Modul_1_PMN

September 12, 2022

0.1 # Modul 1

0.2 Python: Tinjauan secara Ringkas

Bahasa pemrograman Python menjadi salah satu pilhan dalam *Scientific Computing* karena memiliki sintak yang sederhana untuk menjalankan suatu perintah serta mendapatkan banyak dukungan aktif dari para pengembang dalam melengkapi modul tambahan (*library*) yang terkait dengan kebutuhan *scientific*. Fitur Python sebagai sistem *interactive interpreter* merupakan salah satu penyebab sehingga Python relatif mudah untuk dijalankan oleh pengguna awam. Sebagai sistem *interpreter* maka salah satu kendala bawaan yang dimiliki oleh Python adalah pada aspek kecepatan yang pada umumnya belum dapat menyamai kecepatan sistem *compiler* seperti C atau Julia.

Bahasa pemrograman Python dirancang sejak awal oleh Guido van Rossum agar memiliki fitur *extensibility*, yaitu kemudahan bahasa Python untuk dilakukan penambahan fungsi dan kemampuan sesuai kebutuhan. Fitur lain dari Python antara lain: * Bebas dan terbuka (*free* dan *open*) untuk dimanfaatkan sesuai arahan *Python Software Foundation (PSF)*. * Relatif mudah untuk digunakan (*interactive* dan *user-friendly*) * Dukungan modul tambahan (*library*) yang cukup luas dan lengkap

Uraian berikut akan memaparkan secara ringkas tentang sebagian fasilitas yang dimiliki oleh Python dalam melakukan berbagai perintah (*tasks*) yang terkait dengan komputasi. Informasi secara lebih lengkap terkait beberapa uraian yang disajikan di bawah maka dapat merujuk pada Tutorial Python.

0.2.1 Sintak untuk Komentar

Dalam penulisan suatu bahasa pemrograman, kadang diperlukan mekanisme untuk memberikan komentar agar beberapa bagian dalam *source code* menjadi lebih jelas. Untuk itu diperlukan suatu penanda untuk membedakan antara suatu perintah dan suatu komentar. Dalam Python, sintak yang menjadi penanda bagi suatu komentar adalah **#**.

0.2.2 *Variable*

Variable merupakan suatu penamaan yang terkait dengan suatu nilai. Penulisan *variable* dalam Python cukup luas, mirip seperti panamaan dalam bahasa tulis. Nama yang mengandung huruf besar akan dianggap berbeda dengan nama yang mengandung huruf kecil. Berikut adalah contoh penggunaan *variable*:

[1]:
```
# seluruh kata setelah tanda pagar dianggap sebagai komentar, bukan sebagai␣
↪perintah
x = 10
```

[2]:
```
Nilai = 13
```

[3]:
```
nilai = 5
```

0.3 Bilangan Bulat dan Pecahan (*Floating Point*)

0.3.1 Bilangan Bulat

Secara otomatis (*default*), jenis bilangan bulat yang digunakan dalam Python akan dikategorikan dalam tipe `int`. Berbeda dengan bahasa pemrograman lain seperti Julia, cacah bit yang digunakan untuk menyimpan bilangan bulat dalam Python tidak ajeg (*fixed*) berupa 8 bit, 16 bit, 32 bit atau 128 bit, namun berubah tergantung pada bilangan bulat yang diproses. Semakin besar bilangan bulat yang akan diproses maka akan semakin besar cacah bit yang mewakili tipe `int` dan akibatnya akan berpengaruh pada memori dan kecepatan komputer untuk memproses bilangan bulat tersebut.

Beberapa info terkait bilangan bulat yang digunakan pada proses komputasi akan dapat diperoleh dari modul atau *library* `sys`, antara lain `getsizeof` untuk mengetahui cacah bit yang digunakan serta `maxsize` untuk mengetahui bilangan bulat maksimum yang mampu untuk diproses.

[4]:
```
x = 2846
```

[5]:
```
type(x)
```

[5]:
```
int
```

[6]:
```
from sys import getsizeof, maxsize
```

[7]:
```
getsizeof(x)
```

[7]:
```
28
```

[8]:
```
x = 26748585885956969
```

[9]:
```
getsizeof(x)
```

[9]:
```
32
```

[10]:
```
maxsize # bilangan bulat maksimum
```

[10]:
```
9223372036854775807
```

0.3.2 Pecahan (*Float*)

Dalam Python, jenis bilangan pecahan atau *floating point* masuk dalam kategori `float`. Penulisan bilangan jenis `float`dapat berupa penulisan angka pecahan atau dengan notasi-e (sebagai contoh 2.3e-3 sebagai wakilan angka 0.0023).

Seperti halnya bilangan bulat, jenis `float` yang digunakan akan tergantung pada bilangan pecahan saat menjalankan komputasi. Informasi terkait `float` yang mampu diproses oleh komputer yang sedang dijalankan akan dapat diperoleh melalui perintah `float_info` dari modul atau *library* `sys`.

Nilai `epsilon` yang mewakili ketelitian bagi mesin (komputer), yaitu nilai yang menjadi pembeda antara satu *floating point* dengan *floating point* terdekatnya dapat diperoleh melalui perintah `float_info.epsilon` dari modul atau *library* `sys`.

[11]:
```
xf = 2.3e-34
```

[12]:
```
type(xf)
```

[12]:
```
float
```

[13]:
```
from sys import float_info
```

[14]:
```
float_info
```

[14]:
```
sys.float_info(max=1.7976931348623157e+308, max_exp=1024, max_10_exp=308,
min=2.2250738585072014e-308, min_exp=-1021, min_10_exp=-307, dig=15,
mant_dig=53, epsilon=2.220446049250313e-16, radix=2, rounds=1)
```

[15]:
```
float_info.epsilon
```

[15]:
```
2.220446049250313e-16
```

0.3.3 Bilangan Kompleks dan Bilangan Rasional

Dalam berbagai permasalahan, proses komputasi sering melibatkan bilangan kompleks dan bilangan rasional. Dalam Python, sintak `j` digunakan untuk mewakili bilangan imajiner $i = \sqrt{-1}$. Penulisan bilangan kompleks dalam Python mengikuti penulisan matematika baku sebagai contoh `z = 3 + 2j`.

Berbeda dengan Julia, Python tidak memiliki cara alamiah untuk mewakili bilangan rasional, yaitu bilangan yang dapat dinyatakan sebagai pembagian dari 2 bilangan bulat. Apabila diperlukan penggunaan bilangan rasional dalam proses komputasi, Python memfasilitasi hal tersebut melalui *library* `fraction` dan melalui perintah `Fraction`. Penggunaan bilangan rasional dalam proses komputasi sangat berguna dalam berbagai bidang permasalahan.

0.4 Operasi Matematika dan Fungsi Dasar

Beberapa perintah untuk melakukan operasi matematika beserta beberapa fungsi dasar dalam Python memiliki ungkapan yang secara alamiah sama dengan penulisan baku dalam matematika. Beberapa ungkapan operasi matematika beserta fungsi dasar yang lebih lengkap dapat merujuk pada Dokumen Python.

[16]:
```
x = 2.6 + 2.1j
```

[17]:
```
x
```

[17]:
```
(2.6+2.1j)
```

[18]:
```
y = 3
```

[19]:
```
x * y
```

[19]:
```
(7.800000000000001+6.300000000000001j)
```

[20]:
```
x / y
```

[20]:
```
(0.8666666666666667+0.7000000000000001j)
```

[21]:
```
x + y
```

[21]:
```
(5.6+2.1j)
```

[22]:
```
x - y
```

[22]:
```
(-0.3999999999999999+2.1j)
```

[23]:
```
y ** 3
```

[23]:
```
27
```

[24]:
```
y % 2
```

[24]:
```
1
```

[25]:
```
from fractions import Fraction
```

[26]:
```
y + Fraction(1,2)
```

[26]:
```
Fraction(7, 2)
```

[27]:
```
from math import cos
```

[28]:
```
nilai = cos(y)
```

[29]:
```
nilai
```

[29]:
```
-0.9899924966004454
```

0.4.1 Operasi *Boolean* dan Perbandingan Numerik

Dalam beberapa proses komputasi kadang diperlukan untuk memilih berdasar beberapa kondisi atau persyaratan melalui operasi *Boolean*, yaitu operasi untuk melihat suatu keadaan adalah benar (*true*) atau salah (*false*) serta perbandingan 2 nilai secara numerik. Beberapa operasi *Boolean* serta perbandingan numerik yang difasilitasi dalam Python diberikan dalam contoh berikut.

[30]:
```
kondisi1 = 2.5 == 3.7 # operasi kesamaan
```

[31]:
```
kondisi1
```

[31]: False

[32]:
```
kondisi2 = 2.5 != 3.7 # operasi ketidaksamaan
```

[33]:
```
kondisi2
```

[33]: True

[34]:
```
2.5 < 3.7 # operasi kurang dari
```

[34]: True

[35]:
```
2.5 <= 3.7 # operasi kuranf dari atau sama dengan
```

[35]: True

[36]:
```
2.5 >= 3.7 # operasi lebih dari atau sama dengan
```

[36]: False

[37]:
```
not(kondisi1) # operasi boolean NEGATION yaitu kebalikan dari keadaan sebelumnya
```

[37]: True

[38]:
```
kondisi1 & kondisi2 # operasi boolean AND yaitu false jika salah satu false
```

[38]: False

[39]:
```
kondisi1 | kondisi2 # operasi boolean OR yaitu true jika salah satu true
```

[39]: True

0.4.2 Operasi Pembaruan (*updating*)

Dalam beberapa proses komputasi, kadang diperlukan pembaruan suatu nilai oleh operasi matematika tertentu. Sebagai gambaran, untuk memperbarui nilai suatu *variable* agar nilai tersebut bertambah 3 maka Python memberikan operasi ringkas dalam bentuk `x += 3`, yang sama dengan operasi `x = x + 3`. Operasi tersebut juga berlaku untuk bentuk lain yaitu `-=`, `*=`, `/=`, `^=`, `%=` dan lainnya.

[40]:
```
x = 5.6
```

[41]:
```
x
```

```
[41]: 5.6
```

```
[42]: x += 3
```

```
[43]: x
```

```
[43]: 8.6
```

```
[44]: y = 2
```

```
[45]: y
```

```
[45]: 2
```

```
[46]: y **= 3 # sama dengan operasi y = y^3
```

```
[47]: y
```

```
[47]: 8
```

0.5 *Array*

Array merupakan jenis *variable* berupa jajaran dari nilai-nilai yang disajikan dalam bentuk 1, 2 hingga multi-dimensi. *Array* dalam 1 dimensi digunakan untuk melambangkan suatu vektor yang memiliki bentuk berupa jajaran nilai pada beberapa baris dalam satu kolom (biasa disebut sebagai matrik kolom). *Array* dalam 2 dimensi digunakan untuk melambangkan suatu matrik yang memiliki bentuk berupa jajaran nilai pada beberapa baris dan kolom. Secara umum *array* dalam 3 dimensi atau multi-dimensi biasa digunakan untuk melambangkan suatu tensor.

Array memegang peran penting dalam komputasi karena operasi yang melibatkan banyak nilai, yang berarti banyak *variable*, akan dapat diwakili oleh satu *variable* sehingga menyederhanakan prosedur penyelesaian. Tidak seperti dalam Julia, dukungan *array* dalam Python tidak diimplementasikan dalam bentuk perintah bawaan, namun didukung melalui *library* tambahan yaitu modul `Numpy`. Implementasi operasi matematika terkait *array* pada `Numpy` cukup lengkap yang dapat dirujuk pada Dokumentasi *Numpy*. Berikut disajikan beberapa hal penting dari implementasi *array* tersebut.

0.5.1 Beberapa Fungsi Dasar

Fasilitas *array* dalam Python dapat dimanfaatkan dengan memuat *library* `Numpy` dengan sintak `import numpy` atau dalam penggunaan sering ditambahkan alias yaitu `import numpy as np`. Untuk membangkitkan *array* 1 dimensi yang mewakili vektor maka dapat ditulis dengan sintak `array([x1, x2, ...])`, dengan `x1, x2, ...` merupakan unsur-unsur vektor. Sebagai catatan, yang berbeda dengan Julia atau kebiasaan dalam operasi matematika, perintah pembangkit vektor dalam Python tersebut akan berupa vektor baris (yaitu bukan vektor yang diwakili oleh matrik kolom 1 dimensi).

Array dalam 2 dimensi yang berupa matrik baris dan kolom beserta perluasannya dalam *array* multi-dimensi dapat dibangkitkan dengan sintak `array([[x1, x2, ...], [y1, y2, ...],`

...]).

Dalam beberapa permasalahan komputasi, kadang diperlukan *array* dengan bentuk khusus sebagai contoh matrik yang memiliki semua unsur bernilai 1. Beberapa fungsi dasar untuk membangkitkan *array* dalam bentuk khusus tersebut diberikan pada contoh berikut.

[48]:
```
import numpy as np
```

[49]:
```
A = np.array([1,2,3])
```

[50]:
```
print(A)
```

```
[1 2 3]
```

[51]:
```
B = np.array([[1,2,3], [3,4,5], [6,7,8]])
```

[52]:
```
print(B)
```

```
[[1 2 3]
 [3 4 5]
 [6 7 8]]
```

[53]:
```
C = np.zeros((2,3)) # matrik orde 2x3 (2 baris dan 3 kolom) dengan semua unsur␣
↪bernilai 0
```

[54]:
```
print(C)
```

```
[[0. 0. 0.]
 [0. 0. 0.]]
```

[55]:
```
D = np.ones((3,4)) # matrik orde 3x4 dengan semua unsur bernilai 1
```

[56]:
```
print(D)
```

```
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]]
```

[57]:
```
E = np.random.rand(3,2) # matrik orde 3x2 dengan semua unsur bernilai bilangan␣
↪random dalam interval 0 dan 1
```

[58]:
```
print(E)
```

```
[[0.33590251 0.09066349]
 [0.33489183 0.45373704]
 [0.82294341 0.42024133]]
```

[59]:
```
print(E.size) # operasi untuk menunjukkan cacah unsur pada matrik E
```

```
6
```

[60]:
```
print(E.shape) # operasi untuk menunjukkan daftar urutan (*tuple*) dimensi atau orde bagi matrik E
```

```
(3, 2)
```

[61]:
```
print(E.ndim) # operasi untuk menunjukkan dimensi bagi matrik E
```

```
2
```

0.5.2 Indeks dalam *Array*

Berbeda dengan Julia, aturan penomoran atau indeks suatu *array* pada Python dimulai dengan indeks 0 (bukan 1). Kemudian urutan indeks dalam seluruh *array* didasarkan pada urutan baris (bukan kolom). Contoh berikut adalah perintah untuk menampilkan unsur dari suatu *array* (vektor atau matrik) berdasar aturan indeks tersebut.

Penomoran atau pengaturan indeks dari suatu unsur tertentu dalam suatu *array* dalam dilakukan dengan cara *Cartesian Index* yaitu penomoran dengan mengikuti indeks bagi *array* tersebut, sebagai contoh unsur `A[2.3]` adalah unsur beris ke 2 dan kolom ke 3 dari matrik `A`.

Tidak seperti Julia, penomoran secara *Linear Index* dalam Python, yaitu penomoran dengan mengikuti urutan lokasi suatu unsur tertentu bagi *arrtay* tersebut, nampak belum terfasiliasi dalam modul bawaan.

[62]:
```
A = np.array([1,2,3,4,5])
```

[63]:
```
print(A)
```

```
[1 2 3 4 5]
```

[64]:
```
A[4] # mencuplik nilai dari unsur ke 5 (tapi dengan indek 4 karena dimulai dari 0) bagi vektor A
```

[64]:
```
5
```

[65]:
```
B = np.array([[1,2,3],[3,4,5],[6,7,8]])
```

[66]:
```
print(B)
```

```
[[1 2 3]
 [3 4 5]
 [6 7 8]]
```

[67]:
```
B[1,1]
```

[67]:
```
4
```

[68]:
```
C = np.reshape(B,9) # urutan unsur berdasar baris, diikuti kolom
```

[69]:
```
print(C)
```

```
[1 2 3 3 4 5 6 7 8]
```

0.5.3 Operasi Vektorisasi pada *Array*

Operasi vektorisasi merupakan operasi matematika yang bekerja pada seluruh unsur pada *array* secara bersamaan, bukan satu persatu pada tiap unsur *array*. Mekanisme operasi matematika pada seluruh unsur yang semacam ini sehingga operasi vektorisasi kadang disebut pemrosesn berbasis unsur (*element-wise processing*). Fitur operasi vektorisasi dalam Python, melalui modul `Numpy`, bukan hanya menyederhanakan proses komputasi namun secara umum dapat mempercepat proses komputasi, khususnya pada komputer yang memfasilitasi pemrosesan paralel.

Python mengimplementasikan operasi vektorisasi dengan modul `Numpy` melalui operasi matematika dan fungsi yang biasa bekerja pada *array*, bukan pada skalar. Sebagai contoh, untuk mengalikan suatu nilai yang diwakili oleh *variable* `x1` dengan nilai `x2` maka digunakan sintak `x1 * x2`. Adapun untuk mengalikan suatu nilai yang diwakili oleh *variable* `x1` dengan seluruh nilai pada suatu *array* yang diwakili oleh *variable* `X2` maka diganakan sintak `x1 * X2` setelah memuat modul `Numpy` dengan sintak `import numpy`.

[70]:
```
import numpy as np
```

[71]:
```
x1 = 2.0
```

[72]:
```
x2 = 0.4
```

[73]:
```
X2 = np.random.rand(2,2)
```

[74]:
```
print(X2)
```

```
[[0.57282759 0.36119154]
 [0.82753265 0.58303964]]
```

[75]:
```
x1 * x2
```

[75]:
```
0.8
```

[76]:
```
print(x1 * X2)
```

```
[[1.14565518 0.72238308]
 [1.65506529 1.16607928]]
```

Implementasi *element-wise* memerlukan kehati-hatian ketika melibatkan matrik agar tidak rancu dengan operasi matrik sejenis yang secara definisi boleh jadi berbeda. Sebagai contoh, operasi perkalian matrik `A` dengan matrik `B` dengan sintak `A @ B` secara definisi dilakukan dengan mengalikan baris matrik `A` dengan kolom matrik `B`. Oleh karena itu persyaratan agar `A @ B` berlaku adalah jumlah kolom matrik `A` perlu sama dengan jumlah baris matrik `B`.

Sedangkan operasi perkalian *element-wise* matrik `A` dengan matrik `B` dengan sintak `A * B` secara definisi dilakukan dengan mengalikan setiap unsur matrik `A` dengan setiap unsur matrik `B`pada

indeks matrik yang sama. Dengan demikian tidak ada persyaratan bahwa jumlah kolom matrik `A` perlu sama dengan jumlah baris matrik `B`agar operasi perkalian *element-wise* berlaku. Contoh berikut memberikan gambaran jelas terkait uraian tersebut.

[77]:
```
import numpy as np
```

[78]:
```
A = np.random.rand(2,2)
```

[79]:
```
print(A)
```

```
[[0.17451382 0.834815  ]
 [0.4873814  0.64224242]]
```

[80]:
```
B = np.random.rand(2,2)
```

[81]:
```
print(B)
```

```
[[0.70208001 0.08276305]
 [0.70195111 0.90799735]]
```

[82]:
```
C = A @ B
```

[83]:
```
print(C)
```

```
[[0.70852198 0.7724531 ]
 [0.79300352 0.62349158]]
```

[84]:
```
D = A * B
```

[85]:
```
print(D)
```

```
[[0.12252266 0.06909183]
 [0.34211791 0.58315441]]
```

[86]:
```
C == D
```

[86]:
```
array([[False, False],
       [False, False]])
```

0.5.4 Operasi *Broadcasting*

Persyaratan agar operasi vektorisasi berbasis *element-wise* berlaku adalah adanya kesamaan orde bagi *array* yang terlibat dalam operasi. Apabila *array* yang terlibat ternyata tidak memiliki orde yang sama maka proses *element-wise* diimplemnetasikan melalui operasi *broadcasting*.

[87]:
```
import numpy as np
```

[88]:
```
A = np.random.rand(2,2)
```

[89]:
```
print(A)
```

```
[[0.65569324 0.71134735]
 [0.04587177 0.59410897]]
```

[90]:
```
B = np.random.rand(1,2)
```

[91]:
```
print(B)
```

```
[[0.90851564 0.22801329]]
```

[92]:
```
C = A + B
```

[93]:
```
print(C)
```

```
[[1.56420887 0.93936064]
 [0.95438741 0.82212226]]
```

0.6 Blok atau Gabungan Perintah

Suatu blok atau gabungan beberapa perintah kadang perlu diproses agar beberapa hasil yang diperoses tersebut akan dapat digunakan untuk menghasilkan satu hasil akhir. Blok atau gabungan beberapa perintah tersebut dalam Python dapat dimplementasikan dalam berbagai bentuk, seperti bentuk fungsi, proses *conditional* atau proses *looping*, yang memiliki penanda berupa tambahan (*indent*) beberapa spasi pada blok yang sama seperti contoh berikut. Adanya *indent* juga memiliki peran lain yaitu agar blok program menjadi mudah terbaca (*readable*).

[94]:
```
for i in range(1,6):
    i = i**3
    print(i)
```

```
1
8
27
64
125
```

Dalam ungkapan di atas, sintak `range(1:6)` mamiliki arti bahwa *variable* `i` akan bernilai dari `1` hingga nilai `5`.

0.7 Fungsi dalam Python

Kebanyakan fungsi dasar yang biasa digunakan di matematika, sebagai contoh fungsi trigonometri, telah terdefinisi dalam Python melalui pemanggilan *library* yang sesuai sehingga dapat langsung diimplementasikan dalam proses komputasi. Apabila diperlukan bentuk fungsi yang belum termasuk dalam *library* Python, maka fungsi tersebut perlu didefinisikan terlebih dahulu sebelum dapat dimanfaatkan.

Secara umum bentuk fungsi dalam Python akan berupa suatu obyek yang memetakan seperangkat argumen bagi fungsi, untuk menghasilkan satu atau beberapa nilai bagi fungsi tersebut. Contoh

berikut merupakan metode yang dapat dipilih untuk mendefinisikan suatu fungsi. Uraian lebih lengkap terkait fungsi dapat merujuk pada Dokumen Fungsi.

[95]:
```
def f(x,y):
    nilai = x * y
    return nilai
```

[96]:
```
hasil = f(2,4)
```

[97]:
```
hasil
```

[97]: 8

0.8 Proses Bersyarat (*Conditional*)

Dalam proses komputasi, suatu evaluasi tertentu kadang perlu dilakukan ketika satu persyaratan dipenuhi dan sebaliknya, suatu evaluasi tersebut tidak perlu dilakukan ketika persyaratan tersebut tidak terpenuhi. Proses bersyarat seperti hal tersebut dapat diimplementasikan melalui sintak `if-elif-else` seperti contoh berikut.

[98]:
```
def hasil(x,y):
    if (x < y):
        nilai = x + y
        print(nilai)
    elif (x > y):
        nilai = x - y
        print(nilai)
    else:
        nilai = x * y
        print(nilai)
```

[99]:
```
hasil(3,5)
```

```
8
```

[100]:
```
hasil(5,2)
```

```
3
```

[101]:
```
hasil(5,5)
```

```
25
```

0.9 Proses Berulang (*Looping*)

Ketika suatu proses perlu dilakukan secara berulang maka perintah untuk proses tersebut akan menjadi panjang dan oleh karena itu diperlukan mekanisme untuk meringkas perintah dalam bentuk proses berulang (*looping*). Proses *looping* dapat diwujudkan dalam dua sintak yaitu `while`dan `for` seperti contoh berikut.

[102]:
```
def total(i):
    i = 1
    while (i < 10):
        i += 2
        print(i)
```

[103]:
```
total(9)
```

```
3
5
7
9
11
```

[104]:
```
def total(n):
    for i in range(1,n+1):
        j = i + 2
        print(j)
```

[105]:
```
total(9)
```

```
3
4
5
6
7
8
9
10
11
```

0.10 Penggunaan Modul Tambahan

Dalam beberapa keperluan, kadang diperlukan suatu operasi tertentu yang tidak termasuk dalam paket (*library*) bawaan Python. Sebagai contoh, keperluan penggunaan matrik yang sering muncul dalam berbagai perhitungan aljabar linear ternyata tidak termasuk dalam *library* bawaan. Untuk keadaan seperti hal tersebut maka diperlukan pemanggilan *library* tambahan dengan menggunakan sintak `import`. Di dalam Panduan Numpy akan dapat ditemukan bahwa operasi matematika yang melibatkan *array* termuat dalam *library* yang disebut `numpy`.

Untuk memuat suatu modul atau *library* tertentu maka dapat digunakan sintak `import`, sebagai contoh `import numpy`. Ketika nama nama *library* perlu dipanggil beberapa kali maka nama *library* tersebut dapat dipersingkat dengan nama alias lain melalui sintak `as`, sebagai contoh `import numpy as np`.

Untuk meringankan kebutuhan memori, kadang hanya diperlukan sebagian dari keseluruhan *library*. Perintah untuk hal ini dapat dilakukan dengan sintak `from` dan `import` sebagai contoh

```
from sys import getsizeof.
```

[106]:
```
import numpy as np
A = np.random.rand(3,4)
print(A)
```

```
[[0.29768127 0.88974996 0.28971401 0.71290509]
 [0.69005134 0.97847354 0.36814277 0.24387388]
 [0.8690997  0.04762028 0.83566206 0.62309086]]
```

0.11 Plot: Visualisasi Data

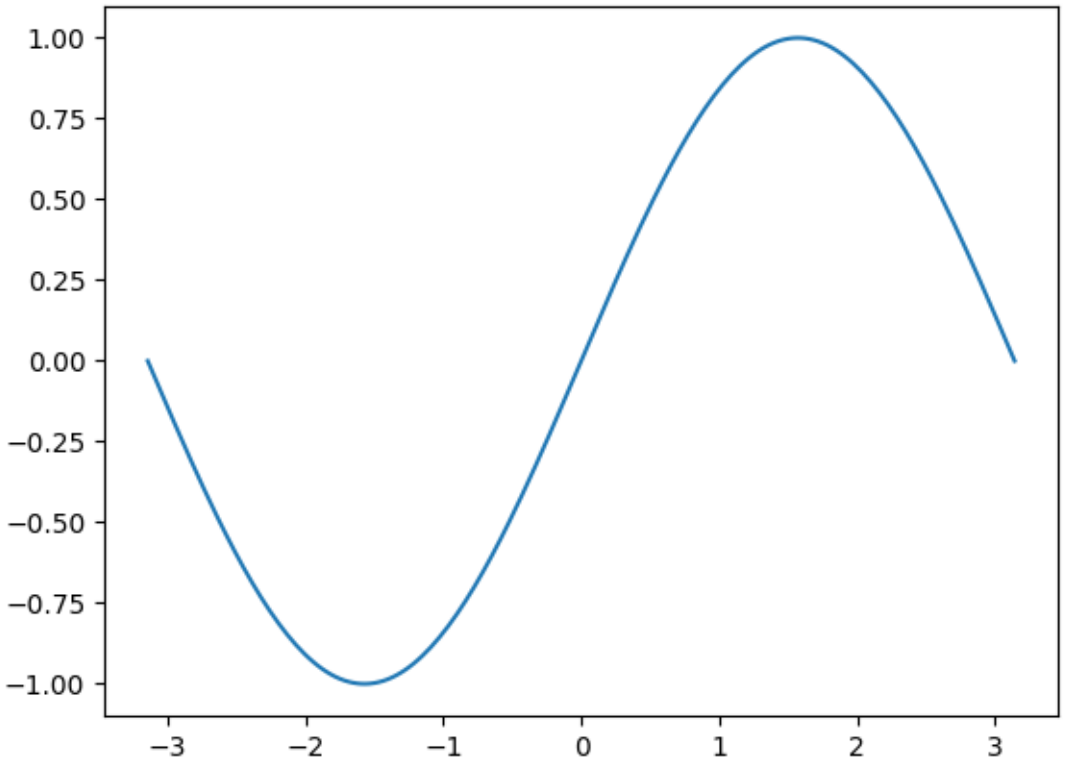
Hasil dari proses komputasi pada umumnya melibatkan banyak data dan memerlukan penanganan lebih lanjut terhadap data tersebut. Salah satu cara untuk penanganan data adalah dengan melakukan visualisasi berupa plot terhadap data tersebut. Python memeiliki fasilitas ynag cukup lengkap untuk membangkitkan berbagai jenis plot sesuai karakteristik data. Berbagai implementasi perintah plot di dalam Python dapat merujuk pada Dokumen Plot.

Sintak untuk membangkitkan plot adalah dengan memuat *library* yaitu `import matplotlib` dan dikuti dengan beberapa perintah tertentu. Karena data yang diperlukan untuk proses plot biasnya perlu disajikan dalam bentuk *array* maka penggunaan `matplotlib` perlu disertai dengan pemanggilan *library* `numpy`. Berikut merupakan beberapa contoh berdasar dokumen tersebut.

[107]:
```
import matplotlib.pyplot as plt
import numpy as np

x = np.linspace(-np.pi, np.pi, 200)
y = np.sin(x)

fig, ax = plt.subplots()
ax.plot(x, y)
plt.show()
```



0.12 Tugas

Pada contoh-contoh yang diberikan pada setiap Topik di atas maka tambahkan beberapa perintah terkait topik tersebut. Setelah memberikan tambahan beberapa perintah, kemudian berikan sedikit narasi yang menunjukkan bahwa topik tersebut telah dipahami.

Sebagai contoh, berikut akan ditunjukkan beberapa tambahan perintah dan narasi yang terkait dengan topik *Variable* pada bagian awal Modul ini.

[108]:
```
# seluruh kata setelah tanda pagar dianggap sebagai komentar, bukan sebagai␣
↪perintah
x = 10
```

[109]:
```
Nilai = 13
```

[110]:
```
nilai = 5
```

[111]:
```
Masukan_saya = "rde"
```

[112]:
```
Masukan_saya
```

[112]:
```
'rde'
```

[113]:
```
masukan_saya = 3
```

[114]:
```
Masukan_saya == masukan_saya
```

[114]:
```
False
```

Panamaaan suatu *variable* dalam *Python* ternyata sangat kaya bentuk karena mampu membedakan antara nama *variable* yang mengandung huruf besar atau huruf kecil.

[ ]:
```

```