-GSpd|-GDir|-Rain-|-Batt\n', '2
0170313 00:00, 886.79, 20.0, 85.9, 1.5,259.9, 2.7,315.0,0, 5.0\n', '20170313 00:
05, 886.79, 20.0, 85.4, 1.5,259.9, 3.3,315.0,0, 5.0\n', '20170313 00:10, 886.81,
 20.0, 85.2, 1.4,277.7, 2.7,270.0,0, 5.0\n', '20170313 00:15, 886.75, 20.0, 85.1
, 1.2,280.8, 3.3,270.0,0, 5.0\n', '20170313 00:20, 886.76, 20.0, 85.0, 1.2,259.9
, 2.7,315.0,0, 5.0\n', '20170313 00:25, 886.69, 20.0, 84.6, 1.3,254.4, 2.0,270.0
,0, 5.0\n', '20170313 00:30, 886.60, 20.0, 84.9, 1.4,271.5, 2.7,270.0,0, 5.0\n',
 '20170313 00:35, 886.56, 20.0, 85.1, 1.2,263.0, 2.7,270.0,0, 5.0\n', '20170313
00:40, 886.54, 20.0, 85.0, 1.5,256.8, 3.3,270.0,0, 5.0\n', '20170313 00:45, 886.
53, 20.0, 84.9, 1.5,269.2, 3.3,270.0,0, 5.0\n', '20170313 00:50, 886.47, 20.0, 8
5.1, 1.0,245.1, 2.7,315.0,0, 5.0\n', '20170313 00:55, 886.43, 20.0, 85.0, 1.4,28
1.6, 2.7,315.0,0, 5.0\n', '20170313 01:00, 886.42, 20.0, 85.0, 1.4,271.5, 2.7,31
5.0,0, 5.0\n']
```

Secara operasional hasil dari fungsi **`readlines()`** sama dengan yang dihasilkan oleh fungsi **`read().split('\n')`**.

Jika file yang hendak kita baca merupakan file besar, mungkin tidak efisien jika membacanya pada satu waktu. Sebagai gantinya, kita dapat membaca satu baris sekaligus menggunakan fungsi **`readline`**. Berikut ini contohnya:

```
>>> f = file("data.txt", "r")
>>> s = "dummy"
>>> while len(s):
...     s = f.readline()
...     if not s .startswith("#"): print s .strip()
...
20170313 00:00, 886.79, 20.0, 85.9, 1.5,259.9, 2.7,315.0,0, 5.0
20170313 00:05, 886.79, 20.0, 85.4, 1.5,259.9, 3.3,315.0,0, 5.0
20170313 00:10, 886.81, 20.0, 85.2, 1.4,277.7, 2.7,270.0,0, 5.0
20170313 00:15, 886.75, 20.0, 85.1, 1.2,280.8, 3.3,270.0,0, 5.0
20170313 00:20, 886.76, 20.0, 85.0, 1.2,259.9, 2.7,315.0,0, 5.0
20170313 00:25, 886.69, 20.0, 84.6, 1.3,254.4, 2.0,270.0,0, 5.0
20170313 00:30, 886.60, 20.0, 84.9, 1.4,271.5, 2.7,270.0,0, 5.0
20170313 00:35, 886.56, 20.0, 85.1, 1.2,263.0, 2.7,270.0,0, 5.0
20170313 00:40, 886.54, 20.0, 85.0, 1.5,256.8, 3.3,270.0,0, 5.0
20170313 00:45, 886.53, 20.0, 84.9, 1.5,269.2, 3.3,270.0,0, 5.0
20170313 00:50, 886.47, 20.0, 85.1, 1.0,245.1, 2.7,315.0,0, 5.0
20170313 00:55, 886.43, 20.0, 85.0, 1.4,281.6, 2.7,315.0,0, 5.0
20170313 01:00, 886.42, 20.0, 85.0, 1.4,271.5, 2.7,315.0,0, 5.0

>>> f.close()
```

Pada contoh tersebut, kita mem-*print* seluruh baris yang tidak diawali dengan tanda '#'. Kemudian, pengulangan **`while`** akan terus bekerja selama perintah **`readline()`** tidak menemukan *string* bernilai nol. Ketika Python menyelesaikan pembacaan hingga akhir file, perintah **`readline`** akan menghasilkan *string* bernilai nol. Patut kalian ketahui, bahwa **`readline()`** akan membaca seluruh baris, termasuk karakter perpindahan baris '\n' yang dikenali bukan sebagai *string* bernilai nol. Maka dari itu, kita juga menggunakan fungsi **`strip()`** untuk menghapus karakter '\n'.

Kita juga dapat menggunakan argumen opsional **`size`** pada fungsi **`read`** dan **`readline`** yang mengatur jumlah maksimum karakter (bit) yang dibaca Python pada suatu waktu. Penggunaan argumen opsional berikutnya akan mengikuti karakter berikutnya pada file hingga mencapai bagian akhir file, di mana Python akan membacanya sebagai string bernilai nol. Untuk lebih jelasnya perhatikan contoh berikut ini:

```
>>> f = file("data.txt","r")
>>> f.read(9)
'#--Date-T'
>>> f.read(9)
'ime---|-P'
```

Fungsi **`seek`** dapat kita gunakan untuk berpindah ke lokasi bit tertentu di dalam file, sementara fungsi **`tell`** dapat kita gunakan untuk mengetahui jumlah bit di posisi tersebut. Berikut ini contohnya:

```
>>> f = file("data.txt","r")
>>> f.seek(9)
>>> f.read(9)
'ime---|-P'
>>> f.tell()
4L
>>> f.close()
>>> f.close()
```

Huruf ‘L’ di belakang angka 4 mengindikasikan bahwa tipe bit tersebut merupakan *integer* panjang, hal ini dapat terjadi karena *integer* biasa tidak sanggup menampung lokasi bit dalam file yang berukuran besar.

Kita akan mengakhiri bagian ini dengan menunjukkan kemudahan Python dalam menangani file. Misalkan kita ingin membaca file `dat.txt` berikut ini menjadi *list*:

```
#pressure temperature average_energy

1.0 1.0 -50.0

1.0 1.5 -27.8

2.0 1.0 -14.5

2.0 1.5 -11.2
```

Untuk mencapai hal tersebut, kita butuh untuk mengabaikan *comment*, mengubah data menjadi tipe *float*, dan menstrukturkannya ke dalam *list*, sebagai berikut:

```
>>> f = file("dat.txt","r")
>>> Data = []
>>> line = f.readline()
>>> while len(line):
...     if not line.startswith("#"):
...             l = line.split()
...             Pres = float(l[0])
...             Temp = float (l[1])
...             Ene = float(l[2])
...             Data.append([Pres, Temp, Ene])
...     line = f.readline()
...
...
>>> f.close()
>>> Data
[[1.0, 1.0, -50.0], [1.0, 1.5, -27.8], [2.0, 1.0, -14.5], [2.0, 1.5, -11.2]]
```

Menuliskan File

Dalam Python, penulisan data ke dalam suatu file sangatlah mudah untuk dilakukan. Untuk memulai penulisan ke file baru, kita harus terlebih dahulu membuka sebuah objek file dengan argumen “w”, seperti berikut ini:

```
>>> f = file("baru.txt", "w")
>>> f.write("Baris pertama")
>>> f.write("masih di baris pertama")
>>> f.write("\n masuk ke baris kedua")
>>> f.close()
```

Berikut tampilan file yang telah kita tuliskan:

```
Baris pertamamasih di baris pertama
 masuk ke baris kedua
```

Argumen “w” berfungsi untuk memberitahukan Python untuk membuat file baru yang siap untuk dituliskan, dan fungsi **`write`** akan menuliskan kata demi kata *string* ke posisi saat ini di dalam file. Untuk menambahkan *string* pada baris baru, kita dapat menggunakan tanda “\n”.

Jika kita hendak menambahkan tulisan pada file yang sudah jadi, maka kita tidak dapat menggunakan argumen “w” karena akan menghapus seluruh isi file tersebut, digantikan dengan tulisan baru kita. Sebagai gantinya, kita dapat menggunakan argumen “a” untuk menambahkan teks, berikut ini contohnya:

```
>>> f = file("baru.txt", "a")
>>> f.write("\nBaris ketiga")
>>> f.close()
```

File kita akan tampak seperti berikut ini:

```
Baris pertamamasih di baris pertama
 masuk ke baris kedua
Baris ketiga >>> f.close()
```

Fungsi **`write`** hanya dapat menerima masukkan dalam tipe *string*. Dengan demikian berarti, nilai numerik harus kita ubah ke dalam format *string* sebelum dituliskan ke dalam file. Berikut ini contohnya:

```
>>> f = file("baru.txt", "w")
>>> pi = 3.14159
>>> f.write(str(pi))
>>> f.write('\n')
>>> f.write("%.2f" %pi)
>>> f.close()
```

Hasilnya adalah:

```
3.14159
3.14
```

Data Biner dan File Terkompresi

Ketika kita bekerja dengan data numerik, tidaklah efisien bagi kita untuk menuliskannya dalam format `.txt` karena memakan banyak tempat di memori, terutama untuk penyimpanan data *float* yang presisi. Pada bagian ini kalian akan mempelajari dua pendekatan yang umunya digunakan oleh para *programmer* Python untuk menuliskan data numerik secara efisien ke dalam file dengan ukuran kecil.

Pendekatan pertama adalah menuliskan file ini dalam format yang tidak dapat dibaca, melainkan dalam format yang mirip ke dalam memori. Untuk melakukannya, kita harus mengubah nilai data menjadi representasi biner dalam format *string*. Kita dapat menggunakan modul **`struct`** untuk mengerjakannya. Namun, karena penggunaan modul ini lebih banyak menggunakan pendekatan bahasa C, ketimbang Python, maka akan sedikit membingungkan.

Alternatif yang kedua adalah menuliskan dan membaca dari file terkompresi. Dengan cara ini, data numerik akan dituliskan dalam bentuk yang dapat dibaca manusia (*human-readable*) dapat kita kompresikan sehingga tidak memboroskan memori. Pendekatan ini seringkali lebih mudah dikerjakan karena nilai numerik pada data masih dapat kita kenali ketika file data ini didekompresikan untuk keperluan di luar lingkungan pemrograman Python.

Kita beruntung karena Python telah menyediakan modul yang memungkinkan kita untuk menuliskan dan membaca dalam format file terkompresi populer dengan cara yang cukup mudah. Terdapat dua format yang dapat kalian coba, yaitu format Gzip yang dapat diproses dengan cepat dan format Bzip2 yang dapat mengkompres file berukuran lebih besar namun agak lambat. Keduanya merupakan format file terkompresi yang umum digunakan, bersifat *open source*, dapat dibaca dalam kebanyakan program dekompresi, serta berbasis file tunggal.

Untuk menuliskan file Gzip baru, kita harus mengimpor modul **`gzip`** dan membuat objek **`GzipFile`** sebagaimana kita membuat objek file pada bagian – bagian terdahulu:

```
>>> import gzip
>>> f = gzip.GzipFile("data.txt.gz", "w")
>>> f.write("tes data terkompresi.")
21
>>> f.close()
>>> print gzip.GzipFile("data.txt.gz", "r").read()
tes data terkompresi.
```

Dalam contoh tersebut, Python menangani seluruh proses kompresi (dan juga dekompresi) file di belakang layar. Satu – satunya perbedaan antara penanganan objek file yang telah kita lakukan sebelumnya dengan penanganan file terkompresi hanyalah pada fungsi file **`gzip . GzipFile`**. Secara umum, objek **`gzip`** berlaku sama seperti objek file lainnya dan mampu menjalankan fungsi – fungsi yang sama (**`read`**, **`readline`**, **`readlines`**, **`write`**). Hal ini jelas memudahkan kita untuk menyimpan data dalam file terkompresi. Satu – satunya pengecualian adalah bahwa fungsi **`seek`** dan **`tell`** pada objek file terkompresi tidak bekerja sama persis seperti pada objek file lainnya, maka dari itu lebih baik kita menghindari penggunaan kedua fungsi ini jika bekerja pada file terkompresi.

Modul **`bz2`** juga dapat kita operasikan dengan cara yang sama. Berikut ini contohnya:

```
>>> import bz2
>>> f = bz2.BZ2File("data.txt.bz2", "w")
>>> f.write("tes data terkompresi.")
>>> f.close()
>>> print bz2.BZ2File("data.txt.bz2", "r").read()
tes data terkompresi.
```

Secara umum, proses kompresi hanya direkomendasikan jika kita bekerja pada data dengan ukuran besar (misalkan melebihi 1 MB) yang tidak begitu sering dibaca dan ditulis dalam program yang kita buat. Untuk file berukuran kecil yang sering kita gunakan dalam program, agaknya tidak efisien jika kita mengoperasikannya dalam bentuk file terkompresi.

Fungsi *File System*

Python menawarkan sejumlah modul dan fungsi – fungsi yang memungkinkan kita untuk mengakses dan memanipulasi file dan direktori dalam memori. Python mengenali hirarki direktori tertentu dengan menggunakan garis miring yang berlaku secara universal pada setiap sistem operasi yang digunakan (Linux, Windows, ataupun MacOS). Pada sistem operasi Windows, kita juga dapat menggunakan karakter *backslash*, namun kita harus menghindari penggunaan karakter "\" pada *string* karena secara normal Python akan menafsirkannya sebagai kode khusus yang sedang kita gunakan. Sebagai contoh, kedua perintah berikut ini merujuk pada lokasi yang sama dalam sistem operasi Windows:

```
>>> print "c:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\Mas Ridho\\data
.txt"
c:\Users\The Power Of Dream\Desktop\kerjaan WCPL\Mas Ridho\data.txt
>>> print "c:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho/data.txt"
c:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho/data.txt
```

Modul **os** memiliki banyak koleksi fungsi file yang sangat bermanfaat. Khususnya submodul **os.path** yang menyediakan banyak fungsi untuk memanipulasi *path* dan file. Misalnya pada contoh berikut ini di mana kita dapat memisahkan file dengan *path*:

```
>>> import os
>>> p = "c:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho/data.txt"
>>> os.path.basename(p)
'data.txt'
>>> os.path.dirname(p)
'c:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho'
>>> os.path.split(p)
('c:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho', 'data.txt')
```

Kebalikan dari fungsi **split** adalah fungsi **join** yang menggabungkan *path* dan file, sebagaimana contoh berikut ini:

```
>>> os.path.join("c:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\Mas Ridho", "
data.txt")
'c:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\Mas Ridho\\data.txt'
>>> os.path.join("c:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\" ,"Mas
Ridho", "data.txt")
'c:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\Mas Ridho\\data.txt'
```

Jika *path* yang kita gunakan merupakan *path* relatif terhadap direktori tempat kita bekerja saat ini dan kita ingin untuk mengubahnya menjadi *path* absolut, maka kita dapat melakukan hal sebagai berikut ini:

```
>>> os.path.abspath("Users/The Power of Dream/Desktop/kerjaan WCPL/Mas Ridho/dat
a.txt")
'C:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\Mas Ridho\\Users\\The Pow
er of Dream\\Desktop\\kerjaan WCPL\\Mas Ridho\\data.txt'
```

Terdapat beberapa fungsi yang memungkinkan kita untuk mengetahui keberadaan dan jenis file serta direktori yang hendak kita ketahui:

```
>>> p = "C:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho/data.txt"
>>> os.path.exists(p)
True
>>> os.path.isfile(p)
True
>>> os.path.isdir(p)
False
```

Di sini, fungsi **`isfile`** dan **`isdir`** berguna untuk mengetahui apakah jenis objek tersebut berupa file ataukah direktori.

Kita juga dapat mengetahui besar ukuran file (bit) dalam suatu direktori:

```
>>> os.path.getsize("C:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho/
data.txt")
905L
```

Beberapa fungsi dalam modul **`os`** memungkinkan kita untuk mengetahui pada direktori mana kita bekerja dan mengubah direktori tempat bekerja kita tersebut. Berikut ini contohnya:

```
>>> os.getcwd()
'C:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\Mas Ridho'
>>> os.chdir("..")
>>> os.getcwd()
'C:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL'
```

Notasi ".." berfungsi untuk mengubah direktori tempat kita bekerja satu tingkat di atas direktori tempat kita bekerja saat ini.

Kita juga dapat membuat direktori baru:

```
>>> os.mkdir("C:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho/baru")
```

Menghapus file di dalam direktori:

```
>>> os.remove("C:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho/hapus.
txt")
```

Menghapus direktori:

```
>>> os.rmdir("C:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho/baru")
```

Mengubah nama file:

```
>>> os.rename("C:/Users/The Power Of Dream/Desktop/kerjaan WCPL/Mas Ridho/data.t
xt", "C:/Users/The Power of Dream/Desktop/kerjaan WCPL/Mas Ridho/tada.txt")
```

Modul `shutil` dapat kita manfaatkan untuk menyalin dan memindahkan file. Berikut ini contohnya:

```
>>> import shutil
>>> shutil.copy("C:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\Mas Ridho
\\tada.txt", "C:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\Mas Ridho\\s
alin.txt")
>>> shutil.move("C:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\Mas Ridho
\\tada.txt", "C:\\Users\\The Power Of Dream\\Desktop\\kerjaan WCPL\\pindah.txt")
```

Kita dapat memanfaatkan modul **`glob`** untuk menemukan jenis file tertentu dalam direktori *wildcard routine* yang disediakannya. Berikut ini contohnya:

```
>>> import glob
>>> glob.glob("E:/TA Baru/indeks nino/grinsted-wavelet-coherence-d987ea4/*.dat")

['E:/TA Baru/indeks nino/grinsted-wavelet-coherence-d987ea4\\frekuensi_moderat_k
ering.dat', 'E:/TA Baru/indeks nino/grinsted-wavelet-coherence-d987ea4\\frekuens
i_severe_kering.dat', 'E:/TA Baru/indeks nino/grinsted-wavelet-coherence-d987ea4
\\tos_HadCM3_past1000_NPAC.dat']
```

Wildcard "*" akan mencocokan seluruh jenis file yang sama tanpa mempedulikan ukuran karakter. Sementara itu, karaketer *wildcard* "?" akan mencocokan dengan tipe file yang memiliki panjang satu karakter. Karena pada direktori tersebut tidak terdapat file `.dat` dengan panjang satu karakter, maka hasilnya adalah himpunan kosong:

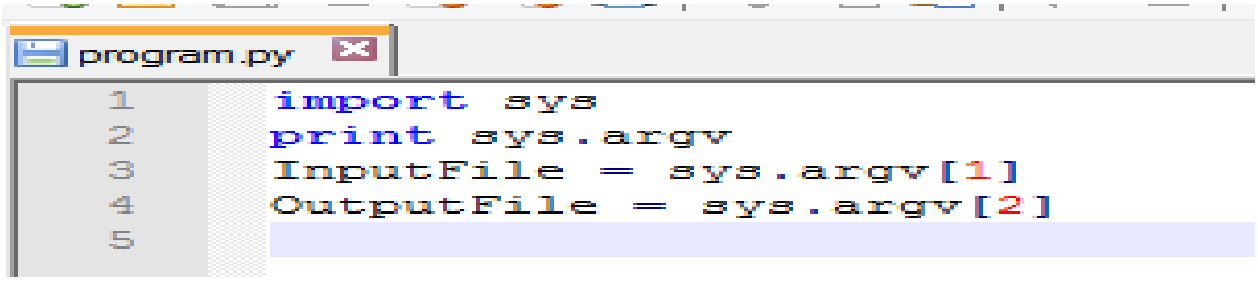
```
>>> glob.glob("E:/TA Baru/indeks nino/grinsted-wavelet-coherence-d987ea4/*/?.dat
")
[]
```

Kita juga dapat memanfaatkan *list comprehension* untuk mencari direktori atau file yang kita inginkan, seperti contoh berikut ini:

```
>>> [g for g in glob.glob("g*") if os.path.isdir(g)]
['grinsted-wavelet-coherence-d987ea4']
```

Argumen *Command Line*

Umum bagi para *programmer* untuk menjalankan opsi program dari *command line* (*command prompt* untuk Windows, atau terminal di Linux dan MacOS). Umumnya seorang *programmer* menyediakan beberapa argumen yang dapat dideteksi oleh program. Misalkan kita ingin program yang kita buat mempunyai *input* `in.txt` dan luaran `out.txt`. Untuk melakukannya kita dapat menggunakan modul **`sys`** dan fungsi **`argv`**. Langkah awal yang kita lakukan adalah membuat **`program.py`**:



Lalu kita jalankan **`program.py`** via *command line*:

```
C:\Python27>python program.py in.txt out.txt
['program.py', 'in.txt', 'out.txt']
```

Perhatikanlah bahwa **`argv`** merupakan sebuah *list* yang memuat argumen *string* secara berurutan. Argumen pertama yang terindeks 0 di dalam *list* merupakan program yang hendak kita jalankan. Argumen berikutnya dikenali oleh Python melalui pemisahan dengan spasi ketika kita menjalankan program.

Kelas

Sejauh ini kita hanya berurusan dengan tipe objek *built-in* seperti *float* dan *integer*. Di samping tipe – tipe *built-in* tersebut, Python juga memungkinkan kita untuk membuat tipe objek baru yang dikenal sebagai kelas. Dengan menggunakan kelas kita dapat mendesain objek secara bebas. Berikut ini contoh pembentukan kelas jenis atom:

atom.py

```
1  class KelasAtom:
2      def __init__(self, Kelajuan, Unsur = "C", Massa = 12.0)
3          self.Kelajuan = Kelajuan
4          self.Unsur = Unsur
5          self.Massa = Massa
6      def Momentum(self):
7          return self.Massa * self.Kelajuan
```

Kita dapat mengimpor modul **`atom.py`** dan mendefinisikan **`KelasAtom`** yang berbeda:

```
>>> import atom
>>> a = atom.KelasAtom(2.0, Unsur = 'O', Massa = 16.0)
>>> b = atom.KelasAtom(1.0)
>>> a.Unsur
'O'
>>> a.Massa
16.0
>>> a.Momentum()
32.0
>>> b.Unsur
'C'
>>> b.Kelajuan
1.0
```

Dalam contoh tersebut, kita memerintahkan Python untuk membentuk kelas baru yaitu `KelasAtom` dengan perintah `class`. Setelah kita membuat kelas, kita harus memberikan definisi berkenaan dengan kelas tersebut. Definisi pertama pada contoh tersebut adalah fungsi khusus `__init__` yang bertindak sebagai konstruktor bagi kelas, yang berarti fungsi ini akan dijalankan secara otomatis oleh Python setiap kali objek baru bertipe `KelasAtom` dibuat. Sesungguhnya terdapat banyak sekali fungsi khusus untuk mendefinisikan kelas, yang selalu diawali dan diakhiri dengan notasi garis bawah dua kali.

Perhatikanlah pada `KelasAtom`, argumen pertama yang kita berikan terhadap fungsi `__init__` adalah objek `self`. Hal ini merupakan fitur umum yang berlaku untuk seluruh fungsi kelas. Sintaks tersebut mengindikasikan bahwa objek akan secara otomatis terkirim ke fungsi ketika kita memanggil fungsi itu sendiri. Dengan demikian, memungkinkan bagi kita untuk membuat perubahan pada objek dengan cara memanipulasi variabel `self`, sebagai contoh kita dapat menambahkan anggota objek baru dengan menuliskan `self.X = Y`. Pendekatan ini nampak tidak umum, namun hal ini menyederhanakan proses di belakang layar pada pendefinisian kelas Python yang sesungguhnya cukup rumit.

Fungsi `__init__` merupakan dasar argumen yang digunakan ketika kita membentuk objek dengan perintah `atom.KelasAtom(2.0, Unsur = 'O', Massa = 16.0)`. Seperti fungsi – fungsi lainnya dalam Python, terdapat argumen opsional juga dalam fungsi ini.

Anggota – anggota objek dapat kita akses dengan menggunakan notasi titik, seperti yang telah ditunjukkan di atas.Setiap objek dari kelas tersebut mendapatkan anggota objeknya sendiri yang terpisah dari objek – objek lain dalam kelas itu. Fungsi juga dapat didefinisikan sebagai anggota objek, seperti yang kita lakukan pada fungsi `Momentum` dalam contoh sebelumnya.

Terdapat fungsi – fungsi khusus yang dapat digunakan untuk memberitahukan Python bagaimana cara menggunakan objek dalam operasi tertentu. Berikut ini daftar fungsi – fungsi khusus yang dapat digunakan dalam operasi kelas:

Fungsi Khusus	**Deskripsi**

`__del__(self)`	Berfungsi sebagai destruktor. Dipanggil ketika suatu objek hendak dihapus menggunakan `del` atau via *garbage collection*.
`__repr__(self)`	Menghasilkan representasi *string* dari suatu objek. Digunakan oleh perintah `print` misalnya.
`__cmp__(self, other)`	Digunakan sebagai metode perbandingan dengan objek lain. Menghasilkan nilai negatif, ketika `self < other`, nol jika `self == other`, dan positif jika `self > other`. Dipergunakan untuk mengevaluasi perintah perbandingan antar dua buah objek misalkan `a > b`, atau untuk pengurutan objek.
`__len__(self)`	Menghasilkan panjang suatu objek, dipergunakan oleh fungsi `len`.

`__getitem__(self, key)`
`__setitem__(self, key, value)`
`__delitem__(self, key)`
	Mendefinisikan metode untuk mengakses dan mengubah elemen dari suatu objek dengan menggunakan notasi kurung, misalnya `a[key] = value`.

`__contains__(self, item)`	Memanggil suatu objek ketika kita menggunakan perintah `in`, misalnya ketika kita memerintahkan `item in a`.

`__add__(self, other)`
`__sub__(self, other)`
`__mul__(self, other)`
`__div__(self, other)`
	Digunakan ketika kita melakukan beragam operasi aritmatika pada objek, misalnya `a + b`, `a * b`, `a / b`, `a % b`, dan `a ** b`. Dalam bahasa

`__mod__(self, other)`
 	
`__pow__(self, other)`
	pemrograman lainnya mungkin metode ini dikenal sebagai *operator overloading*.

Kelas mungkin merupakan cara termudah bagi kita untuk mengorganisasikan data ketika melakukan komputasi ilmiah. Meskipun demikian hal ini bukan tanpa kelemahan, terkadang dengan mengorganisasikan data kita ke dalam kelas, program yang kita buat menjadi lambat ketika dijalankan. Perhatikanlah contoh kelas atom yang tadi kita buat. Kita sesungguhnya dapat menempatkan komponen vektor kecepatan dalam kelas tersebut, namun hal ini akan memperlambat program yang hendak kita jalankan karena akan membuat Python mengakses memori setiap variabel individual dalam kelas secara bergantian.Tentu akan lebih efisien bagi Python untuk menyimpan atribut tersebut dalam sebuah *array* besar yang menempati satu posisi khusus dalam memori. Tentu pemisahan kelas kedalam *array* ini nampak membingungkan, namun dalam dunia komputasi ilmiah yang mementingkan efisiensi penggunaan memori, hal ini mutlak untuk dilakukan.

Eksepsi

Python menyediakan bagi kita cara yang sangat mudah untuk memeriksa kesalahan pada program yang kita buat dengan menggunakan perintah **`try`** dan **`except`**. Perhatikanlah contoh berikut ini:

tes.py

```
1  def kali(x, y):
2      try:
3          ret = x * y
4      except StandardError:
5          ret = 0
6      return ret
```

Di sini kita telah mendefinisikan fungsi perkalian yang berlaku untuk seluruh tipe data. Jika ternyata operasi perkalian tidak dapat dilakukan pada tipe – tipe data tertentu, maka *error* yang terjadi akan ditangani oleh perintah **`except`**. Alih – alih menghentikan program secara keseluruhan, dengan eksepsi kita dapat mengantisipasi dan mengetahui pada bagian mana kita membuat kesalahan yang akan ditangani oleh perintah **`try`**. Untuk dapatlebih memahaminya, coba kalian perhatikan contoh berikut ini:

```
>>> import tes

>>> tes.kali(2, 7)

14

>>> tes.kali("2", "7")

0

>>> "2" * "7"

Traceback (most recent call last):

  File "<stdin>", line 1, in <module>

TypeError: can't multiply sequence by non-int of type 'str'
```

Kesalahan pada program di atas masuk ke dalam kategori kesalahan **`StandardError`** yang merupakan kategori kesalahan yang luas, di mana mencakup juga **`TypeError`** ke dalamnya. Terdapat hirarki *error* tersendiri yang dapat dikenali oleh Python, seperti yang dapat kalian lihat berikut ini:

```
BaseException
      +-- SystemExit
      +-- KeyboardInterrupt
      +-- Exception
            +-- GeneratorExit
            +-- StopIteration
            +-- StandardError
            |     +-- ArithmeticError
            |     |     +-- FloatingPointError
            |     |     +-- OverflowError
            |     |     +-- ZeroDivisionError
            |     +-- AssertionError
            |     +-- AttributeError
            |     +-- EnvironmentError
            |     |     +-- IOError
            |     |     +-- OSError
            |     |           +-- WindowsError (Windows)
            |     |           +-- VMSError (VMS)
            |     +-- EOFError
            |     +-- ImportError
            |     +-- LookupError
            |     |     +-- IndexError
            |     |     +-- KeyError
            |     +-- MemoryError
            |     +-- NameError
            |     |     +-- UnboundLocalError
```

```
|     +-- ReferenceError
|     +-- RuntimeError
|     | +-- NotImplementedError
|     +-- SyntaxError
|     |     +-- IndentationError
|     |           +-- TabError
|     +-- SystemError
|     +-- TypeError
|     +-- ValueError
|     |     +-- UnicodeError
|     |           +-- UnicodeDecodeError
|     |           +-- UnicodeEncodeError
|     |           +-- UnicodeTranslateError
+-- Warning
      +-- DeprecationWarning
      +-- PendingDeprecationWarning
      +-- RuntimeWarning
      +-- SyntaxWarning
      +-- UserWarning
      +-- FutureWarning
  +-- ImportWarning
  +-- UnicodeWarning
```

Kita dapat membangkitkan kesalahan tertentu dengan menggunakan perintah **raise**, meskipun sesungguhnya tidak terdapat kesalahan tersebut dalam program yang kita buat. Perhatikanlah contoh berikut ini:

```
>>> raise NameError
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError
```

Dalam contoh ini, kita membangkitkan secara paksa eksepsi bertipe **NameError**, meskipun kita tidak pernah menampilkan variabel yang belum kita definisikan sebelumnya. Kegunaan sesungguhnya dari eksepsi **raise** adalah untuk mengetahui pada bagian mana kita melakukan suatu tipe *error* tertentu, sehingga dapat mempermudah proses *debugging*. Berikut ini contoh lain penggunaan eksepsi **raise**:

```
>>> raise FloatingPointError, "Terjadi floating point error."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
              FloatingPointError: Terjadi floating point error
```

Routine Waktu

Memahami *routine* waktu dalam Python seringkali berguna untuk kegiatan komputasi yang kita lakukan. Pendekatan yang paling sederhana untuk memahami *routine* waktu adalah dengan menggunakan modul **time**, seperti contoh berikut ini:

```
>>> import time
>>> time.time()
```

Fungsi **time()** dalam modul **time** memberitahukan kepada kita waktu dalam satuan sekon yang diukur relatif terhadap tanggal acuan yang dikenal sebagai *epoch*. *Epoch* merupakan tanggal 1 Januari 1970 pukul 12.00 malam menurut sistem UNIX. Anggaplah hal ini sebagai pengetahuan tambahan, karena kerap kali para *programmer* Python tidak begitu perlu untuk mengetahui waktu secara eksak, melainkan yang justru dibutuhkan untuk menjalankan suatu bagian program tertentu. Tentu hal ini ditujukan agar program yang dibuat menjadi lebih efektif dan efisien. Mari kita coba untuk menghitung waktu komputasi pada contoh program **prima.py** yang telah kita buat berikut ini:

prima.py

```
import time
t1 = time.time()
def nextprime(primelist):
    k = max(primelist)+1
    while True:
        FoundDivisor = False
        for prime in primelist:
            if k % prime == 0:
                FoundDivisor = True
                break
        if FoundDivisor:
            k +=1
        else:
            return k
upperlimit = 50
l = [2]
while l[-1] < upperlimit:
    l.append(nextprime(l))
l = l[:-1]
print l
t2 = time.time()
print "Waktu yang dibutuhkan untuk menjalankan program prima.py adalah %.2f detik" %(t2-t1)
```

Setelah dijalankan maka kita dapat mengetahui total waktu komputasi program ini:

```
C:\Users\The Power Of Dream\Anaconda2>python prima.py
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
Waktu yang dibutuhkan untuk menjalankan program prima.py adalah 0.02 detik
1504035470.59
```

Kita juga dapat mengetahui waktu komputasi suatu fungsi dengan menggunakan modul *profiling* seperti **`profile`** dan **`cProfile`**. Pada dasarnya kedua modul ini identik, hanya saja bahasa C mendominasi komponen penulisan modul **`cProfile`**, sehingga membuatnya lebih cepat untuk dieksekusi. Ada baiknya kalian selalu menggunakan modul **`cProfile`**. Berikut ini contoh penggunaan modul **`cProfile`** pada satu fungsi:

prima.py

```
1  import cProfile
2  def nextprime(primelist):
3      k = max(primelist)+1
4      while True:
5          FoundDivisor = False
6          for prime in primelist:
7              if k % prime == 0:
8                  FoundDivisor = True
9                  break
10         if FoundDivisor:
11             k +=1
12         else:
13             return k
14 upperlimit = 50
15 l = [2]
16 while l[-1] < upperlimit:
17     l.append(nextprime(l))
18 l = l[:-1]
19 print l
20 cProfile.run("max()")
```

Hasilnya:

```
C:\Users\The Power Of Dream\Anaconda2>python prima.py
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
         3 function calls in 0.000 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.000    0.000    0.000    0.000 <string>:1(<module>)
        1    0.000    0.000    0.000    0.000 {max}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Prof
iler' objects}


Traceback (most recent call last):
  File "prima.py", line 20, in <module>
    cProfile.run("max()")
  File "C:\Users\The Power Of Dream\Anaconda2\lib\cProfile.py", line 29, in run
    prof = prof.run(statement)
  File "C:\Users\The Power Of Dream\Anaconda2\lib\cProfile.py", line 135, in run

    return self.runctx(cmd, dict, dict)
  File "C:\Users\The Power Of Dream\Anaconda2\lib\cProfile.py", line 140, in run
ctx
    exec cmd in globals, locals
  File "<string>", line 1, in <module>
TypeError: max expected 1 arguments, got 0
```

Perlu kalian ingat, bahwa fungsi yang hendak kita profilkan dalam fungsi `cProfile.run()` harus dituliskan dalam bentuk *string*. Sesudah fungsi `max()` selesai dijalankan, cProfile akan menampilkan statistik berkenaan dengan waktu pada fungsi tersebut.

Kita juga dapat menjalankan **`cProfile`** untuk *profiling* waktu dalam satu *script* secara menyeluruh dengan cara seperti berikut ini:

```
C:\Users\The Power Of Dream\Anaconda2>python -m cProfile prima.py
```

Sesudah kita menjalankannya, maka kita akan menjumpai tampilan seperti ini:

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
         47 function calls in 0.003 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    0.003    0.003    0.003    0.003 prima.py:1(<module>)
       15    0.000    0.000    0.000    0.000 prima.py:1(nextprime)
       15    0.000    0.000    0.000    0.000 {max}
       15    0.000    0.000    0.000    0.000 {method 'append' of 'list' objects
}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Prof
iler' objects}
```

Bagian paling kanan memuat nama – nama modul dan fungsi yang dipanggil oleh program yang kita jalankan. Beberapa di antaranya mungkin nampak tidak familiar bagi kita yang membuat program ini, kasus ini umumnya terjadi karena fungsi dalam suatu modul memanggil fungsi lain dalam modul yang sama atau pada modul yang berbeda. Sementara itu, kolom – kolom pada sebelah kiri menampilkan statistik berkenaan dengan *timing* program tersebut. Berikut detailnya:

- **`ncalls`**: menyatakan berapa kali suatu fungsi dipanggil.
- **`tottime`**: total waktu yang dihabiskan untuk suatu fungsi. Merupakan akumulasi dari seluruh panggilan terhadap fungsi tersebut.
- **`percall`**: waktu rata – rata yang dihabiskan dalam setiap panggilan terhadap suatu fungsi.
- **`cumtime`**: total waktu yang dihabiskan dalam suatu fungsi ditambah waktu yang dihabiskan fungsi – fungsi lain yang dipanggil melalui fungsi tersebut.
- **`percall`**: waktu rata – rata yang dihabiskan dalam setiap panggilan terhadap suatu fungsi fungsi ditambah waktu yang dihabiskan fungsi – fungsi lainnya yang dipanggil melalui fungsi tersebut.
-

Mengenal NumPy dan SciPy

NumPy dan SciPy merupakan modul *add-on* Python yang bersifat *open source*, dan menyediakan *routine* matematik dan numerik umum secara pra-kompilasi, serta beragam fungsi – fungsi yang dapat kita jalankan dengan cepat. Dengan kedua modul ini, ditambah dengan matplotlib membuat

Python menjadi ‘mesin’ komputasi yang setara dengan (bahkan melebihi) piranti lunak berbayar MatLab. Paket NumPy (*Numerical Python*) menyediakan *routine – routine* dasar untuk memanipulasi *array* dalam skala besar dan matriks untuk memproses data numerik. Sementara itu, paket SciPy (*Scientific Python*) berfungsi untuk memperluas kegunaan dari NumPy dengan koleksi algoritma matematika terapan, seperti Transformasi Fourier; regresi; dan masih banyak lagi.

Mengimpor Modul NumPy

Terdapat banyak cara untuk mengimpor modul NumPy. Cara standarnya adalah dengan menggunakan perintah `import` sederhana seperti berikut ini:

```
>>> import numpy
```

Namun, ketika kita bekerja dengan banyak data numerik tentu akan sangat merepotkan ketika setiap kali kita menggunakan fungsi dalam NumPy kita harus menuliskan `numpy.x`. Berikut ini adalah hal yang umum dilakukan oleh para *programmer* Python untuk menyingkat `numpy` menjadi `np`:

```
>>> import numpy as np
```

Dengan cara ini, kita dapat mengakses objek – objek NumPy dengan menggunakan perintah `np.x`. Memungkinkan juga bagi kita untuk mengimpor modul NumPy secara langsung dalam *namespace* tempat kita bekerja, sehingga kita tidak perlu menggunakan notasi titik guna mengaksesnya. Kita dapat mengakses objek – objek di dalam NumPy seperti layaknya objek – objek *built-in* lainnya, berikut ini perintahnya:

```
>>> from numpy import *
```

Namun umumnya cara ini tidak disukai oleh para *programmer* Python, karena meniadakan susunan yang biasa digunakan untuk mengakses suatu modul. Sebagai pengingat untuk kalian, dalam tutorial ini kita akan tetap menggunakan perintah `import numpy as np`.

Array

Salah satu fitur NumPy yang paling bermanfaat untuk komputasi ilmiah adalah *array*. Pada dasarnya *array* sangat mirip dengan *list* dalam Python, hanya saja seluruh elemen yang disimpan

dalam *array* harus dalam tipe yang sama, yang umumnya berupa tipe numerik sepeti **`float`** atau **`int`**. Dengan menggunakan *array* kita dapat melakukan operasi data numerik skala besar dengan cepat dan umumnya jauh lebih efisien dibandingkan kita menggunakan *list*.

Kita dapat membuat *array* dari sebuah *list* seperti berikut ini:

```
>>> import numpy as np
>>> a = np.array([1,2,3,4,5,6], float)
>>> a
array([ 1.,  2.,  3.,  4.,  5.,  6.])
>>> type(a)
<type 'numpy.ndarray'>
```

Terdapat dua argumen dalam fungsi *array* tersebut, yaitu *list* yang hendak dikonversi menjadi *array*, dan tipe setiap elemen dalam *list* tersebut. Kita dapat mengakses, memotong, dan memanipulasi *array* seperti layaknya *list*. Perhatikan contoh berikut ini:

```
>>> a[:2]
array([ 1.,  2.])
>>> a[5]
6.0
>>> a[0] = 8
>>> a
array([ 8.,  2.,  3.,  4.,  5.,  6.])
```

Kita juga dapat mengoperasikan *array* multidimensi sebagaimana kita mengoperasikan matriks. Perhatikan contoh *array* dua dimensi berikut ini:

```
>>> a = np.array([[1,2,3,4,5],[6,7,8,9,10]],float)
>>> a
array([[  1.,   2.,   3.,   4.,   5.],
       [  6.,   7.,   8.,   9.,  10.]])
>>> a[0,0]
1.0
>>> a[0,1]
2.0
```

Kita dapat menjalankan operasi pemotongan (*slicing*) pada *array* multidimensi yang sama seperti yang kita jalankan pada *array* berdimensi tunggal, dengan notasi awal mengindikasikan baris dan notasi selanjutnya mengindikasikan kolom, begitu seterusnya. Sebagai catatan, ketika kalian dapat menggunakan notasi ":" untuk memotong seluruh bagian dalam dimensi tersebut. Perhatikan contoh pemotongan *array* dua dimensi berikut ini:

```
>>> a = np.array([[1,2,3],[4,5,6]], float)
>>> a[1,:]
array([ 4.,  5.,  6.])
>>> a[:, 2]
array([ 3.,  6.])
>>> a[-1:, -2:]
array([[ 5.,  6.]])
```

Untuk mengetahui ukuran suatu *array* kita dapat menjalankan properti **`.shape`** yang menghasilkan ukuran *array* tersebut dalam bentuk *tuple*:

```
>>> a.shape
(2L, 3L)
```

Untuk mengetahui tipe nilai yang disimpan dalam *array*, kita dapat menggunakan properti **`.dtype`**:

```
>>> a.dtype
dtype('float64')
```

`float64` merupakan tipe numerik yang digunakan oleh NumPy untuk menyimpan bilangan riil berpresisi ganda (*double-precision*: 8 bit) yang berfungsi sama seperti tipe **`float`** dalam Python. Ketika kita menggunakan fungsi **`len`** pada suatu *array*, maka jumlah elemen – elemen terluarlah yang akan dihitung sebagai panjang *array* tersebut, berikut contohnya:

```
>>> a = np.array([[1,2,3], [4,5,6],[1,2,4]], float)
>>> len(a)
3
```

Perintah **`in`** dapat kita gunakan untuk memeriksa apakah suatu nilai tertentu terdapat dalam *array*. Perhatikan contoh berikut ini:

```
>>> a = np.array([[1,2,3], [4,5,6],[1,2,4]],float)
>>> 2 in a
True
>>> 0 in a
False
```

Kita dapat me-*reshape array* ke dalam dimensi yang berbeda dengan menggunakan *tuple* yang memuat dimensi *array* yang kita hendaki. Pada contoh berikut ini kita akan mencoba mengubah *array* 1×1 menjadi *array* 5×2:

```
>>> a = np.array(range(12), float)
>>> a
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,
        11.])
>>> a = a.reshape(6,2)
>>> a
array([[  0.,   1.],
       [  2.,   3.],
       [  4.,   5.],
       [  6.,   7.],
       [  8.,   9.],
       [ 10.,  11.]])
>>> a.shape
(6L, 2L)
```

Perlu kalian ketahui, bahwa alih – alih mengubah *array* semula, fungsi **reshape** justru membuat *array* baru yang dimodifikasi dari *array* semula.

Pendekatan *name-binding* yang telah kita pelajari sebelumnya juga berlaku pada *array*. Dengan demikian, kita dapat memanfaatkan fungsi **copy** untuk membuat salinan *array* yang disimpan secara terpisah di dalam memori. Berikut contohnya:

```
>>> a = np.array([1,2,8], float)
>>> b = a
>>> a[-1] = 0
>>> a = np.array([1,2,8], float)
>>> b = a
>>> c = a.copy()
>>> a[-1] = 0
>>> a
array([ 1.,  2.,  0.])
>>> b
array([ 1.,  2.,  0.])
>>> c
array([ 1.,  2.,  8.])
```

Kita juga dapat membuat *list* dari *array*:

```
>>> a = np.array([1,2,8], float)
>>> a
array([ 1.,  2.,  8.])
>>> a.tolist()
[1.0, 2.0, 8.0]
>>> list(a)
[1.0, 2.0, 8.0]
```

Dengan menggunakan fungsi **tostring** kita dapat mengonversikan data mentah dalam sebuah *array* menjadi bentuk data *string* biner. Sebaliknya, kita juga dapat mengonversi bentuk data biner *string* menjadi *array* dengan menggunakan fungsi **fromstring**. *Routine* ini terkadang bermanfaat untuk menyimpan data *array* yang berukuran besar ke dalam bentuk file (untuk menghemat memori) yang dapat dimanfaatkan pada waktu kita membutuhkannya kembali. Berikut ini contohnya:

```
>>> a = np.array([1,2,8], float)
>>> s = a.tostring()
>>> s
'\x00\x00\x00\x00\x00\x00\xf0?\x00\x00\x00\x00\x00\x00\x00@\x00\x00\x00\x00\x00\
x00 @'
>>> np.fromstring(s)
array([ 1.,  2.,  8.])
```

Dengan menggunakan fungsi **`fill`**, kita dapat mengisi *array* dengan data tunggal:

```
>>> a = np.array([1,2,8], float)
>>> a
array([ 1.,  2.,  8.])
>>> a.fill(6)
>>> a

array([ 6.,  6.,  6.])
```

Operasi transpose *array* juga dapat kita lakukan:

```
>>> a = np.array(range(8),float). reshape (2,4)
>>> a
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.]])
>>> a.transpose()
array([[ 0.,  4.],
       [ 1.,  5.],
       [ 2.,  6.],
       [ 3.,  7.]])
```

Array multidimensi dapat kita jadikan *array* berdimensi tunggal dengan menggunakan fungsi **`flatten`**:

```
>>> a = np.array([[1,2,8], [1,2,0]], float)
>>> a
array([[ 1.,  2.,  8.],
       [ 1.,  2.,  0.]])
>>> a.flatten()
array([ 1.,  2.,  8.,  1.,  2.,  0.])
```

Dua atau lebih *array* dapat digabungkan dengan menggunakan fungsi **`concatenate`**:

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([4,5,6], float)
>>> c = np.array([7,8,9,10], float)
>>> np. concatenate(a,b,c)
>>> np. concatenate((a,b,c))
array([  1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.])
```

Jika *array* memiliki lebih dari satu dimensi, memungkinkan juga bagi kita untuk menentukan dimensi (`axis`) mana yang hendak digabungkan. Jika kita tidak menentukan dimensi mana yang hendak digabungkan, maka NumPy akan menggabungkannya dalam dimensi baris (`axis = 0`), maka untuk menggabungkan *array* pada dimensi yang kita inginkan kita perlu menambahkan keterangan `axis`. Perhatikan contoh berikut ini:

```
>>> a = np.array([[1,2,3], [4,5,6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> b = np.array([[7,8,9], [10,11,12]], float)
>>> b
array([[  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
>>> np. concatenate((a,b))
array([[  1.,   2.,   3.],
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
>>> np. concatenate((a,b), axis = 0)
array([[  1.,   2.,   3.],
       [  4.,   5.,   6.],
       [  7.,   8.,   9.],
       [ 10.,  11.,  12.]])
>>> np. concatenate((a,b), axis = 1)
array([[  1.,   2.,   3.,   7.,   8.,   9.],
       [  4.,   5.,   6.,  10.,  11.,  12.]])
```

Dengan menggunakan kontanta `newaxis`, kita juga dapat menambahkan dimensi *array*:

```
>>> a = np.array([1,2,8], float)
>>> a
array([ 1.,  2.,  8.])
>>> a[:, np.newaxis]
array([[ 1.],
       [ 2.],
       [ 8.]])
>>> a[np.newaxis, :]
array([[ 1.,  2.,  8.]])
>>> a[:, np.newaxis].shape
(3L, 1L)
>>> a[np.newaxis, :].shape
(1L, 3L)
```

Pendekatan dengan menggunakan `newaxis` ini akan sangat bermanfaat dalam operasi matematika matriks dan ruang vektor.

Cara – Cara Lain Untuk Membentuk *Array*

Fungsi `arange` (mirip dengan `range`) dapat kita manfaatkan untuk membentuk suatu *array*:

```
>>> np.arange(7, dtype = float)
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.])
>>> np.arange(1,6,2, dtype = int)
array([1, 3, 5])
```

Fungsi **zeros** dan **ones** juga dapat kita manfaatkan untuk membentuk *array* yang memuat nilai 0 dan 1, dengan dimensi yang kita kehendaki:

```
>>> np.ones((2,3), dtype = float)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.zeros(7, dtype = int)
array([0, 0, 0, 0, 0, 0, 0])
```

Dengan memanfaatkan fungsi **ones_like** dan **zeros_like** kita dapat membentuk *array* yang memuat nilai 1 dan 0 yang memiliki ukuran sama dengan *array* yang dijadikan acuan:

```
>>> a = np.array([[1,2,3], [4,5,6]], float)
>>> a
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
>>> np.ones_like(a)
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])
>>> np.zeros_like(a)
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
```

NumPy juga menyediakan beberapa fungsi yang dapat digunakan untuk membentuk *array* khusus, seperti *array* identitas:

```
>>> np.identity(5, dtype = int)
array([[1, 0, 0, 0, 0],
       [0, 1, 0, 0, 0],
       [0, 0, 1, 0, 0],
       [0, 0, 0, 1, 0],
       [0, 0, 0, 0, 1]])
```

Kita juga dapat membuat *array* identitas $n \times n$ dimulai pada kolom ke **k** melalui fungsi **eye**:

```
>>> np.eye(7, k = 2, dtype = float)
array([[ 0.,  0.,  1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  1.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  0.]])
>>> np.eye(7, k = 0, dtype = float)
array([[ 1.,  0.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  0.,  0.,  1.]])
```

Matematika *Array*

Ketika kita menerapkan operasi matematika pada *array*, maka operasi yang dikerjakan adalah operasi berbasis antar elemen yang berbeda dengan operasi matematika pada matriks. Dengan demikian, operasi matematika pada *array* harus memenuhi sifat – sifat operasinya sendiri, seperti kesamaan ukuran *array* pada setiap operasi. Perhatikanlah contoh operasi matematika *array* berikut ini:

```
>>> a = np.array([1,2,3], float)
>>> b = np.array([5,2,6], float)
>>> a + b
array([ 6.,  4.,  9.])
>>> a - b
array([-4.,  0., -3.])
>>> a * b
array([  5.,   4.,  18.])
>>> a % b
array([ 1.,  0.,  3.])
>>> b ** a
array([   5.,    4.,  216.])
>>> a / b
array([ 0.2,  1. ,  0.5])
```

Pada operasi matematika *array* multidimensi juga berlaku operasi antar elemen yang berbeda dengan operasi matematika pada matriks:

```
>>> a = np.array([[1,2],[3,4]], float)
>>> b = np.array([[2,1], [5,3]], float)
>>> a * b
array([[  2.,   2.],
       [ 15.,  12.]])
```

Kita akan mendapati *error* jika ukuran *array* yang kita operasikan tidak tepat:

```
>>> a = np.array([1,2,3], int)
>>> b = np.array([4,5], int)
>>> a - b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together with shapes (3,) (2,)
```

Namun, jika perbedaan antar komponen *array* tersebut hanya pada dimensi baris, Python akan memaksa (*broadcast*) terjadinya operasi matematika.Seringkali dengan cara melakukan pengulangan terhadap *array* yang lebih kecil akan memiliki dimensi yang sama dengan *array* yang lebih besar. Berikut ini contohnya:

```
>>> a = np.array([[1,2], [3,4], [5,6]], float)
>>> b = np.array([-1,3], float)
>>> a
array([[ 1.,  2.],
       [ 3.,  4.],
       [ 5.,  6.]])
>>> b
array([-1.,  3.])
>>> a - b
array([[ 2., -1.],
       [ 4.,  1.],
       [ 6.,  3.]])
>>> a + b
array([[ 0.,  5.],
       [ 2.,  7.],
       [ 4.,  9.]])
```

Array `b` mengalami penambahan dimensi sehingga menjadi seukuran dengan *array* `a`. Sebelum dilakukan operasi penjumlahan dan pengurangan, *array* `b` mengalami pengulangan sehingga menjadi:

```
array([[-1.,  3.],
       [-1.,  3.],
       [-1.,  3.]])
```

Di samping operasi – operasi matematika standar, NumPy menyediakan kita *library* besar yang memuat fungsi – fungsi matematika umum yang dapat diterapkan pada operasi antar elemen di *array*. Fungsi – fungsi tersebut antara lain: `abs`, `sign`, `sqrt`, `log`, `log10`, `exp`, `sin`, `cos`, `tan`, `arcsin`, `arccos`, `arctan`. `sinh`, `cosh`, `tanh`, `arcsinh`, `arccosh`, dan `arctanh`.

```
>>> a = np.array([1,4,9], float)
>>> np.sqrt(a)
array([ 1.,  2.,  3.])
```

Fungsi – fungsi seperti `floor`, `ceil`, dan `rint` dapat kita terapakan pada *array* untuk melakukan pembulatan *integer* ke bawah, ke atas, dan ke nilai terdekat:

```
>>> a = np.array([2.1, 2.5, 2.9], float)
>>> np.floor(a)
array([ 2.,  2.,  2.])
>>> np.ceil(a)
array([ 3.,  3.,  3.])
>>> np.rint(a)
array([ 2.,  2.,  3.])
```

Sebagai informasi tambahan, modul NumPy juga memuat dua konstanta matematika penting berikut ini:

```
>>> np.pi
3.141592653589793
>>> np.e
2.718281828459045
```

Iterasi Dalam *Array*

Kita dapat melakukan proses iterasi dalam *array*, seperti layaknya pada *list*:

```
>>> a = np.array([1,2,8], int)
>>> for x in a:
...     print x
...
1
2
8
```

Hal ini juga berlaku pada *array* multidimensi:

```
>>> a = np.array([[1,2], [3,4], [5,6]], float)
>>> for x in a:
...     print x
...
[ 1.  2.]
[ 3.  4.]
[ 5.  6.]
```

Kita juga dapat menerapkan penugasan berganda dalam proses iterasi *array*:

```
>>> a = np.array([[1,2], [3,4], [5,6]], float)
>>> for (x, y) in a:
...     print x * y
...
2.0
12.0
30.0
```

Operasi Dasar *Array*

Terdapat banyak sekali fungsi yang dapat digunakan untuk mengekstraksi seluruh isi *array*. Misalnya, kita dapat menjulahkan dan mengalikan seluruh elemen dalam suatu *array* dengan menggunakan fungsi – fungsi berikut ini:

```
>>> a = np.array([1,2,8], float)
>>> a.sum()
11.0
>>> a.prod()
16.0
```

Sebagai alternatif, kita juga dapat mengakses fungsi – fungsi tersebut melalui variabel *array*:

```
>>> np.sum(a)
11.0
>>> np.prod(a)
16.0
```

NumPy juga menyediakan *routine – routine* yang dapat kita manfaatkan untuk menghitung kuantitas statistik (rata – rata, varainsi, dan standar deviasi) dalam himpunan data suatu *array*:

```
>>> a = np.array([1,2,8], float)
>>> a.mean()
3.6666666666666665
>>> a.var()
9.5555555555555571
>>> a.std()
3.0912061651652349
```

Tentunya, kita juga dapat mengetahui nilai terendah dan tertinggi dari elemen – elemen dalam suatu *array*:

```
>>> a = np.array([1,2,8], float)
>>> a.min()
1.0
>>> a.max()
8.0
```

Dengan menggunakan fungsi **`argmin`** dan **`argmax`**, kita dapat mengetahui pada indeks mana terdapat nilai terendah dan tertinggi dalam suatu *array*:

```
>>> a = np.array([1,2,8], float)
>>> a.argmin()
0
>>> a.argmax()
2
```

Pada *array* multidimensi, fungsi – fungsi yang telah kita gunakan tadi dapat ditambahkan argumen opsional **`axis`** agar dapat beroperasi hanya pada dimensi tertentu saja, berikut ini contohnya:

```
>>> a = np.array([[0,2], [3,-1], [3,5]], float)
>>> a.mean(axis = 0)
array([ 2.,  2.])
>>> a.mean(axis = 1)
array([ 1.,  1.,  4.])
>>> a.min(axis = 1)
array([ 0., -1.,  3.])
>>> a.max(axis = 0)
array([ 3.,  5.])
```

Sebagaimana *list*, kita juga dapat mengurutkan elemen – elemen dalam *array*:

```
>>> a = np.array([5,1,4,-2,0], float)
>>> sorted(a)
[-2.0, 0.0, 1.0, 4.0, 5.0]
>>> a.sort()
>>> a
array([-2.,  0.,  1.,  4.,  5.])
```

Kita juga dapat memotong *array* sesuai dengan rentang nilai tertentu. Hal ini sama seperti perintah `min(max(x, nilaimin), nilaimaks)` untuh setiap elemen `x` dalam *array*:

```
>>> a = np.array([6,2,5,-1,0], float)
>>> a.clip(0,4)
array([ 4.,  2.,  4.,  0.,  0.])
```

Kita dapat mengekstraksi elemen – elemen unik (tertentu) dalam *array* dengan cara sebagai berikut:

```
>>> a = np.array([1,1,1,2,2,3,4,4,4,4,5,5,5,5,5], float)
>>> np.unique(a)
array([ 1.,  2.,  3.,  4.,  5.])
```

Kita juga dapat mengekstraksi elemen – elemen diagonal dalam suatu *array*:

```
>>> a = np.array([[1,2,3], [4,5,6], [7,8,9]], float)
>>> a.diagonal()
array([ 1.,  5.,  9.])
```

Operator Perbandingan dan Pengujian Nilai

Perbandingan *boolean* dapat kita gunakan dalam perbandingan antar elemen pada posisi yang sama pada dua buah *array* dengan ukuran sama. Hasilnya adalah *array* yang memuat konstanta `True`/`False`:

```
>>> a = np.array([1,2,8], float)
>>> b = np.array([1,2,0], float)
>>> a > b
array([False, False,  True], dtype=bool)
>>> a == b
array([ True,  True, False], dtype=bool)
>>> a <= b
array([ True,  True, False], dtype=bool)
```

Kita juga dapat menyimpan perbandingan *boolean* ini ke dalam *array* tersendiri:

```
>>> c = a > b
>>> c
array([False, False,  True], dtype=bool)
```

Kita juga dapat membandingkan suatu *array* dengan nilai tunggal:

```
>>> a = np.array([1,2,8], float)
>>> a > 2
array([False, False,  True], dtype=bool)
```

Dengan menggunakan operator **`any`** dan **`all`**, kita dapat mengetahui apakah sebagian atau seluruh *array boolean* bernilai **`True`**:

```
>>> c = np.array([True, False, False], bool)
>>> any(c)
True
>>> all(c)
False
```

Operasi *boolean* gabungan dapat kita terapkan pada *array* dengan memanfaatkan fungsi – fungsi khusus seperti **`logical_and`**, **`logical_or`**, dan **`logical_not`**:

```
>>> a = np.array([1,3,0], float)
>>> np.logical_and(a > 0, a < 3)
array([ True, False, False], dtype=bool)
>>> b = np.array([True, False, True], bool)
>>> np.logical_not(b)
array([False,  True, False], dtype=bool)
>>> c = np.array([False, True, False], bool)
>>> np.logical_or(b,c)
array([ True,  True,  True], dtype=bool)
```

Fungsi **`where`** dapat kita gunakan untuk membentuk *array* baru dari dua buah *array* yang berdimensi sama dengan menggunakan filter *boolean* untuk melakukan pemilihan di antara dua elemen tersebut. Sintaks dasarnya adalah **`where(boolarray, truearray, falsearray)`**:

```
>>> a = np.array([2,2,0], float)
>>> np.where(a != 0, 1/a, a)
__main__:1: RuntimeWarning: divide by zero encountered in divide
array([ 0.5,  0.5,  0. ])
```

Kita juga dapat menerapkan *broadcast* dalam fungsi **where**:

```
>>> np.where(a > 0, 3, 2)
array([3, 3, 2])
```

Fungsi **nonzero** dapat kita manfaatkan untuk menghasilkan indeks nilai yang bukan nol dalam suatu *array*. Jumlah *item* dalam *tuple* sama dengan jumlah dimensi pada *array*:

```
>>> a = np.array([[0,1], [1,0]], float)
>>> a.nonzero()
(array([0, 1], dtype=int64), array([1, 0], dtype=int64))
```

Kita juga dapat memeriksa keberadaan nilai NaN (*not a number*) dan bilangan hingga (*finite*) dalam suatu *array*:

```
>>> a = np.array([1, np.NaN, np.Inf], float)
>>> a
array([  1.,  nan,  inf])
>>> np.isnan(a)
array([False,  True, False], dtype=bool)
>>> np.isfinite(a)
array([ True, False, False], dtype=bool)
```

Seleksi Elemen Dan Manipulasi *Array*

Kita telah mempelajari, bahwa elemen –elemen *array*, seperti juga *list*, dapat diseleksi dengan menggunakan notasi "[]". Berbeda dengan *list*, hasil seleksi *array* ini dapat dipergunakan untuk membentuk *array* lainnya. Proses seleksi elemen – elemen *array* ini umumnya membutuhkan *array selector*. *Array boolean* dapat kita gunakan sebagai salah satu *array selector*:

```
>>> a = np.array([[1,2], [0,1], [2,8]], float)
>>> a  >= 2
array([[False,  True],
       [False, False],
       [ True,  True]], dtype=bool)
>>> a[a >= 2]
array([ 2.,  2.,  8.])
```

Perhatikanlah bahwa dengan mengirimkan proses seleksi *boolean* **a >= 6** ke dalam notasi kurung siku **a**, dihasilkan *array* hasil seleksi yang hanya memuat elemen – elemen bernilai **True** pada *array selector*. Kita juga dapat menyimpan *array selector* dalam sebuah variabel:

```
>>> a = np.array([[1,2], [0,1], [2,8]], float)
>>> sel = (a >=2)
>>> a[sel]
array([ 2.,  2.,  8.])
```

Untuk melakukan seleksi yang lebih rumit, kita dapat memanfaatkan operator ekspresi *boolean*:

```
>>> a[np.logical_and(a > 1, a < 9)]
array([ 2.,  2.,  8.])
```

Array yang memuat elemen – elemen bertipe *integer* juga dapat kita gunakan sebagai *array selector*. Elemen – elemen pada *array integer* akan berperan sebagai indeks dari *array* yang diseleksi untuk kemudian dituliskan ulang ke dalam bentukan indeksnya. Perhatikan contoh berikut ini:

```
>>> a = np.array([2,4,6,8], float)
>>> b = np.array([0,0,0,1,3,2,1], int)
>>> a[b]
array([ 2.,  2.,  2.,  4.,  8.,  6.,  4.])
```

Dengan kata lain, kita menggunakan *array* **`b`** untuk menyeleksi elemen ke 0,0,0,1,3,2,1 dari *array* **`a`**. Kita juga dapat menggunakan *list* sebagai *array selector* yang sama seperti contoh di atas:

```
>>> a = np.array([2,4,6,8], float)
>>> a[[0,0,0,1,3,2,1]]
array([ 2.,  2.,  2.,  4.,  8.,  6.,  4.])
```

Untuk menyeleksi *array* berdimensi dua, kita membutuhkan dua buah *array integer* berdimensi tunggal untuk menyeleksi masing – masing sisi. Kemudian, hasil seleksi ini akan dijadikan deret dengan elemen pertama merupakan indeks pada dimensi pertama yang diambil dari elemen pertama dari *array selector* pertama, indeks kedua diambil dari elemen pertama dari *array selector* yang kedua, dan begitu seterusnya. Untuk memahaminya, perhatikanlah contoh berikut ini:

```
>>> a = np.array([[1,4], [9,16]], float)
>>> b = np.array([0,0,1,1,0], int)
>>> c = np.array([0,1,1,1,1], int)
>>> a[b,c]
array([  1.,   4.,  16.,  16.,   4.])
```

Dengan menggunakan fungsi **`take`**, kita juga dapat melakukan seleksi yang sama seperti dengan menggunakan *array selector* yang memuat *integer*:

```
>>> a = np.array([2,4,6,8], float)
>>> b = np.array([0,0,0,1,3,2,1], int)
>>> a.take(b)
array([ 2.,  2.,  2.,  4.,  8.,  6.,  4.])
```

Fungsi **`take`** juga menyediakan argumen dimensional, sehingga dapat diterapkan pada **`axis`** mana yang hendak kita seleksi. Berikut ini contohnya:

```
>>> a = np.array([[0,1], [2,3]], float)
>>> b = np.array([0,0,1], int)
>>> a.take(b, axis = 0)
array([[ 0.,  1.],
       [ 0.,  1.],
       [ 2.,  3.]])
>>> a.take(b, axis = 1)
array([[ 0.,  0.,  1.],
       [ 2.,  2.,  3.]])
```

Kebalikan dari fungsi **take**, fungsi **put** dapat kita manfaatkan untuk menempatkan nilai tertentu dari suatu *array* pada indeks *array* yang hendak kita ubah:

```
>>> a = np.array([0,1,2,3,4,5], float
>>> b = np.array([6,7,8], float)
>>> a.put([0,3], b)
>>> a
array([ 6.,  1.,  2.,  7.,  4.,  5.])
```

Perhatikanlah, bahwa nilai 8 pada *array selector* **b** tidak digunakan karena kita hanya menggunakan dua indeks pada *source array* **[0,3]**. *Source array* yang kita gunakan juga dapat mengalami pengulangan, jika ditempatkan tidak pada ukuran yang sama:

```
>>> a = np.array([0,1,2,3,4,5], float)
>>> a.put([0,3], 5)
>>> a
array([ 5.,  1.,  2.,  5.,  4.,  5.])
```

Operasi Matematika Matriks Dan Vektor

NumPy menyediakan berbagai macam fungsi yang dapat kita gunakan untuk melakukan operasi perkalian matriks dan vektor. Misalnya, untuk melakukan perkalian titik:

```
>>> a = np.array([1,2,0], float)
>>> b = np.array([1,2,8], float)
>>> np.dot(a,b)
5.0
```

Fungsi **dot** juga dapat kita manfaatkan untuk melakukan perkalian matriks secara umum:

```
>>> a = np.array([[0,1], [2,3]], float)
>>> b = np.array([2,3], float)
>>> c = np.array([[1,1], [4,0]], float)
>>> a
array([[ 0.,  1.],
       [ 2.,  3.]])
>>> np.dot(b, a)
array([  6.,  11.])
>>> np.dot(a, b)
array([  3.,  13.])
>>> np.dot(a, c)
array([[  4.,   0.],
       [ 14.,   2.]])
>>> np.dot(b, c)
array([ 14.,   2.])
```

Kita juga dapat melakukan operasi perkalian dalam (*inner product*), perkalian luar (*outer product*), dan perkalian silang (*cross product*) untuk setiap matriks dan vektor dalam NumPy. Perlu kalian ingat bahwa hasil perkalian dalam sama dengan perkalian titik. Untuk memahaminya, perhatikanlah contoh berikut ini:

```
>>> a = np.array([1,4,0], float)
>>> b = np.array([2,2,1], float)
>>> np.outer(a,b)
array([[ 2.,  2.,  1.],
       [ 8.,  8.,  4.],
       [ 0.,  0.,  0.]])
>>> np.inner(a,b)
10.0
>>> np.cross(a,b)
array([ 4., -1., -6.])
```

NumPy juga mempunyai *built-in routines* khusus untuk operasi aljabar linier yang dapat kalian temukan dalam submodul **`linalg`**. Kebanyakan *routine* pada submodul ini berurusan dengan operasi matriks dan inversnya. Dengan menggunakan submodul ini, kita dapat menghitung determinan dari suatu matriks:

```
>>> a = np.array([[1,2,8], [1,2,0], [2,2,0]], float)
>>> a
array([[ 1.,  2.,  8.],
       [ 1.,  2.,  0.],
       [ 2.,  2.,  0.]])
>>> np.linalg.det(a)
-15.999999999999998
```

Kita juga dapat mengetahui nilai dan vektor eigen dari suatu matriks:

```
>>> vals, vecs = np.linalg.eig(a)
>>> vals
array([ 5.4106068 , -3.30528215,  0.89467535])
>>> vecs
array([[-0.87221524, -0.88504352,  0.73247253],
       [-0.25573609,  0.16682308, -0.66267638],
       [-0.41694079,  0.43458949,  0.15602567]])
```

Untuk mengetahui matriks invers `a`, kita dapat melakukan hal berikut ini:

```
>>> b = np.linalg.inv(a)
>>> b
array([[ 0.   , -1.   ,  1.   ],
       [ 0.   ,  1.   , -0.5  ],
       [ 0.125, -0.125,  0.   ]])
```

Terakhir, kita juga dapat melakukan perhitungan *Singular Value Decomposition* pada suatu matriks:

```
>>> a = np.array([[1,2,8], [1,2,0], [2,2,0]], float)
>>> U, s, Vh = np.linalg.svd(a)
>>> U
array([[-0.99086559,  0.1347968 ,  0.00389827],
       [-0.08574503, -0.60745728, -0.78971098],
       [-0.10408248, -0.7828317 ,  0.61346668]])
>>> s
array([ 8.37036016,  3.40920784,  0.56068942])
>>> Vh
array([[-0.15349108, -0.28211286, -0.94702314],
       [-0.59788783, -0.73653015,  0.31631231],
       [ 0.78674686, -0.61476473,  0.05562104]])
```

Persamaan Suku Banyak

NumPy kita kebutuhan yang diperlukan untuk operasi matematika suku banyak (polinomial). Misalnya ketika kita ingin mengetahui polinom mana yang menghasilkan akar – akar sebagai berikut:

```
>>> np.poly([-1, 1, 1, 10])
array([  1., -11.,   9.,  11., -10.])
```

Dalam contoh tersebut *array* yang dihasilkan merupakan koefisien suku banyak. *Array* tersebut merupakan suku banyak $x^4 - 11x^3 + 9x^2 + 11x - 10$.

Sebaliknya, kita juga dapat mengetahui akar – akar suku banyak dengan meng-*input* koefisien pada suku banyak itu:

```
>>> np.roots([1, 4, -2, 3])
array([-4.57974010+0.j        ,  0.28987005+0.75566815j,
        0.28987005-0.75566815j])
```

Dua buah akar dari persamaan $x^3 + 4x^2 - 2x + 3$ ternyata merupakan bilangan kompleks.

Kita juga dapat melakukan integrasi terhadap koefisien suku banyak, dengan nilai konstanta *default* –nya bernilai nol. Perhatikanlah contoh berikut ini, di mana kita akan mengintegrasikan persamaan $x^3 + x^2 + x + 1$, sehingga menghasilkan persamaan $x^4/4 + x^3/3 + x^2/2 + x + C$:

```
>>> np.polyint([1, 1, 1, 1])
array([ 0.25      ,  0.33333333,  0.5       ,  1.        ,  0.        ])
```

Sebaliknya, kita juga dapat melakukan operasi turunan:

```
>>> np.polyder([1./4., 1./3., 1./2., 1., 0.])
array([ 1.,  1.,  1.,  1.])
```

Dengan menggunakan fungsi **`polyval`**, kita dapat memeriksa nilai persamaan suku banyak pada titik tertentu. Misalnya, kita ingin memeriksa nilai persamaan $x^3 + x^2 + x + 1$ pada titik $x = 81$:

```
>>> np.polyval([1, 1, 1, 1], 81)
538084
```

Kita juga dapat mencocokan dua buah kumpulan data ke dalam fungsi suku banyak pada orde yang kita inginkan:

```
>>> x = [1, 2, 3, 4, 5]
>>> y = [4, 15, 40, 85, 156]
>>> np.polyfit(x, y, 3)
array([ 1.,  1.,  1.,  1.])
```

Hasilnya merupakan koefisien persamaan suku banyak. Untuk menerapkan metode interpolasi yang lebih canggih, kita harus menggunakan modul SciPy.

Operasi Statistik

Di samping fungsi – fungsi **`mean`**, **`var`**, dan **`std`**, NumPy juga menyediakan beberapa fungsi lain yang dapat dimanfaatkan untuk operasi statistik pada *array*. Seperti fungsi **`median`** berikut ini:

```
>>> a = np.array([1,5,7,12,81,12,1,5,70,2,171,220], float)
>>> np.median(a)
9.5
```

Kita dapat menghitung koefisien korelasi antar variabel yang berbeda dengan memanfaatkan *array* yang berbentuk **`[[x1, x2, ...], [y1, y2, ...], [z1, z2, ...], ...]`**, di mana variabel **`x`**; **`y`**; **`z`**

merupakan data yang berbeda, dan angka yang tertera di sana mengindikasikan kesamaan ciri pengamatan tertentu. Perhatikanlah contoh berikut ini:

```
>>> a = np.array([[1, 2, 3, 4, 5], [4, 15, 40, 85, 156]], float)
>>> c = np.corrcoef(a)
>>> c
array([[ 1.        ,  0.95297837],
       [ 0.95297837,  1.        ]])
```

Dalam contoh tersebut, *array* `c[i, j]` merupakan koefisien korelasi antar pengamatan data ke – `i` dan `j`. Kita juga dapat menghitung nilai kovariansi dari data tersebut:

```
>>> np.cov(a)
array([[  2.50000000e+00,   9.35000000e+01],
       [  9.35000000e+01,   3.85050000e+03]])
```

Bilangan Acak

Salah satu hal terpenting dalam bahasa komputasi ilmiah adalah kemampuannya untuk menghasilkan bilangan acak. Untuk keperluan tersebut, NumPy telah menyediakan *built-in pseudonumber generator routines* dalam submodul `random`. Bilangan acak yang dihasilkan merupakan bilangan acak semua karena dihasilkan melalui suatu bilangan *seed*, namun distribusi yang dilakukan secara statistik mirip dengan pemrosesan bilangan acak yang sesungguhnya. Guna menghasilkan bilangan acak, NumPy menggunakan algoritma yang dikenal sebagai Mesenne Twister.

Kita dapat melakukan pengaturan bilangan *seed* acak sebagai berikut ini:

```
>>> np.random.seed(12014081)
```

Bilangan *seed* selau bertipe *integer*. Setiap program yang diawali dengan *seed* yang sama akan menghasilkan sikuen bilangan acak yang persis sama ketika dijalankan. Hal ini mungkin bermanfaat untuk keperluan *debugging*, namun seringkali kita tidak begitu membutuhkannya.

Kita dapat mengeluarkan *array* bilangan acak dalam interval [0.0, 1.0) dengan perintah berikut ini:

```
>>> np.random.rand(15)
array([ 0.07964548,  0.50524609,  0.0652865 ,  0.42812233,  0.09653092,
        0.12715997,  0.59674531,  0.226012  ,  0.10694568,  0.22030621,
        0.34982629,  0.46778748,  0.20174323,  0.64040673,  0.48306984])
```

Dalam komputasi ilmiah, fungsi **rand** biasanya digunakan untuk menghasilkan *array* multidimensional yang memuat bilangan acak seperti berikut ini:

```
>>> np.random.rand(2,3)
array([[ 0.50523672,  0.38689265,  0.79363745],
       [ 0.58000418,  0.1622986 ,  0.70075235]])
```

Berikut ini contoh untuk menghasilkan bilangan acak tunggal dalam rentang [0.0, 1.0):

```
>>> np.random.random()
0.9645510800892552
```

Untuk menghasilkan bilangan *integer* acak dalam rentang tertentu [min, maks), kita dapat memanfaatkan fungsi **randint(min, maks)**:

```
>>> np.random.randint(1,7)
4
```

Pada contoh – contoh sebelumnya, kita menghasilkan bilangan acak dengan menggunakan distribusi seragam. Di samping itu, NumPy juga menyediakan *generator* bilangan acak dengan menggunakan distribusi – distribusi lain, seperti beta; binomial; khi-kuadrat; Dirichlet; eksponensial; F; gamma; geometri; Gumbel; hipergeometri; Laplace; logistik; log.normal; multinomial; peubah banyak; binomial negatif; khi-kuadrat tidak terpusat; F tidak terpusat; normal; pareto; *power*; Rayleigh; Cauchy; t Student; segitiga; von Mises; Wald; Weibull; dan Zipf. Berikut ini bilangan acak yang dihasilkan dari distribusi Poisson dengan $\lambda = 6.0$:

```
>>> np.random.poisson(6.0)
6
```

Bilangan acak yang dihasilkan melalui distribusi normal dengan $\mu = 1.5$ dan $\sigma = 4.0$:

```
>>> np.random.normal(1.5, 4.0)
4.935565772092483
```

Untuk menghasilkan bilangan acak dengan menggunakan distribusi normal standar ($\mu = 0$, $\sigma = 0$), kita tinggal meniadakan argumen pada fungsi di atas:

```
>>> np.random.normal()
-0.6504372999719978
```

Untuk menghasilkan bilangan acak berjumlah banyak dengan menggunakan distribusi normal, kita dapat menambahkan argumen opsional **size**:

```
>>> np.random.normal(size = 13)
array([-0.16924147,  0.00482963,  0.70737783,  1.05877489,  1.03570718,
        0.57170527, -0.0235708 , -1.80572522, -0.7541702 ,  0.15239462,
        1.8466415 ,  0.13901666,  0.99653418])
```

Kita juga dapat menggunakan submodul `random` untuk mengacak urutan elemen dalam suatu *list*:

```
>>> l = range(7)
>>> l
[0, 1, 2, 3, 4, 5, 6]
>>> np.random.shuffle(l)
>>> l
[0, 1, 6, 5, 4, 3, 2]
```

Perhatikanlah, bahwa fungsi `shuffle` memodifikasi *list* dalam tempat yang sama pada memori, alih – alih menyimpannya dalam lokasi baru di memori.

Fungsi – Fungsi NumPy Lain Yang Perlu DiKetahui

NumPy memuat berbagai fungsi *built-in* lain yang mungkin tidak dapat kita bahas satu per satu di sini. Kegunaan fungsi – fungsi tersebut antara lain adalah untuk melakukan transformasi Fourier diskrit; fungsi – fungsi untuk melakukan operasi aljabar linier yang lebih kompleks; untuk menguji ukuran dan/atau jenis *array* tertentu; memisahkan dan menggabungkan *array*; membuat histogram; memberikan nomor pada *array* dengan berbagai macam cara; membuat dan mengevaluasi fungsi pada *grid* tertentu dalam *array*; bekerja dengan *array* yang mempunyai nilai – nilai khusus (`NaN` dan `Inf`); operasi himpunan; membuat matriks – matriks khusus; dan terakhir untuk bekerja dengan fungsi – fungsi khusus dalam matematika (misalnya, fungsi Bessel). Kalian dianjurkan untuk membaca dokumentasi resmi tentang NumPy di http://docs.scipy.org/doc/ untuk mempelajarinya secara lebih mendalam.

Modul SciPy

SciPy merupakan modul yang memperluas kebermanfaatan *routine - routine* dalam NumPy. Tutorial ini tidak akan membahas SciPy secara mendetail, namun kita akan melihat kapabilitas yang dimiliki SciPy dalam menangani permasalahan komputasi ilmiah. Kita dapat memulai penggunaan SciPy dengan menuliskan perintah sebagai berikut:

```
>>> import scipy
```

Kemudian kita dapat memanfaatkan fungsi **`help`** untuk mengetahui apa saja yang dapat kita lakukan dengan menggunakan SciPy:

```
>>> help(scipy)
Help on package scipy:

NAME
    scipy

FILE
    c:\users\the power of dream\anaconda2\lib\site-packages\scipy\__init__.py

DESCRIPTION
    SciPy: A scientific computing package for Python
    ================================================

    Documentation is available in the docstrings and
    online at https://docs.scipy.org.

    Contents
    --------
    SciPy imports all the functions from the NumPy namespace, and in
    addition provides:

    Subpackages
    -----------
    Using any of these subpackages requires an explicit import.  For example,
    ``import scipy.cluster``.

    ::

     cluster                      --- Vector Quantization / Kmeans
     fftpack                      --- Discrete Fourier Transform algorithms
     integrate                    --- Integration routines
     interpolate                  --- Interpolation Tools
     io                           --- Data input and output
     linalg                       --- Linear algebra routines
     linalg.blas                  --- Wrappers to BLAS library
     linalg.lapack                --- Wrappers to LAPACK library
     misc                         --- Various utilities that don't have
                                      another home.
     ndimage                      --- n-dimensional image package
     odr                          --- Orthogonal Distance Regression
     optimize                     --- Optimization Tools
     signal                       --- Signal Processing Tools
     sparse                       --- Sparse Matrices
     sparse.linalg                --- Sparse Linear Algebra
     sparse.linalg.dsolve         --- Linear Solvers
     sparse.linalg.dsolve.umfpack --- :Interface to the UMFPACK library:
                                      Conjugate Gradient Method (LOBPCG)
     sparse.linalg.eigen          --- Sparse Eigenvalue Solvers
     sparse.linalg.eigen.lobpcg   --- Locally Optimal Block Preconditioned
                                      Conjugate Gradient Method (LOBPCG)
     spatial                      --- Spatial data structures and algorithms
     special                      --- Special functions
     stats                        --- Statistical Functions

-- More  --
```

Sebagaimana dijelaskan pada keterangan di atas, ketika kita hendak mengakses submodul tertentu dalam SciPy, maka kita perlu melakukan perintah **`import`** secara eksplisit:

```
>>> import scipy
>>> import scipy.interpolate
```

Fungsi – fungsi yang terdapat pada setiap submodul didokumentasikan baik dalam *docstring* internal dan pada *website* dokumentasi resmi tentang SciPy. Dengan menggunakan SciPy, kita dapat menghemat waktu pengerjaan komputasi ilmiah karena SciPy memuat banyak sekali *library* yang memuat *routine – routine* yang bersifat *pre-written* dan *pre-tested*.

Tutorial ini tidak akan menjelaskan penggunaan SciPy secara detail, namun tabel berikut ini akan menjelaskan kegunaan masing – masing submodul dalam SciPy:

Submodul	Kegunaan
`scipy.constants`	Untuk membantu pekerjaan yang berhubungan dengan konstanta fisika dan matematika.
`scipy.special`	Fungsi – fungsi khusus dalam fisika matematika, seperti fungsi Airy; elips; Bessel; beta; hipergeometri; gelombang spheroid; Sturve; dan Kelvin.
`scipy.integrate`	Fungsi – fungsi yang dapat digunakan untuk keperluan integrasi numerik, seperti metode trapesium; Simpson; Romberg; dan berbagai metode lainnya. Submodul ini juga menyedikan metode – metode yang dapat digunakan untuk melakukan integrasi pada persamaan diferensial biasa.
`scipy.optimize`	*Minimization/maximization routines* standar yang dioperasikan pada *generic user-defined objective functions*. Algoritma – algoritma yang disediakan oleh submodul ini, antara lain adalah Nelder-Mead Simplex; Powell's; gradien konjugasi; BFGS; *least-squares*; *constrained optimizers*; metode Brent; metode Newton; metode *bisection*; Broden; Anderson; dan *line search*.
`scipy.linalg`	Menyediakan *routine – routine* aljabar linier yang lebih banyak jika dibandingkan modul NumPy. Submodul ini menawarkan kontrol yang lebih bagi pengguna dalam menggunakan *routine* khusus yang lebih cepat untuk keperluan komputasi spesifik (misalnya untuk matriks tridiagonal). Metode – metode yang terdapat di dalamnya antara lain adalah invers matriks; determinan; penyelesaian sistem persamaan linier; *computing*

	norms dan *pseudo/generalized inverses*; nilai eigen/dekomposisi vektor eigen; *singular value decomposition*; dekomposisi LU; dekomposisi Cholesky; dekomposisi QR; dekomposisi Schur; dan berbagai operator matematika matriks lainnya.
`scipy.sparse`	*Routine - routine* yang dibutuhkan untuk menangani matriks berukuran besar yang tersebar (*sparse*).
`scipy.interpolate`	*Routine* dan kelas objek interpolasi yang digunakan untuk data numerik yang bersifat diskrit.
`scipy.fftpack`	*Routine – routine* yang dibutuhkan untuk memproses *Fast Fourier Transform* (FFT).
`scipy.signal`	*Routine – routine* yang dibutuhkan untuk memproses sinyal, seperti konvolusi; korelasi; transformasi fourier berhingga; *B-spline smoothing*; *filtering*; dll.
`scipy.state`	*Library* yang memuat berbagai macam distribusi statistik dan fungsi statistik yang dapat diterapkan pada himpunan data tertentu.

Komunitas *scientific developer* akan senantiasa merawat dan menambahkan hal – hal baru dalam SciPy. Ketika kalian hendak menuliskan program numerik dalam Python, hal pertama yang harus kalian lakukan adalah memeriksa dokumentasi SciPy, karena jika *routine* numerik yang hendak kalian gunakan merupakan hal yang bersifat umum, maka biasanya SciPy sudah menyediakannya.

Mengenal matplotlib

matplotlib merupakan *library* grafis yang umum digunakan dalam Python untuk memproses gambar – gambar ilmiah, baik secara dua dimensi maupun tiga dimensi. Pada tutorial ini kita akan memanfaatkan salah satu submodul dalam matplotlib, yaitu Pylab yang mampu menggabungkan kegunaan pyplot (untuk melakukan *plotting*) dan NumPy (untuk perhitungan matematis pada *array*) dalam *namespace* tunggal, sehingga membuatnya nampak seperti lingkungan kerja pada Matlab.

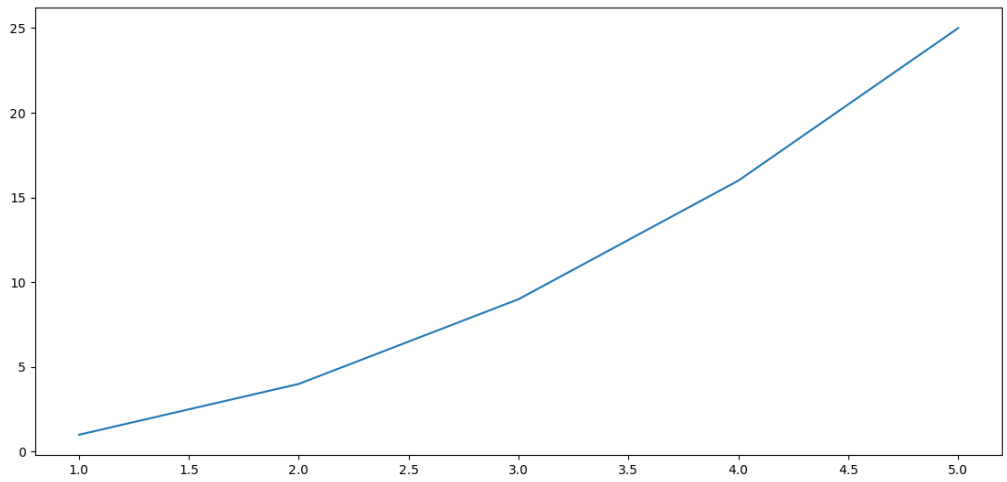
Plotting Dasar

Dalam komputasi ilmiah terdapat dua jenis *plotting* dasar yang wajib kita kuasai karena sangat sering digunakan. Ketiga jenis *plotting* dasar itu antara lain *plot* garis (x, y); dan *scatter plot*.

Pertama kali kita akan membahas *plot* grafik yang paling sering digunakan yaitu sebuah garis yang menghubungkan suatu nilai pada sumbu x dengan suatu nilai pada sumbu y. Perhatikanlah contoh berikut ini:

```
>>> import numpy as np
>>> import matplotlib.pylab as pl
>>> x = np.array([1,2,3,4,5], float)
>>> y = np.array([1,4,9,16,25], float)
>>> import numpy as np
>>> import matplotlib.pylab as pl
>>> x = np.array([1,2,3,4,5], float) # membuat nilai array sumbu x
>>> y = np.array([1,4,9,16,25], float) # membuat nilai array sumbu y
>>> pl.plot(x,y) # menggunakan pylab untuk memplot x dan y
[<matplotlib.lines.Line2D object at 0x0000000006064CC0>]
>>> pl. show() # menampilkan hasil plot pada layar
```

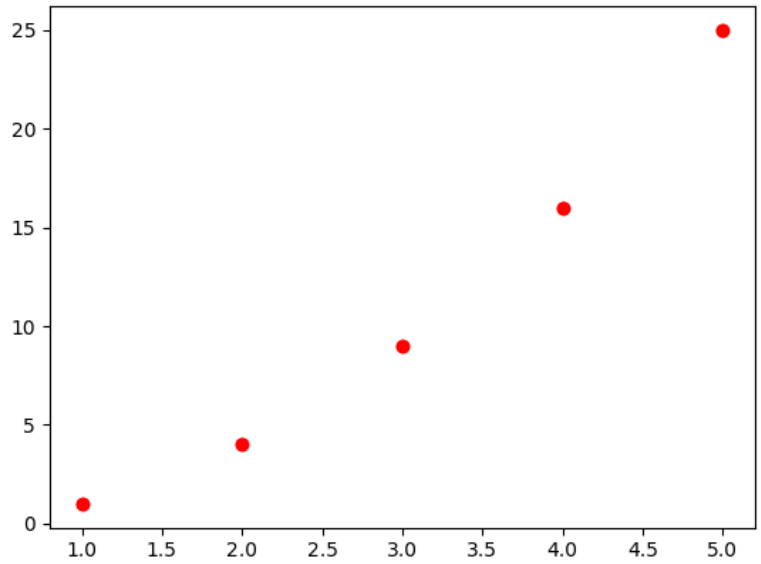
Berikut ini hasil *plot* yang ditampilkan pada layar:



Sebagai alternatif lain untuk menampilkan grafik hubungan antara nilai x dan y, biasanya kita akan menggunakan titik – titik (baik berupa bintang maupun bulatan) yang dalam komputasi ilmiah dikenal sebagai *scatter plot*. Berikut ini contohnya:

```
>>> x = np.array([1,2,3,4,5], float)
>>> y = np.array([1,4,9,16,25], float)
>>> # menggunakan pylab untuk memplot x dan y menjadi titik lingkaran merah
>>> pl.plot(x,y, 'ro')
[<matplotlib.lines.Line2D object at 0x0000000001E25F28>]
>>> pl.show()
```

Berikut ini hasil yang ditampilkan pada layar:



Pengaturan *Plot*

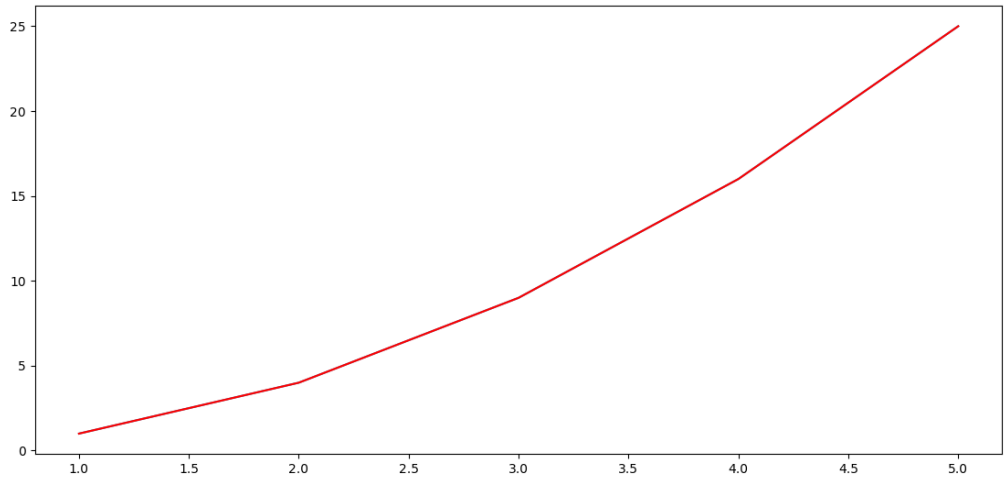
Terkadang dalam komputasi ilmiah kita butuh untuk mem*plot* lebih dari satu himpunan data dalam satu bidang, dan untuk membedakan *plot* tersebut umumnya kita membedakan penanda seperti garis, warna, dll. Untuk melakukan perubahan warna pada plot garis (x, y) yang telah kita buat, misalnya, kita dapat menambakan argumen opsional pada perintah:

```
>>> pl.plot(x,y)
```

Menjadi:

```
>>> pl.plot(x,y,'r')
```

Sehingga *plot* garis yang dihasilkan akan berubah menjadi berwarna merah:

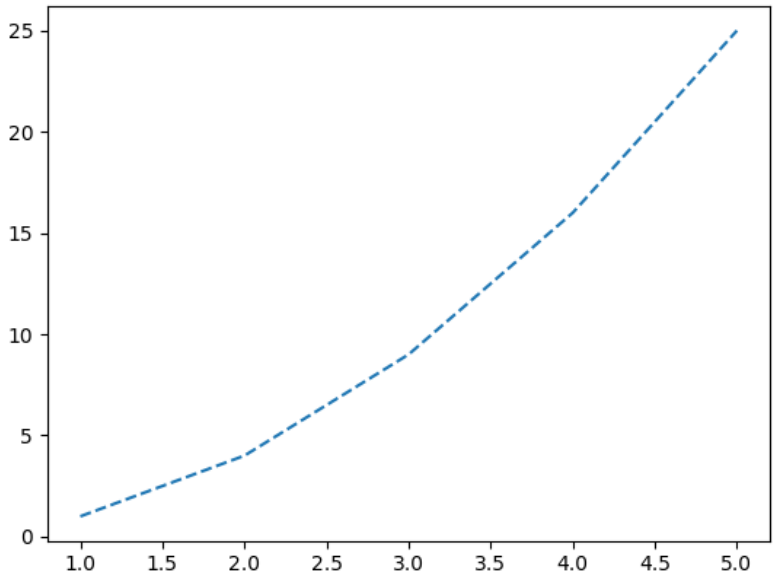


Berikut ini daftar memuat argumen opsional warna:

Karakter	Warna
`r`	Merah
`g`	Hijau
`b`	Biru
`c`	Sian
`m`	Magenta
`y`	Kuning
`k`	Hitam
`w`	Putih

Kita juga dapat melakukan perubahan terhadap *style* garis, misalnya agar menjadi berbentuk titik – titik, garis putus – putus, dll dengan menambahkan argumen berikut ini pada perintah:

```
>>> pl.plot(x,y, '--')
```

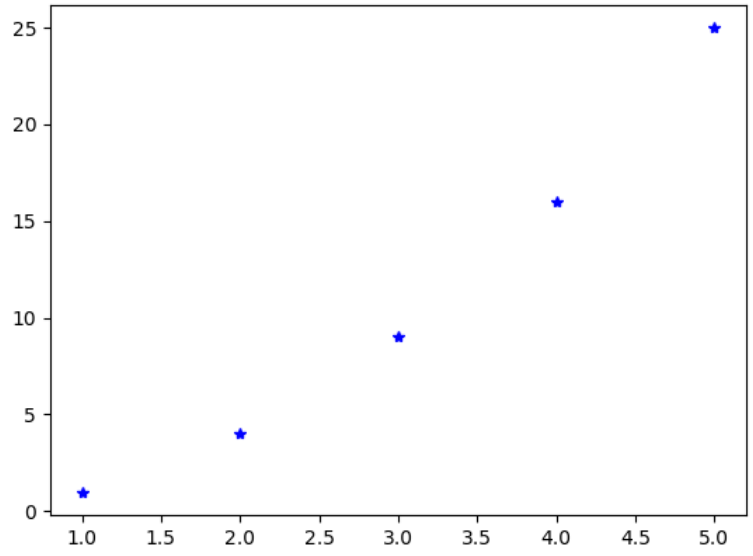


Sehingga *style* garis yang ditampilkan akan berbentuk garis putus – putus. Untuk memperoleh daftar argumen yang digunakan untuk mengubah *style* garis lainnya, kalian dapat mengunjungi dokumentasi resmi matplotlib di http://matplotlib.sourceforge.net/api/pyplot_api.html#matplotlib..pyplot.plot.

Terakhir, jika kalian ingin mengubah *marker style* pada hasil *plot* kalian, kalian dapat mencoba perintah berikut ini:

```
>>> pl.plot(x,y,'b*')
```

Maka hasilnya adalah bintang – bintang berwarna biru berikut ini:



Berikut ini daftar *marker style* yang dapat kalian gunakan:

Karakter	Penanda
`s`	Kotak
`p`	Segi Lima
`*`	Bintang
`h`	Segi Enam 1
`H`	Segi Enam 2
`+`	Tambah
`x`	Huruf x
`D`	*Diamond*
`d`	*Diamond* tipis

Dalam banyak kasus, kita diharuskan untuk memberi keterangan pada sumbu tertentu. Berikut ini perintah yang dapat kalian jalankan ketika hendak memberi keterangan pada grafik *plot* (x, y):

```
pl.xlabel("tuliskan keterangan sumbu x")
pl.ylabel("tuliskan keterangan sumbu y")
```

Untuk menuliskan judul utama pada grafik yang kita buat, lakukan perintah berikut ini:

```
pl.title("tuliskan judul grafik")
```

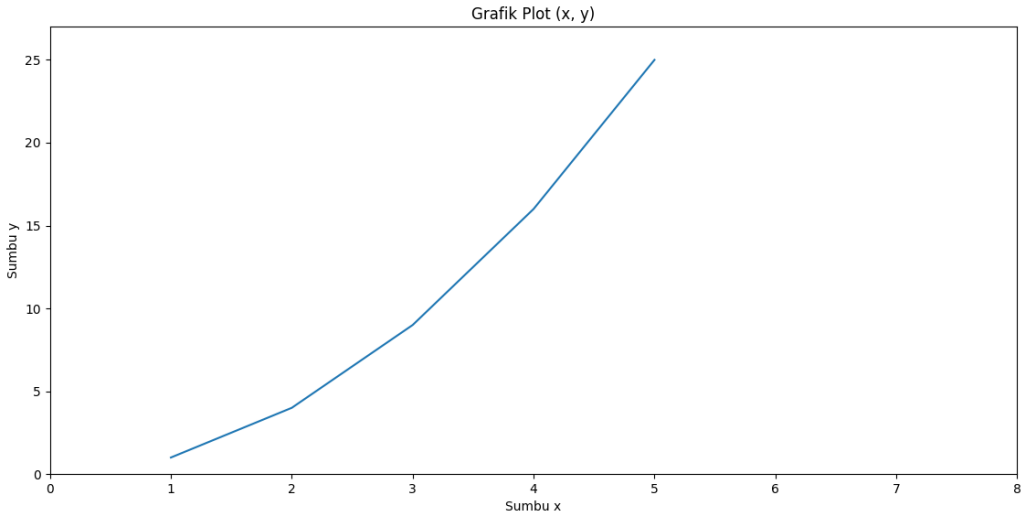
Kalian juga dapat melakukan pengaturan rentang nilai x dan y dengan menggunakan perintah ini:

```
pl.xlim(x_min, x_maks)
pl.ylim(y_min, y_maks)
```

Berikut ini contoh penggunaan ketiga pengaturan tersebut:

```
>>> x = np.array([1,2,3,4,5], float)
>>> y = np.array([1,4,9,16,25], float)
>>> pl.plot(x, y)
[<matplotlib.lines.Line2D object at 0x0000000004E7B94
>>> # memberikan judul grafik
...
>>> pl.title("Grafik Plot (x, y)")
<matplotlib.text.Text object at 0x0000000004B50A58>
>>> # memberikan keterangan pada masing masing sumbu
...
>>> pl.xlabel("Sumbu x")
<matplotlib.text.Text object at 0x0000000004CE0198>
>>> pl.ylabel("Sumbu y")
<matplotlib.text.Text object at 0x0000000004AEAE48>
>>> # mengatur rentang grafik
...
>>> pl.xlim(0.0, 8.0)
(0.0, 8.0)
>>> pl.ylim(0.0, 27.0)
(0.0, 27.0)
>>> pl.show()
```

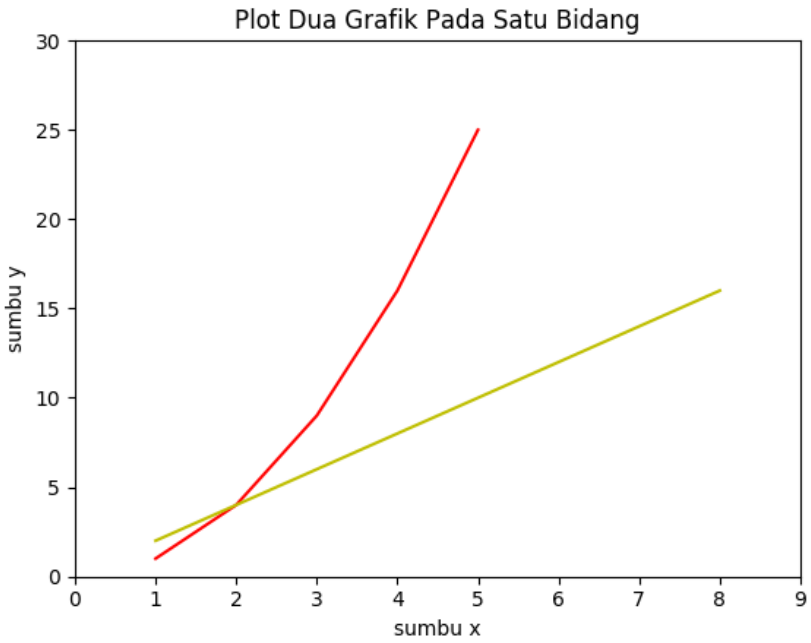
Hasilnya:



Ketika kita hendak menambahkan lebih dari satu *plot* dalam sebuah bidang, Pylab memberikan kemudahan karena tinggal hanya mendefinisikan *array* `x` dan `y` yang berbeda untuk masing – masing *plot* nya:

```
>>> # mendefinisikan array untuk masing masing grafik
...
>>> x1 = np.array([1,2,3,4,5], float)
>>> y1 = np.array([1,4,9,16,25], float)
>>> x2 = np.array([1,2,4,6,8], float)
>>> y2 = np.array([2,4,8,12,16], float)
>>> # menggunakan pylab untuk memplot kedua grafik pada bidang yang sama
...
>>> pl.plot(x1,y1, 'r')
[<matplotlib.lines.Line2D object at 0x0000000005076320>]
>>> pl.plot(x2,y2, 'y')
[<matplotlib.lines.Line2D object at 0x0000000004D17EF0>]
>>> pl.title("Plot Dua Grafik Pada Satu Bidang")
<matplotlib.text.Text object at 0x0000000004F466A0>
>>> pl.xlabel("sumbu x")
<matplotlib.text.Text object at 0x0000000004D45358>
>>> pl.ylabel("sumbu y")
<matplotlib.text.Text object at 0x0000000004D20A58>
>>> pl.xlim(0.0, 9.0)
(0.0, 9.0)
>>> pl.ylim(0.0, 30.0)
(0.0, 30.0)
>>> pl.show()
```

Hasilnya adalah:

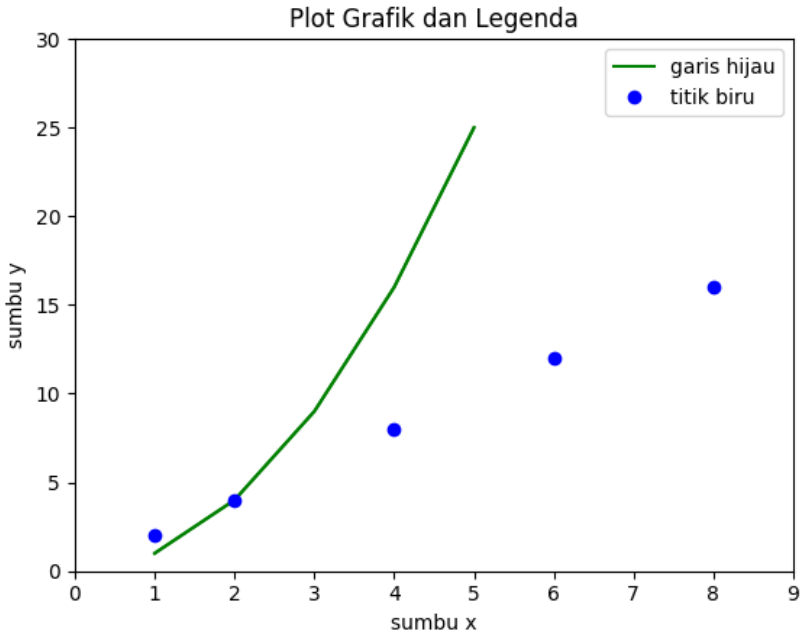


Untuk menambahkan legenda pada grafik dalam Pylab kita cukup memanggil fungsi **`legend`**. Terdapat beberapa parameter dalam pendefinisian legenda yang harus kalian pahami. Parameter pertama yang menjadi *input*-an adalah grafik yang ingin kita berikan keterangan. Parameter kedua merupakan keterangan apa yang hendak kita tuliskan. Parameter ketiga merupakan pilihan pada

lokasi mana kita hendak menempatkan legenda tersebut. **best** bermakna bahwa matplotlib akan menempatkan legenda tersebut pada posisi terbaik yang tidak mengganggu grafik. Selain **best**, pilihan lainnya antara lain adalah **upper right**, **upper left**, **center**, **lower left**, dan **lower right**. Berikut ini contoh pembuatan legenda pada grafik:

```
>>> x1 = np.array([1,2,3,4,5], float)
>>> y1 = np.array([1,4,9,16,25], float)
>>> x2 = np.array([1,2,4,6,8], float)
>>> y2 = np.array([2,4,8,12,16], float)
>>> plot1 = pl.plot(x1,y1, 'g', label = 'garis hijau')
>>> plot2 = pl.plot(x2,y2, 'bo', label = 'titik biru')
>>> pl.title("Plot Grafik dan Legenda")
<matplotlib.text.Text object at 0x000000000B5EF9B0>
>>> pl.xlabel("sumbu x")
<matplotlib.text.Text object at 0x000000000B4E0400>
>>> pl.ylabel("sumbu y")
<matplotlib.text.Text object at 0x000000000521D9E8>
>>> pl.xlim(0., 9.)
(0.0, 9.0)
>>> pl.ylim(0., 30.)
(0.0, 30.0)
>>> pl.legend(loc = 'best')
<matplotlib.legend.Legend object at 0x000000000B686F28>
>>> pl.show()
```

Hasilnya adalah:



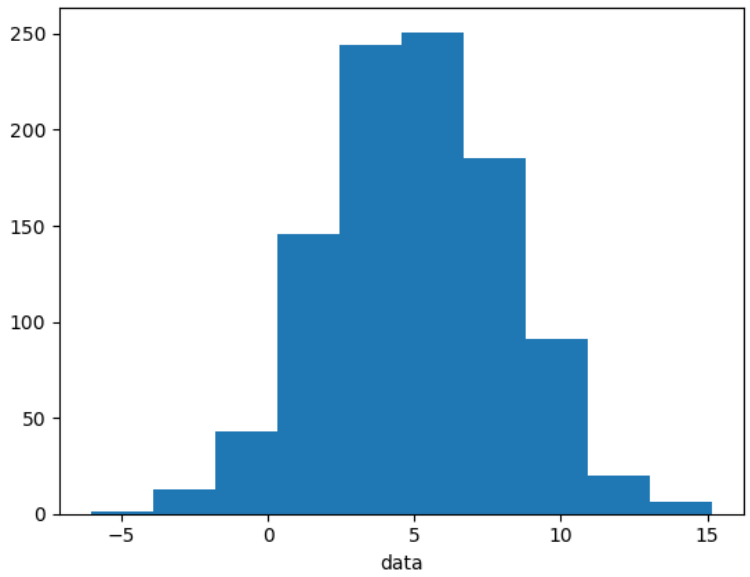
Plotting Histogram

Histogram merupakan salah satu grafik distribusi yang paling banyak digunakan dalam statistika. Di dalam matplotlib, kalian dapat mengakses histogram secara mudah dengan cukup memanggil fungsi `hist`. Perhatikanlah contoh berikut ini:

```
>>> pl.show()
>>> # mendefinisikan array yang memuat bilangan acak, dengan:
...
>>> # rata rata = 5.0
...
>>> # rms = 3.0
...
>>> # jumlah titik = 1000
...
>>> data = np.random.normal(5., 3., 1000)
>>> # membuat histogram dari data array
...
>>> pl.hist(data)
(array([   1.,   13.,   43.,  146.,  244.,  251.,  185.,   91.,   20.,    6.]),
array([ -6.05563989,  -3.93179046,  -1.80794103,   0.3159084 ,
         2.43975783,   4.56360726,   6.68745669,   8.81130612,
        10.93515555,  13.05900498,  15.18285441]), <a list of 10 Patch objects>)

>>> pl.xlabel("data")
<matplotlib.text.Text object at 0x000000000B6969B0>
>>> pl.show()
```

Hasilnya adalah:

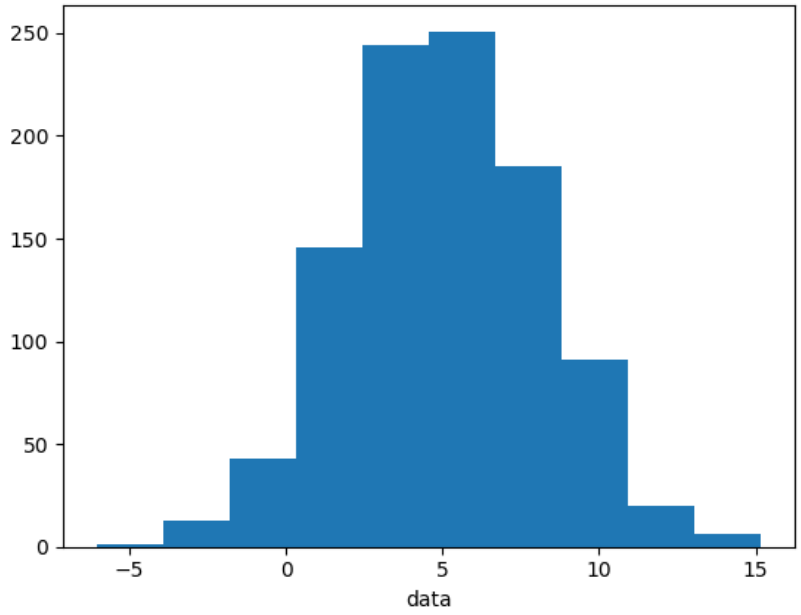


Kita dapat mengatur lebar *bins* histogram kita sendiri dengan melakukan perintah berikut ini:

```
>>> bins = np.arange(-5., 18., 1.)
>>> pl.hist(data, histtype = 'bar')
(array([   1.,   13.,   43.,  146.,  244.,  251.,  185.,   91.,   20.,    6.]),
array([ -6.05563989,  -3.93179046,  -1.80794103,   0.3159084 ,
         2.43975783,   4.56360726,   6.68745669,   8.81130612,
        10.93515555,  13.05900498,  15.18285441]), <a list of 10 Patch objects>

>>> pl.show()
```

Berikut ini hasilnya:

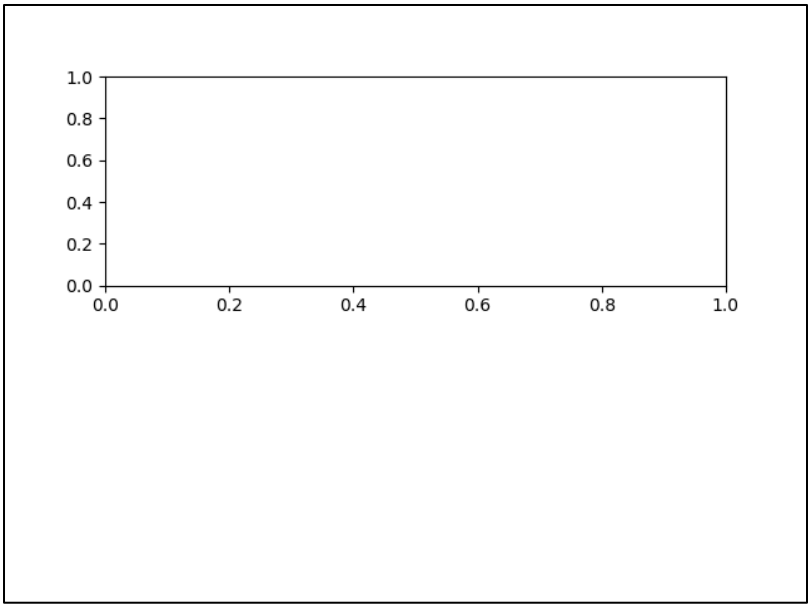


Multiple Plotting Dalam Sebuah Kanvas

matplotlib dapat digunakan untuk membuat lebih dari satu *plot* dalam sebuah kanvas. Hal pertama yang harus kalian lakukan adalah membuat sebuah gambar dan memberikan keterangan mengenai *subplot* di dalamnya:

```
>>> fig1 = pl.figure(1)
>>> pl.subplot(211)
<matplotlib.axes._subplots.AxesSubplot object at 0x0000000005229320>
```

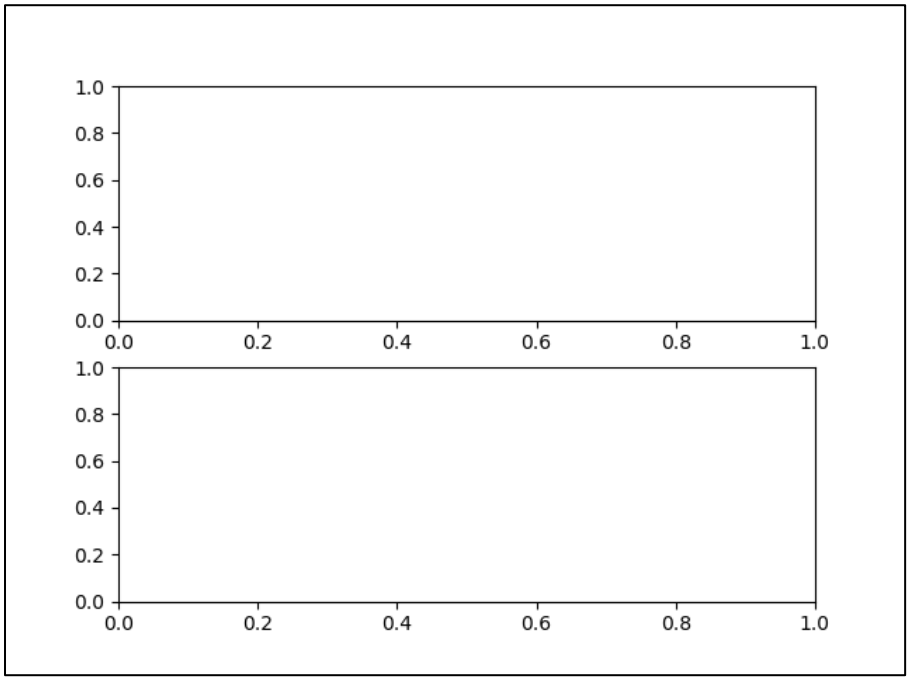
Subplot(211) bermakna bahwa kita akan membuat gambar yang mempunyai dua buah baris dan satu buah kolom, serta ditempatkan pada bagian atas kanvas seperti berikut ini:



Jika kita hendak menempatkan gambar dengan ukuran sama di bagian bawahnya, kita tinggal menambahkan perintah:

```
>>> pl.subplot(212)
<matplotlib.axes._subplots.AxesSubplot object at 0x0000000004AB60F0>
```

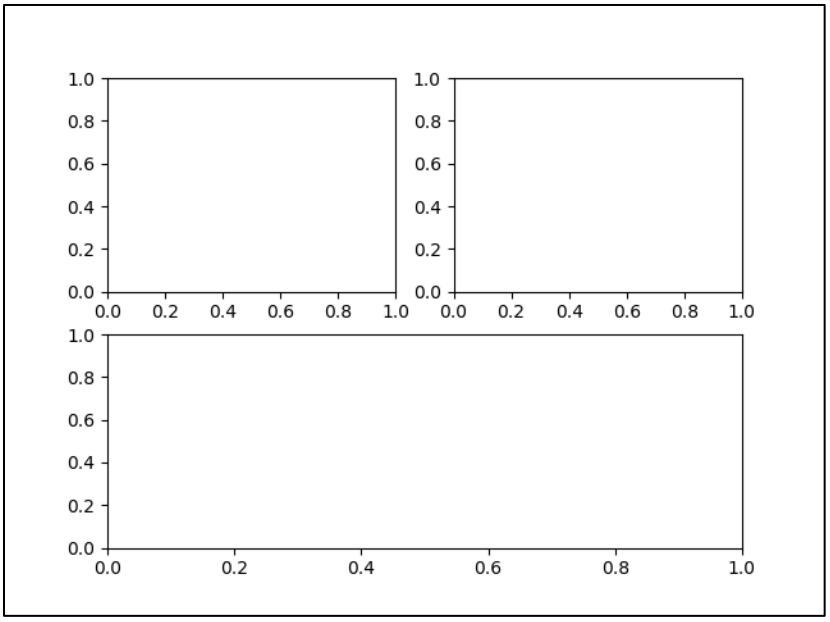
Tampilan yang dihasilkannya sebagai berikut:



Kalian juga dapat dengan bebas melakukan modifikasi tampilan, misalnya seperti contoh ini:

```
>>> f1 = pl.figure(1)
>>> pl.subplot(221)
<matplotlib.axes._subplots.AxesSubplot object at 0x0000000004CD8358>
>>> pl.subplot(222)
<matplotlib.axes._subplots.AxesSubplot object at 0x0000000004CCDF98>
>>> pl.subplot(212)
<matplotlib.axes._subplots.AxesSubplot object at 0x000000000BCD02E8>
>>> pl.show()
```

Sehingga menghasilkan tampilan sebagai berikut:



Untuk melakukan pengaturan ukuran garis tepi dan jarak antar gambar, kalian dapat melakukan perintah seperti ini:

```
>>> pl.subplots_adjust(left = 0.08, right = 0.95, wspace = 0.25, hspace = 0.45)
```

Plotting Data Dalam Suatu File

Dalam komputasi ilmiah, kita seringkali dituntut untuk melakukan *plotting* dari data dalam format ascii (`.txt`). Pada bagian terakhir tutorial ini kita akan secara singkat membahas tentang bagaimana cara membuka, membaca, dan menulis file ascii.

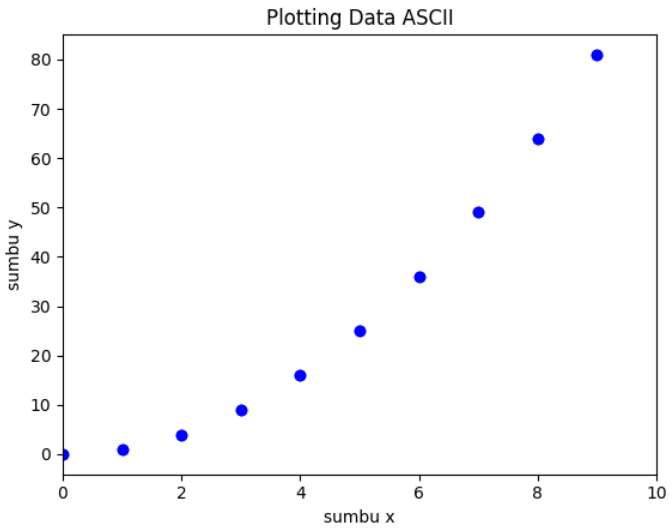
Ada banyak cara untuk membaca data dari suatu file pada Python. Pada bagian ini kita hanya akan membahas salah satu cara sederhana untuk membaca data ascii. Untuk keperluan latihan ada baiknya kalian salin `databohongan.txt` di bawah ini:

```
# databohongan.txt
0    0
1    1
2    4
3    9
4    16
5    25
6    36
7    49
8    64
9    81
0    0
1    1
2    4
3    9
4    16
5    25
6    36
7    49
8    64
9    81
```

Kita dapat membaca `databohongan.txt` tersebut menjadi *array* dua dimensi dan melakukan *plotting* nilai kolom pertama dan kolom kedua dengan menggunakan cara berikut ini:

```
>>> data = np.loadtxt('databohongan.txt')
>>> pl.plot(data[:,0], data[:,1], 'bo')
[<matplotlib.lines.Line2D object at 0x0000000004A78E10>]
>>> pl.xlabel('sumbu x')
<matplotlib.text.Text object at 0x0000000004913278>
>>> pl.ylabel('sumbu y')
<matplotlib.text.Text object at 0x00000000049D7780>
>>> pl.title('Plotting Data ASCII')
<matplotlib.text.Text object at 0x0000000004A087B8>
>>> pl.xlim(0., 10.)
(0.0, 10.0)
>>> pl.show()
```

Berikut tampilan grafiknya:

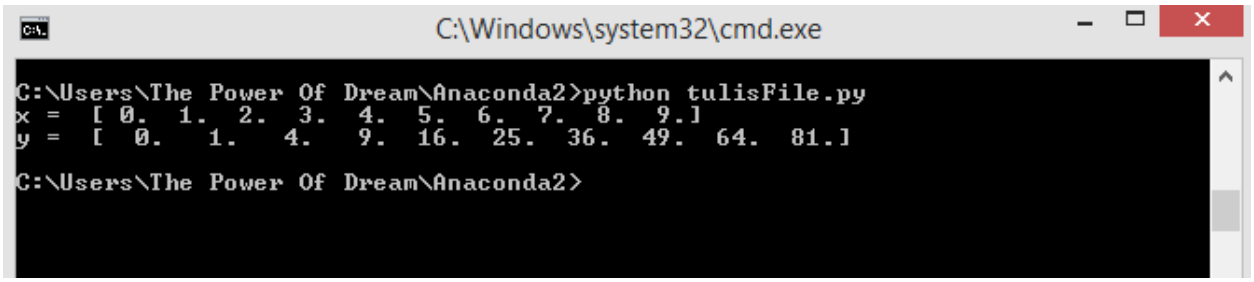


Terdapat banyak cara juga untuk menuliskan teks ke dalam file `.txt`. Di sini kita hanya akan mempelajari salah satu cara yang paling mudah. Hal pertama yang harus kita lakukan adalah membuka suatu file baru untuk ditulis, kemudian melakukan penulisan di dalam file tersebut, dan terakhir menutup file tersebut. Untuk memahaminya coba kalian jalankan *script* berikut ini:

tulisFile.py

```
1  import numpy as np
2  # Buatlah dua buah array (x, y) yang akan kita tuliskan ke dalam file
3  # x merupakan array satu dimensi yang memuat bilangan 1 hingga 10 dengan interval 1
4  x = np.arange(0., 10., 1.)
5  # y merupakan array yang memuat kuadrat nilai x
6  y = x**2
7
8  print "x = ", x
9  print "y = ", y
10
11 # Sekarang bukalah sebuah file untuk dituliskan
12 file = open('tes.txt', 'w')
13
14 #Lakukan pengulangan yang dibutuhkan untuk penulisan file
15 for i in range(len(x)):
16     # membuat string untuk setiap baris dalam file
17     txt = str(x[i]) + '\t' + str(y[i]) + '\n'
18     file.write(txt)
19
20 # menutup file
21 file.close()
```

Berikut hasilnya setelah kita menjalankan *script* tersebut:


C:\Windows\system32\cmd.exe
C:\Users\The Power Of Dream\Anaconda2>python tulisFile.py
x = [0. 1. 2. 3. 4. 5. 6. 7. 8. 9.]
y = [0.   1.   4.   9. 16. 25. 36. 49. 64. 81.]
C:\Users\The Power Of Dream\Anaconda2>

Tutorial ini merupakan sumber pembelajaran alternatif matakuliah pemrograman dasar yang diajarkan di berbagai program studi teknik dan MIPA tingkat diploma dan sarjana pada berbagai perguruan tinggi di Indonesia. Melalui tutorial ini para mahasiswa diharapkan dapat memahami dasar – dasar teknik komputasi ilmiah dengan menggunakan bahasa pemrograman Python.

"Dengan bahasa yang lugas dan informatif, tutorial ini mencakup berbagai hal dasar bahasa pemrograman Python. Sangat cocok bagi pemula atau siapapun yang ingin mengenal Python lebih jauh. Bravo Sandy!"

- **RM Dwiriyo Suryo Sasmoko, Mahasiswa S1 Program Studi Astronomi ITB, *Python Programmer***

WCPL Press

Program Studi Meteorologi ITB
Gedung Labtek XI Lt. 2
Jalan Ganesha 10, Bandung 40132
Telepon: 022-2500494
Faksimili: 022-2534139

Website: http://weather.meteo.itb.ac.id/