

Appendix

In this appendix we first describe how to build an environment in which all the program code in this book can be performed on our computer. The platforms covered here are Windows, macOS and Raspberry Pi OS. We introduce the following two environments for using Python:

- IDLE --- a classical integrated development environment of Python;
- Jupyter --- an educational environment for using Python.

We need the following external libraries of Python:

- PIL --- for reading image files;
- NumPy --- for mathematical functions, numeric vector and matrix calculation and random numbers;
- Matplotlib --- for drawing 2D/3D figures and graphs of functions;
- SciPy --- for reading and writing audio data;
- SymPy --- for symbolic calculation;
- VPython --- for 3D rendering and animation.

These are very powerful widely used libraries, but the uses listed above are just a few. This book is not an introduction to these libraries. We use Python and these libraries as an aid for mathematical formulas and natural language in order to explain the idea of linear algebra.

Second, we will roughly explain uses of Python and the libraries listed above, which the reader must know before starting Chapter 1. However, the actual understanding of Python programming will be obtained by studying the many code examples found in this book. We also recommend that the reader refers to internet pages such as <https://docs.python.org/3/reference/index.html> when necessary.

Finally, we introduce other Python libraries and non-Python applications, which were used by the authors on a Raspberry Pi to write this book.

A.1 Python Environment Used in This Book

A.1.1 Windows

We recommend to install a system called Anaconda, because it includes Python3 and all the libraries we need except VPython. Get Anaconda from <https://www.anaconda.com> and install it. There are two versions for 32-bit and 64-bit Windows, but if unsure which to chose, try to install the latter first. When we start the installation, we will be asked if it is our machine or just our own account, but the latter is a good choice.

A.1.2 macOS

Anaconda is also available; get it from <https://www.anaconda.com> and install it. Here, we explain another way to install Python using the latest version of Python3 obtained from <https://www.python.org/>. When the installation of Python3 is complete, a new folder named \texttt{Python3.*} will be created in the Application folder. Open a terminal window from the Utility folder and execute the following commands line by line.

```
python3 -m pip install --upgrade pip
python3 -m pip install jupyterlab
python3 -m pip install pillow
python3 -m pip install numpy
python3 -m pip install matplotlib
python3 -m pip install scipy
python3 -m pip install sympy
python3 -m pip install vpython
```

If the installed version is Python 3.9, executable commands will be found in `/Library/Frameworks/Python.framework/Versions/3.9/bin`, and a path to this directory is automatically created.

A.3. Python on Raspberry Pi OS

Before installing Python and the libraries on a Raspberry Pi, even if the reader already has a micro SD card with OS installed, we highly recommend that the reader prepares another card, downloads the latest OS image file from <http://raspberrypi.org> and installs it. This is not only to be able to use the latest OS, but also to prevent any damage to the card. Here, we will start with a micro SD card with a newly installed Raspberry Pi OS. We have several choices for installing libraries.

If the reader has Raspberry Pi 4 or later (recommended), open LXTerminal from the task bar on the desktop, and execute the following commands line by line.

For Raspberry Pi 4/400 (Recommended)

```
sudo apt install idle3 -y
python3 -m pip install --upgrade --user pip
~/.local/bin/pip install --user jupyterlab
~/.local/bin/pip install --user vpython
sudo apt install python3-matplotlib -y
sudo apt install python3-scipy -y
sudo apt install python3-sympy -y
```

Jupyter, VPython and related libraries are installed in the folder `home/pi/.local/`.

If using a Raspberry Pi Zero to 3 model, omit lines 2--4. This is because using Jupyter Notebook or VPython on these models is not practical due to inadequate cpu performance. However, it is worth noting that much of the code in this book that does not use them can also be run on these models, allowing linear computations to be performed locally as an embedded system.

There are also ways to create a virtual environment and install multiple different Python environments, but since Raspberry Pi can create different OS environments by easily replacing the micro SD card, we will not explain the Python virtual environment here.

If the reader has Raspberry Pi zero or up to 3, execute the following commands.

For Raspberry Pi zero and up to Model 3

```
sudo apt install idle3 -y
sudo apt install python3-matplotlib -y
sudo apt install python3-scipy -y
sudo apt install python3-sympy -y
```

In this case, Jupyter and VPython cannot be used.

The following is an alternative for Raspberry Pi 4/400.

Raspberry Pi 4/400 (Alternative)

```
sudo apt install idle3 -y
sudo apt install libatlas-base-dev -y
sudo python3 -m pip install --upgrade pip
sudo python3 -m pip install --upgrade pillow
sudo python3 -m pip install --upgrade numpy
sudo python3 -m pip install matplotlib
sudo python3 -m pip install scipy
sudo python3 -m pip install sympy
sudo python3 -m pip install jupyterlab
sudo python3 -m pip install vpython
```

Without the second line, we will encounter an error when importing Numpy upgraded. The forth line upgrades PIL preinstalled.

Ubuntu users are to proceed as follows.

Ubuntu 64-bit Desktop for Raspberry Pi 4/400

```
sudo apt install idle3 -y
sudo apt install python3-pip -y
sudo python3 -m pip install --upgrade pip
sudo python3 -m pip install --upgrade pillow
sudo python3 -m pip install numpy
sudo python3 -m pip install matplotlib
sudo python3 -m pip install scipy
sudo python3 -m pip install sympy
sudo python3 -m pip install jupyterlab
sudo python3 -m pip install vpython
```

When this book is written, the latest version of Python3 is 3.10.6.

The following steps to install it need the skill to build Python from the source. This is for the advanced reader.

To install the latest version of Python3 for Raspberry Pi 64-bit (Advanced)

```

sudo apt install tk-dev -y
sudo apt install libncurses5-dev -y
sudo apt install libncursesw5-dev -y
sudo apt install libreadline6-dev -y
sudo apt install libdb5.3-dev -y
sudo apt install libgdbm-dev -y
sudo apt install libsqlite3-dev -y
sudo apt install libssl-dev -y
sudo apt install libbz2-dev -y
sudo apt install liblzma-dev -y
sudo apt install libffi-dev -y

```

Get the source release from <https://www.python.org/downloads/>. Extract the downloaded compressed file into the `\texttt{/tmp}` folder. Follow the steps below to compile and install Python-3.10.6.

```

cd /tmp/Python-3.10.6
./configure --enable-optimizations
make -j 4
sudo make altinstall

```

This completes the Python 3.9.6 installation. Then, prepare to install the libraries we need.

```

sudo apt install libjpeg-dev -y
sudo apt install liblapack-dev -y
sudo apt install gfortran -y
sudo apt install rustc -y
sudo /usr/local/bin/python3.10 -m pip install --upgrade pip
sudo /usr/local/bin/python3.10 -m pip install wheel
sudo /usr/local/bin/python3.10 -m pip install cryptography

```

Finally, install the libraries.

```

sudo /usr/local/bin/python3.10 -m pip install pillow
sudo /usr/local/bin/python3.10 -m pip install numpy
sudo /usr/local/bin/python3.10 -m pip install matplotlib
sudo /usr/local/bin/python3.10 -m pip install scipy
sudo /usr/local/bin/python3.10 -m pip install sympy
sudo /usr/local/bin/python3.10 -m pip install jupyterlab
sudo /usr/local/bin/python3.10 -m pip install vpython

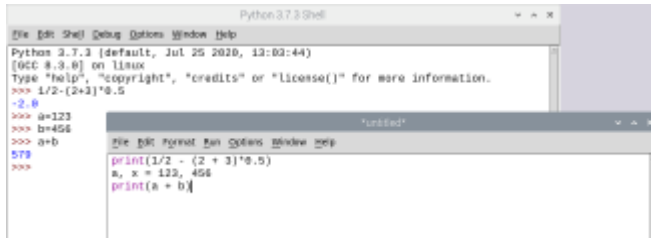
```

Executable commands are installed in the folder `/usr/local/bin`.

A.4. Launching Python

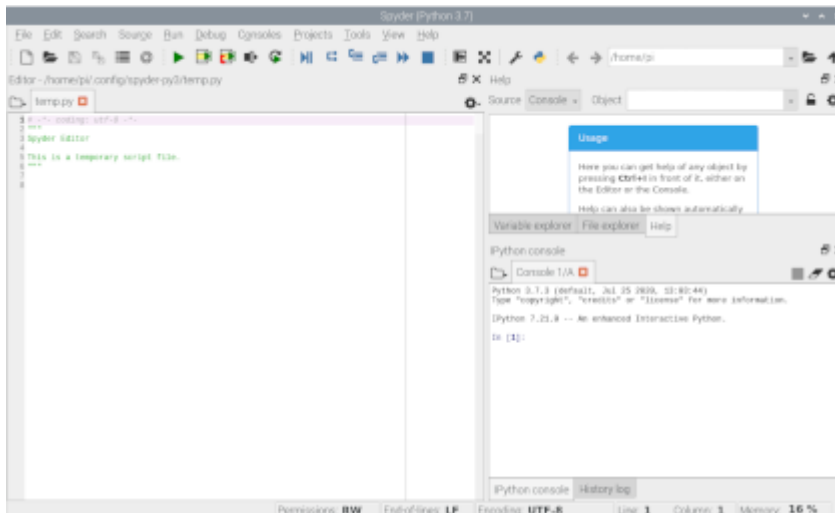
To work with Python, it is convenient to use an *IDE* (Integrated Development Environment). The IDE integrates an editor that writes program code and an environment that runs it interactively. With the settings up to this point, an IDE named `IDLE` can be used on all platforms, so here we will proceed using `IDLE`.

IDLE has been the standard for Python's IDE, but recently, other more sophisticated IDEs have appeared, and **IDLE** is gradually being superseded. In this book, we introduce some different Python environments, but we cannot find a common new IDE to recommend. Although they are of higher performance than **IDLE**, we need to explain the functions one by one and it may be a high threshold for beginners. Python installed from python.org ships with **IDLE**, which still has merits.



- **Windows:** On Windows with Anaconda installed, open `Anaconda PowerShell Prompt` and execute the `idle` command to launch it.

Anaconda ships with a newer Python IDE called **Spyder**, which we can also use.

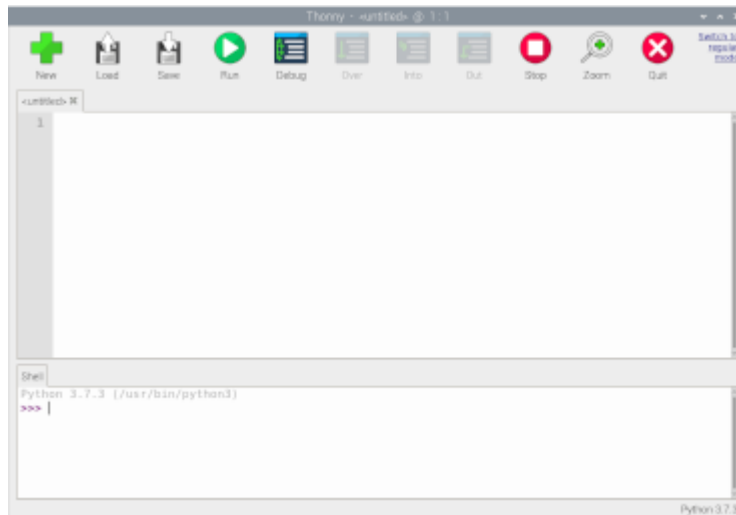


- **macOS:** When using Python obtained from python.org, we can find a folder called Python3 in the Application folder, so launch it from the icon named **IDLE** in the folder.

Alternatively, open a terminal window and execute the `idle3` command to launch it.

- **Raspberry** Launch it from **Python 3 (IDLE)** in Programming of the main menu, or open **LXTerminal** and execute the `idle` command.

Raspberry Pi's default Python IDE named **Thonny Python IDE** found in the same menu is also available.



- **Ubuntu Desktop** Launch it from IDLE in Show Applications of Dock, or open a terminal and execute the `idle` command.

When we launch `IDLE`, we see a message containing the Python version etc. This window is called a *shell window*, the place where we interact with Python.

The first `>>>` is the prompt for input.

Here we have a simple dialogue.

Interaction on the shell window of IDLE:

```
>>> 1/2-(2+3)*0.5
-2.0
>>> a=123
>>> x=456
>>> a+x
579
>>> x
456
>>> x=x+a
>>> x
579
```

Expression `x=x+a` is called an assignment statement and it overwrites `x`. `x=x+a` is simply written as `x+=a`, which is called an *augmented assignment statement*. There are other augmented assignment operators `*=`, `-=` and `/=`.

In some new type of IDE, for example, Spyder, the window is called the console and its prompt is `In[*]`, where `*` is a number. The dialogue looks like this:

Interaction on the console of Spyder:

```
In [1]: 1/2-(2+3)*0.5
Out[1]: -2.0

In [2]: a=123

In [3]: x=456
```

```
In [4]: a+x
Out[4]: 579
```

After the prompt `>>>` or `In[*]`, we type the arithmetic formula on the keyboard and press the Enter/Return key, then the calculation result is returned. We can use variables as memory. This usage of Python is called an *interactive mode*, which is a sophisticated calculator. Do not enter a space at the beginning of the line following the shell prompt. A space at the beginning of a line has an important meaning in Python. This will be discussed later. It is allowed to put spaces before and after the operators `+`, `*`, `=`, etc., for readability. Before pressing Enter/Return, we can use the arrow keys, Backspace key, and Delete key to edit a line.

When we use IDLE, from the menu bar of the shell window, select "New File" in "File" menu. In the case of macOS, activate the shell window and select it from the menu bar displayed at the top of the screen. Then another window called the *edit window* will open. We write the Python program code there. Let us program the same calculation done in the above interactive modes.

Program on the Editor (example01.py):

```
print(1/2 - (2 + 3)*0.5)
a, x = 123, 456
print(a + x)
```

As in interactive mode, do not put a space at the beginning of each line. In this example, spaces are placed on both sides of some operators. In Python code, the `print` function is used to print the calculation result. If we write only the formula in a program, it will not be displayed, though it will be calculated. If we want to see the value assigned in a variable, after running the program, enter an interactive mode and refer to the variable. When the program is complete, select "Run Module" from "Run" in the menu bar of the edit window, alternatively, simply press the function key F5. In the case of macOS, activate the edit window and select it from the menu bar displayed at the top of the screen. Because we will be asked if we want to save the file, click the "OK" button. Then, a dialog for specifying the save destination (file name and folder) is displayed. Let us name it `example01` and save it to the current folder. Click the "OK" button to save the file and display the calculation results in the shell window. Python program files are automatically given names with extension `.py`.

Results and interaction on the shell window:

```
-2.0
579
>>> a, x
(123, 456)
```

If we modify the code and run it, the program file will be overwritten.

Let us quit IDLE once here by closing both the shell window and the edit window. Then, open a terminal, move to the folder where `example01.py` is saved, and execute the

following command.

Windows:

```
python example01.py
```

macOS, Raspberry Pi OS and Ubuntu Desktop:

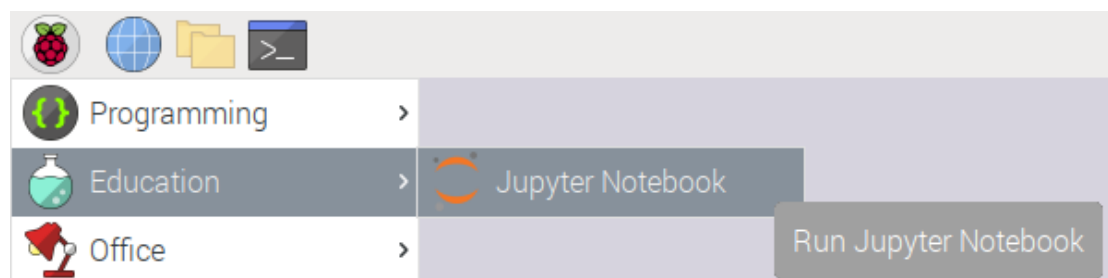
```
python3 example01.py
```

The calculation result of the program will be displayed on the terminal.

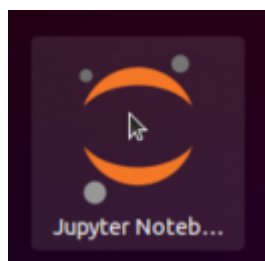
A.5. Using Jupyter Notebook

We can use Jupyter Notebook that is a Web application instead of Python's IDE.

- **Windows or macOS:** When Anaconda is installed, launch it from Anaconda Navigator.
- **Raspberry Pi 4/400:** Select item "Jupyter Notebook" in the menu of "Programming" or "Education".



- ***Ubuntu 64-bit Desktop for Raspberry Pi 4/400:** Select Jupyter Notebook icon in "Show Applications" of Dock.

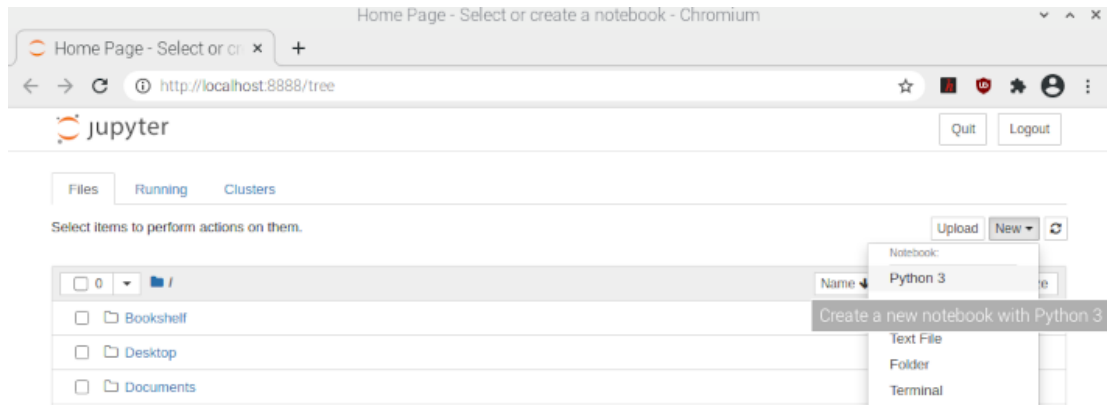


In all environments we can launch it by executing the following command.

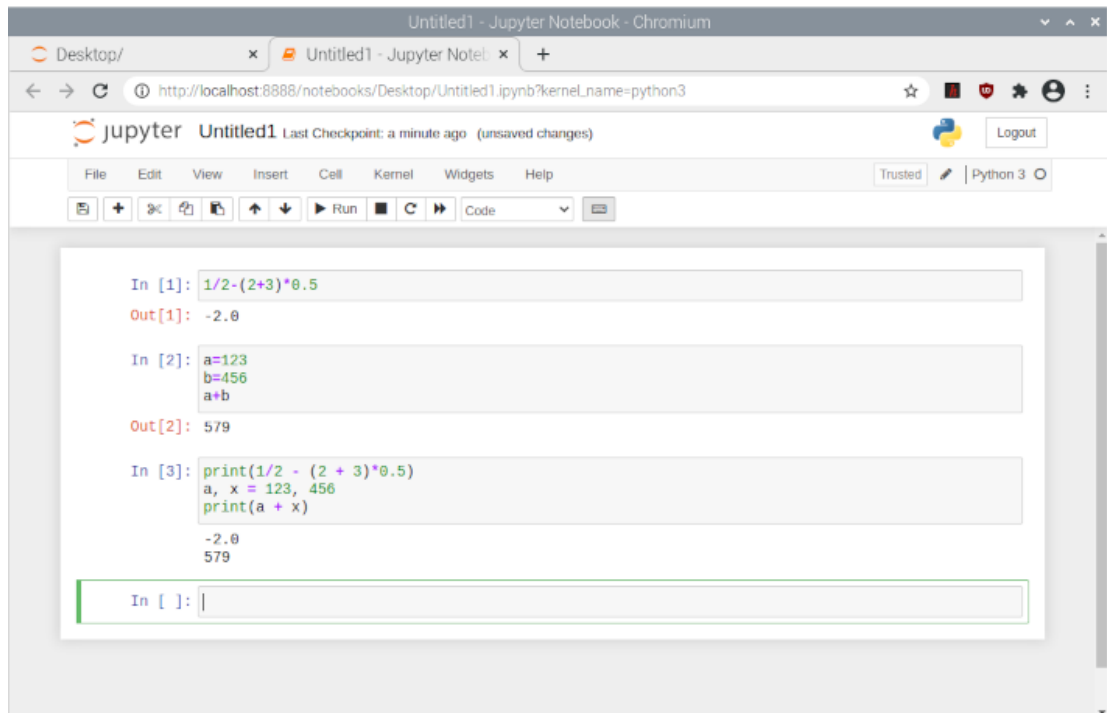
```
jupyter notebook
```

We may replace the above command by `jupyter-notebook`.

If we launch it, the Web browser starts up. Move to a suitable directory displayed on the tab if necessary. Next, create a new notebook of Python 3.



Then another tab called a notebook will open.



In the notebook `In[n]` is a prompt, where n indicates the order of dialogue with Python. The gray zone is called a *cell*, where we can write a question with Python code. To tell Python a question, hold down the shift key and press enter (Shift+Enter). Pressing only the enter key means a line break. We can write multiple lines of Python code in a cell.

```
In [1]: a=123
        x=456
        a+x
```

```
Out[1]: 579
```

By Shift+Enter we get the answer.

We can also write multiple Python program codes in a cell. If we want to see the line numbers, press the escape key and then the "L" key. This operation is toggled.

[Untitled.ipynb](#)

```
In [2]: 1/2-(2+3)*0.5
```

```
Out[2]: -2.0
```

```
In [3]: print(1/2 - (2 + 3)*0.5)
a, x = 123, 456
print(a + x)
```

```
-2.0
579
```

```
In [4]: a, x
```

```
Out[4]: (123, 456)
```

These interactions are automatically saved in a file with the file extension `.ipynb` on the current directory after a short time. If necessary, we may save it with another name or rename it later.

A.6.Using Libraries

In Python we can add new features by using *libraries*. In Python the terms *library* and *module* are interchangeable and there is no clear difference in definition. It seems that relatively large ones are often called libraries, and relatively small ones such as libraries included in libraries and self-made libraries are called modules.

To use a library, we need to import it both in an interactive mode and in a program. Importing will be explained again in the next section. Let us use `math`, which is a standard library for mathematical functions.

Interaction on the shell window of IDLE:

```
>>> from math import pi, sin, cos
>>> pi
3.141592653589793
>>> sin(pi)
1.2246467991473532e-16
>>> sin(pi/2)
1.0
>>> cos(pi/4)**2
0.5000000000000001
```

We will explain the above interactions line by line in the following Jupyter Notebook style.

[Untitled.ipynb](#)

```
In [1]: from math import pi, sin, cos
```

This is called an *import statement*, and allows us to use the names `pi`, `sin`, and `cos` defined in the libraries `math`, which represent π , \sin , and \cos in mathematics, respectively.

```
In [2]: pi
```

Out[2]: 3.141592653589793

The displayed value of π is rounded to the specified number of effective digits after the decimal point because it continues infinitely without repetition.

In [3]: `sin(pi)`

Out[3]: 1.2246467991473532e-16

The calculated answer of $\sin(\pi)$ is not exactly 0 because `pi` is rounded and contains error. `1.2246467991473532e-16` means $1.2246467991473532 \times 10^{-16}$.

In [4]: `sin(pi/2)`


Out[4]: 1.0

Calculate $\sin(\pi/2) = 1$, which is actually rounded to 1.

In [5]: `cos(pi/4)**2`

Out[5]: 0.5000000000000001

Calculate $\cos(\pi/4)^2 = 0.5$ (`**` means the power operation in Python), in which we can see some error.

If the number `n` of the prompt `In[n]` starts with 1 as `In[1]`, it means that we restart the shell of IDLE (select "Shell > Restart Shell (Ctrl+F6)" from the menu bar) or restart the kernel of the Jupyter Notebook (click the  icon on the menu bar). If we run codes continuously in the same shell/kernel, `n` increases in order.

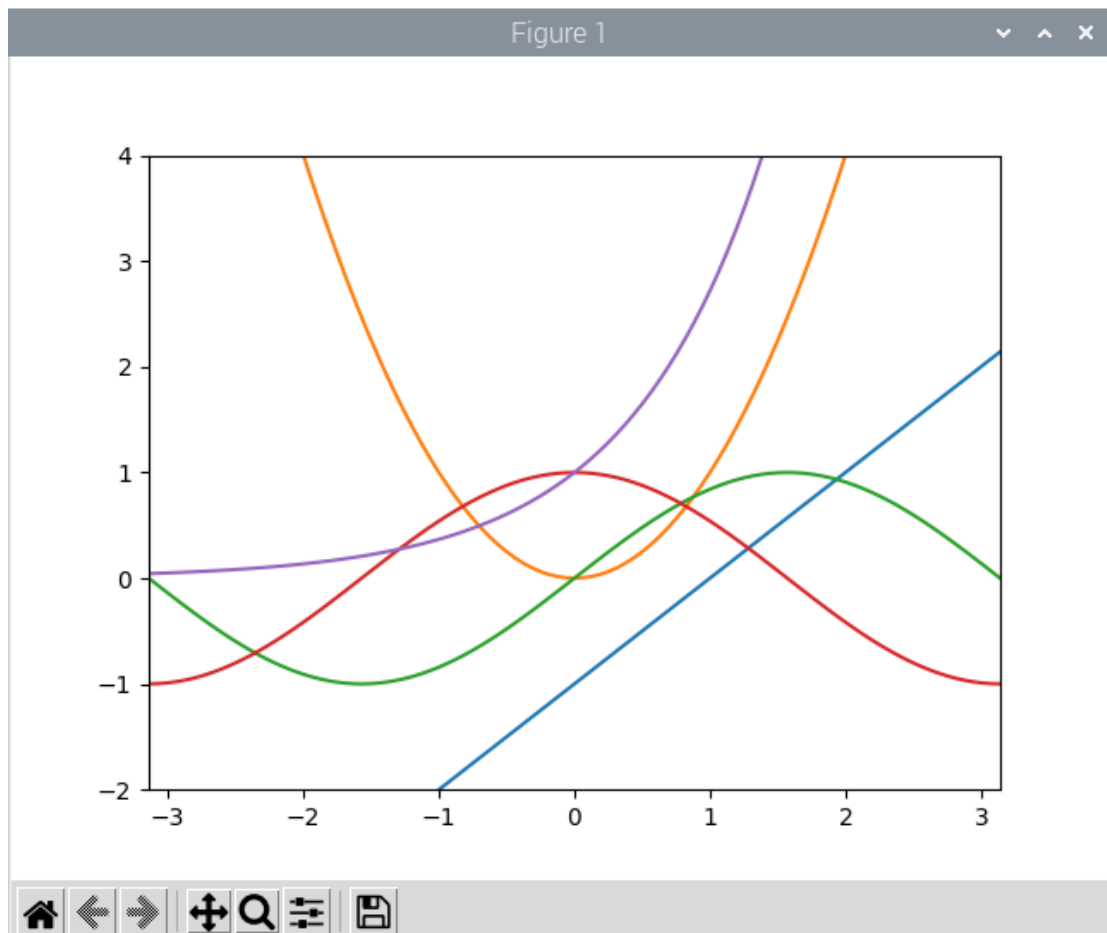
Let us use the external libraries NumPy and Matplotlib. NumPy is a library that supports numerical calculations of vectors and matrices, which are the main casts of this book. Matplotlib is a library used to display graphs of functions. Let us create a new file called `example02.py` in the same way as `example01.py` with IDLE, and run it.

Program on the edit window of IDLE `example02.py`

```
from numpy import linspace, pi, sin, cos, exp
import matplotlib.pyplot as plt

x = linspace(-pi, pi, 101)
plt.xlim(-pi, pi), plt.ylim(-2, 4)
for y in [x - 1, x**2, sin(x), cos(x), exp(x)]:
    plt.plot(x, y)
plt.show()
```

When we run this program with IDLE, another window will open on which the following graph is displayed.



If we want to save this graph as an image file, replace `plt.show()` in Line 8 by `plt.savefig('example02.png')`, where the `example02` can be replaced by another file name as we like, and `png` can be replaced by another file extension such as `jpg`, `pdf` etc.

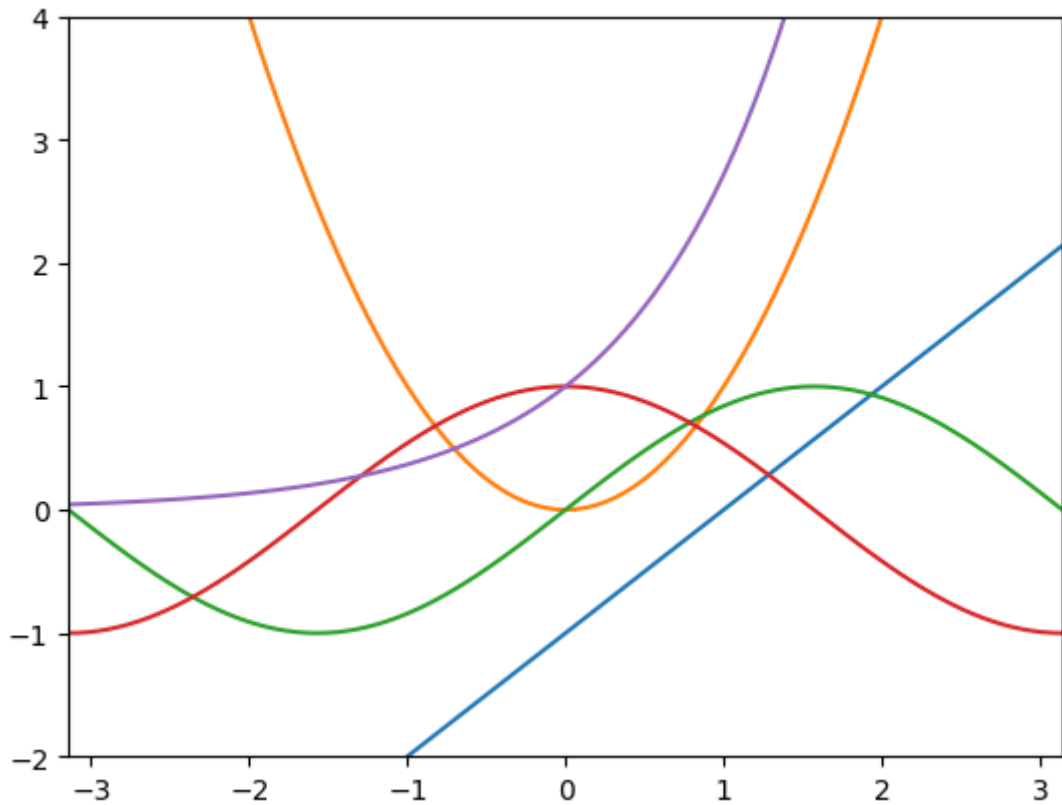
If we are using Jupyter Notebook, create a new notebook named `example02.ipynb`.

We can write down this program in a cell as follows. The function `plt.show()` of Line 8 of `example02.py` above is not needed (though it causes no error) in the notebook.

Program on the Jupyter Notebook [example02.ipynb](#)

```
In [1]: from numpy import linspace, pi, sin, cos, exp
import matplotlib.pyplot as plt

x = linspace(-pi, pi, 101)
plt.xlim(-pi, pi), plt.ylim(-2, 4)
for y in [x - 1, x**2, sin(x), cos(x), exp(x)]:
    plt.plot(x, y)
```



In the Jupyter Notebook, we can write the above program across multiple cells as shown below. [Untitled1.ipynb](#)

```
In [2]: from numpy import linspace, pi, sin, cos, exp
```

Import `linspace`, `pi`, `sin`, `cos` and `exp` defined in NumPy. Mathematical functions and constants are also defined in the standard library `math`, but in this book we mainly use those defined in NumPy.

```
In [3]: import matplotlib.pyplot as plt
```

Import the library `pyplot` included in the library Matplotlib. The library may have a tree structure. Declaring in this way allows all names defined in `matplotlib.pyplot` to be referenced with prefix `plt`.

```
In [4]: x = linspace(-pi, pi, 101)
```

Python variable `x` refers the arithmetic sequence consisting of 101 points

$$-\pi = x_0, x_1, \dots, x_{100} = \pi$$

including both ends of the real interval from $-\pi$ to π divided into 100 equal parts. Let us look at what `x` is like.

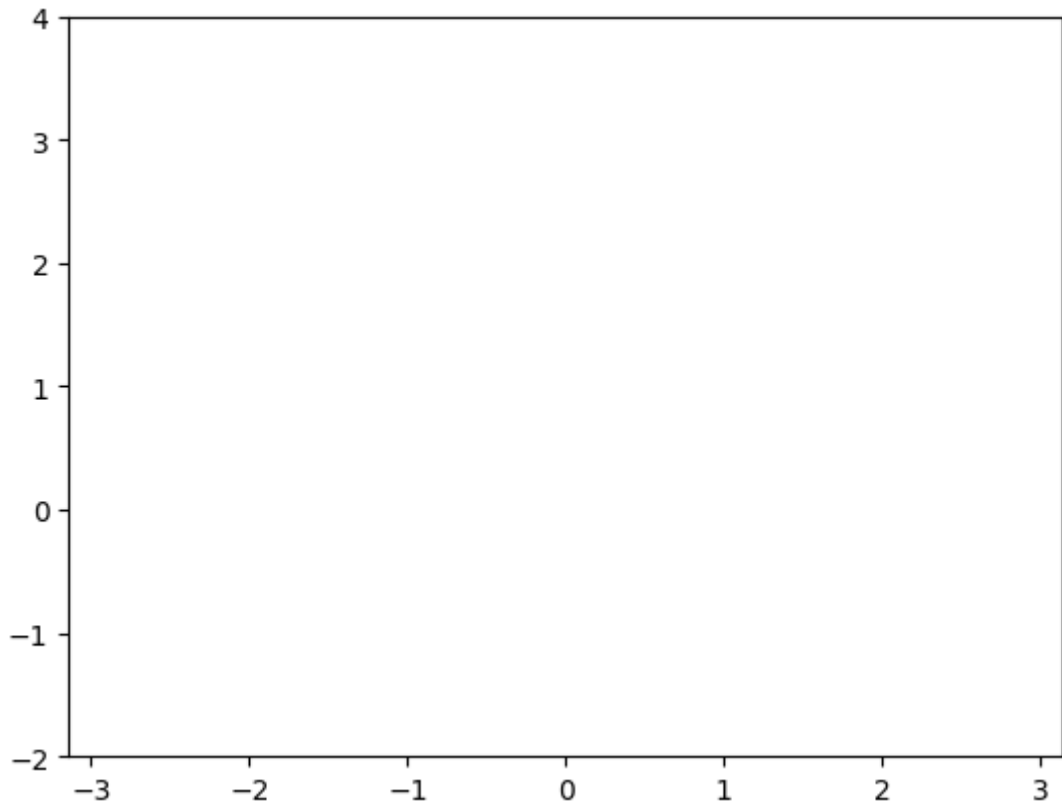
```
In [5]: x
```

```
Out[5]: array([-3.14159265e+00, -3.07876080e+00, -3.01592895e+00, -2.95309709e+0
0,
-2.89026524e+00, -2.82743339e+00, -2.76460154e+00, -2.70176968e+0
0,
-2.63893783e+00, -2.57610598e+00, -2.51327412e+00, -2.45044227e+0
0,
-2.38761042e+00, -2.32477856e+00, -2.26194671e+00, -2.19911486e+0
0,
-2.13628300e+00, -2.07345115e+00, -2.01061930e+00, -1.94778745e+0
0,
-1.88495559e+00, -1.82212374e+00, -1.75929189e+00, -1.69646003e+0
0,
-1.63362818e+00, -1.57079633e+00, -1.50796447e+00, -1.44513262e+0
0,
-1.38230077e+00, -1.31946891e+00, -1.25663706e+00, -1.19380521e+0
0,
-1.13097336e+00, -1.06814150e+00, -1.00530965e+00, -9.42477796e-0
1,
-8.79645943e-01, -8.16814090e-01, -7.53982237e-01, -6.91150384e-0
1,
-6.28318531e-01, -5.65486678e-01, -5.02654825e-01, -4.39822972e-0
1,
-3.76991118e-01, -3.14159265e-01, -2.51327412e-01, -1.88495559e-0
1,
-1.25663706e-01, -6.28318531e-02, 4.44089210e-16, 6.28318531e-0
2,
1.25663706e-01, 1.88495559e-01, 2.51327412e-01, 3.14159265e-0
1,
3.76991118e-01, 4.39822972e-01, 5.02654825e-01, 5.65486678e-0
1,
6.28318531e-01, 6.91150384e-01, 7.53982237e-01, 8.16814090e-0
1,
8.79645943e-01, 9.42477796e-01, 1.00530965e+00, 1.06814150e+0
0,
1.13097336e+00, 1.19380521e+00, 1.25663706e+00, 1.31946891e+0
0,
1.38230077e+00, 1.44513262e+00, 1.50796447e+00, 1.57079633e+0
0,
1.63362818e+00, 1.69646003e+00, 1.75929189e+00, 1.82212374e+0
0,
1.88495559e+00, 1.94778745e+00, 2.01061930e+00, 2.07345115e+0
0,
2.13628300e+00, 2.19911486e+00, 2.26194671e+00, 2.32477856e+0
0,
2.38761042e+00, 2.45044227e+00, 2.51327412e+00, 2.57610598e+0
0,
2.63893783e+00, 2.70176968e+00, 2.76460154e+00, 2.82743339e+0
0,
2.89026524e+00, 2.95309709e+00, 3.01592895e+00, 3.07876080e+0
0,
3.14159265e+00])
```

These are not stored in the computer as they are. Each number is stored as a *floating point binary number*.

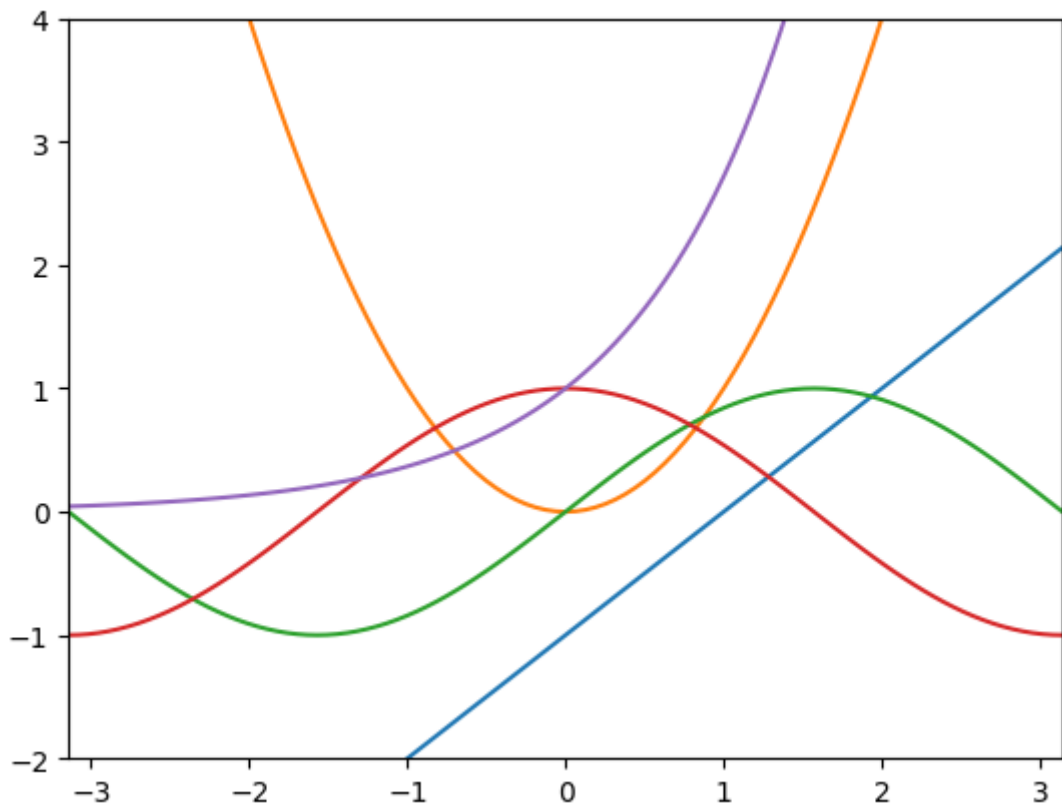
```
In [5]: plt.xlim(-pi, pi), plt.ylim(-2, 4)
```

```
Out[5]: ((-3.141592653589793, 3.141592653589793), (-2.0, 4.0))
```



Functions `plt.xlim(-pi, pi)` and `plt.ylim(-2, 4)` specify the drawing range of the x -axis and y -axis of the graph. Functions can be written on one line, separated by commas.

```
In [6]: plt.xlim(-pi, pi), plt.ylim(-2, 4)
for y in [x - 1, x**2, sin(x), cos(x), exp(x)]:
    plt.plot(x, y)
```



Display the graphs of the four functions $y = x - 1$, $y = x^2$, $y = \sin x$, $y = \cos x$ and

$y = \exp(x)$, where `x` is an object affiliated with the class `array` (precisely `ndarray`). Python classes and objects are discussed in the next chapter. Because `x` is an array, for example, `y = x**2` expresses the sequence y_n with

$$y_0 = x_0^2, y_1 = x_1^2, \dots, y_{100} = x_{100}^2,$$

while, `plt.plot(x, y)` expresses a line graph connecting 100 points

$$(x_0, y_0), (x_1, y_1), \dots, (x_{100}, y_{100}).$$

Next, let us solve an equation using SymPy, an external library for symbolic calculation.

Program: [example03.py](#) / [example03.ipynb](#)

```
In [1]: import sympy
from sympy.abc import x, y

ans1 = sympy.solve([x + 2*y - 1, 4*x + 5*y - 2])
print(ans1)
ans2 = sympy.solve([x**2 + x + 1])
print(ans2)
ans3 = sympy.solve([x**2 + y**2 - 1, x - y])
print(ans3)

{x: -1/3, y: 2/3}
[{x: -1/2 - sqrt(3)*I/2}, {x: -1/2 + sqrt(3)*I/2}]
[{x: -sqrt(2)/2, y: -sqrt(2)/2}, {x: sqrt(2)/2, y: sqrt(2)/2}]
```

When using IDLE, write this program in the program editor of IDLE, save it with the file name `example03.py` (the file extension `.py` is automatically added) and run it.

When using Jupyter Notebook, create a new notebook, name it `example03.ipynb` (the file extension `.ipynb` is automatically added), write the program in the first cell and press Shift+Enter to run it.

Line 1: By importing in this way, all the names defined in SymPy can be used with prefix `sympy`. The name `solve` defined in SymPy is used as `sympy.solve`.

Line 2: Import and use symbols `x` and `y` for unknowns in the equation. These are different from Python variables.

Lines 4, 5: Solve the simultaneous equations. Pass the left-hand side (LHS) of the equation $LHS = 0$ enclosing with braces `[]` to function `solve`.

Lines 6, 7: Solve the quadratic equation.

Lines 8, 9: Solve the simultaneous quadratic equations. Here, we are looking for the intersection of a circle and a straight line.

When using IDLE, results are printed on the shell window. When using Jupyter Notebook, results are printed right after the input cell. Solutions are given as expressions using fractions and radicals instead of numerical values, and are expressed in a format called a dictionary that uses variable symbols as keys. Also, when there are multiple

solutions, they are expressed in a format called a list whose elements are dictionaries. Python lists and Dictionaries will be explained in detail in Chapter 1.

In the shell or in the notebook, after running the program, we can refer names defined in it.

In [2]: ans1

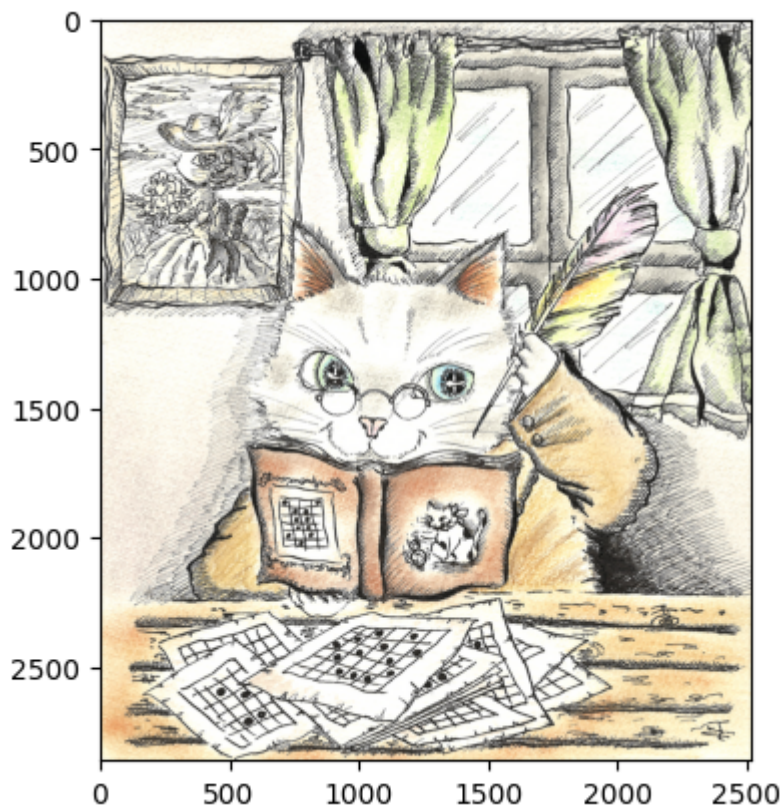
Out[2]: {x: -1/3, y: 2/3}

Next, we use the external library PIL to directly manipulate the contents of image files. Let us convert the original color image as shown in Figure(left) to grayscale and reduce the size for use in some of the experiments in this book. Here, it is assumed that the file name is `mypict.jpg`.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
import PIL.Image as Img

im0 = Img.open('A.6/mypict.jpg')
im1 = np.asarray(im0)
plt.imshow(im1)
```

Out[1]: <matplotlib.image.AxesImage at 0x7f83297f90>



You can use a photo image taken with your mobile phone or downloaded from the internet. In that case, replace the file name with that of yours. Most image file formats such as `jpg / png` and others are available. If the image is too large or you want to use only part of it, use an application GIMP mentioned later in this chapter to reduce or crop it.

Program: `mypict.py`

```
In [1]: import PIL.Image as Img

im0 = Img.open('mypict.png')
print(im0.size, im0.mode)
im1 = im0.convert('L')
im1.thumbnail((100, 100))
print(im1.size, im1.mode)
im1.save('mypict1.jpg')
```

```
(2519, 2860) RGBA
(88, 100) L
```

Line 1: Import a module `Image` of the library `PIL`, and refer to it as a name of `Img`.

Line 3: Load the image file `mypict.jpg`. A part of a picture painted by Uemura Shoen (Japan, 1875 -- 1949).} and name it `im0`. The format of the image file is automatically determined. The image file should be in the same folder where the program is, but if it is in a different folder, specify the path, such as `photos/mypict.jpg`.

Line 4: Display the size and color information of image `im0`.

Line 5: From `im0`, create a new image `im1` with the color information converted to grayscale.

Line 6: Reduce the size of the image `im1` to 100×100 pixels. It is converted to fit in the specified size without changing the aspect ratio of the image.

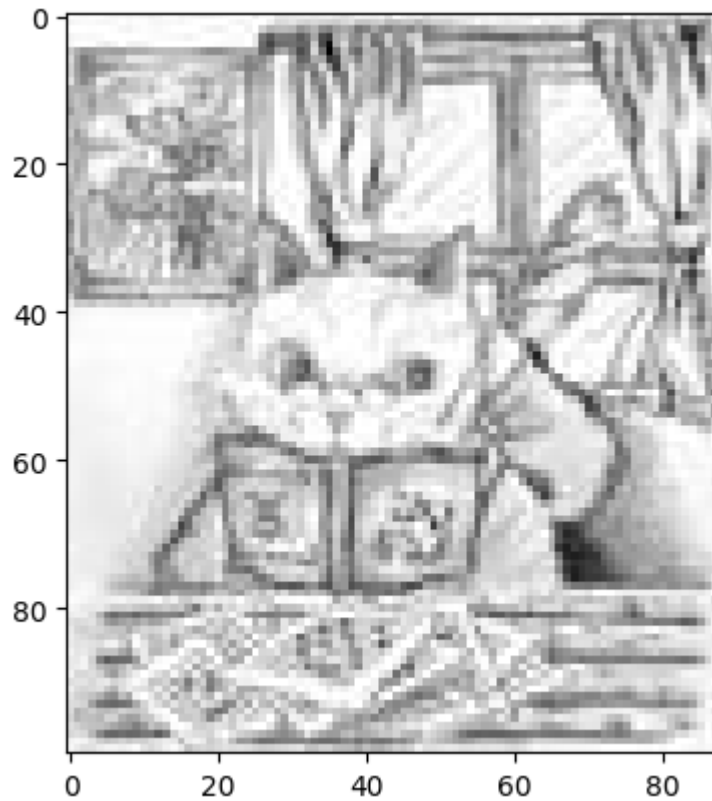
Line 7: Display the size and color information of image `im1`.

Line 8: Save image `im1` in the specified format with a file extension. The converted image of `mypict1.jpg` is shown in below.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
import PIL.Image as Img

im0 = Img.open('A.6/mypict1.jpg')
im1 = np.asarray(im0)
plt.imshow(im1, cmap='gray')
```

```
Out[1]: <matplotlib.image.AxesImage at 0x7f4bdc14d0>
```



VPython is a great library for easily generating 3D images and animations.

Anaconda users need to install VPython before using it first. Type the following command from *Anaconda PowerShell Prompt* of Windows or the terminal of macOS

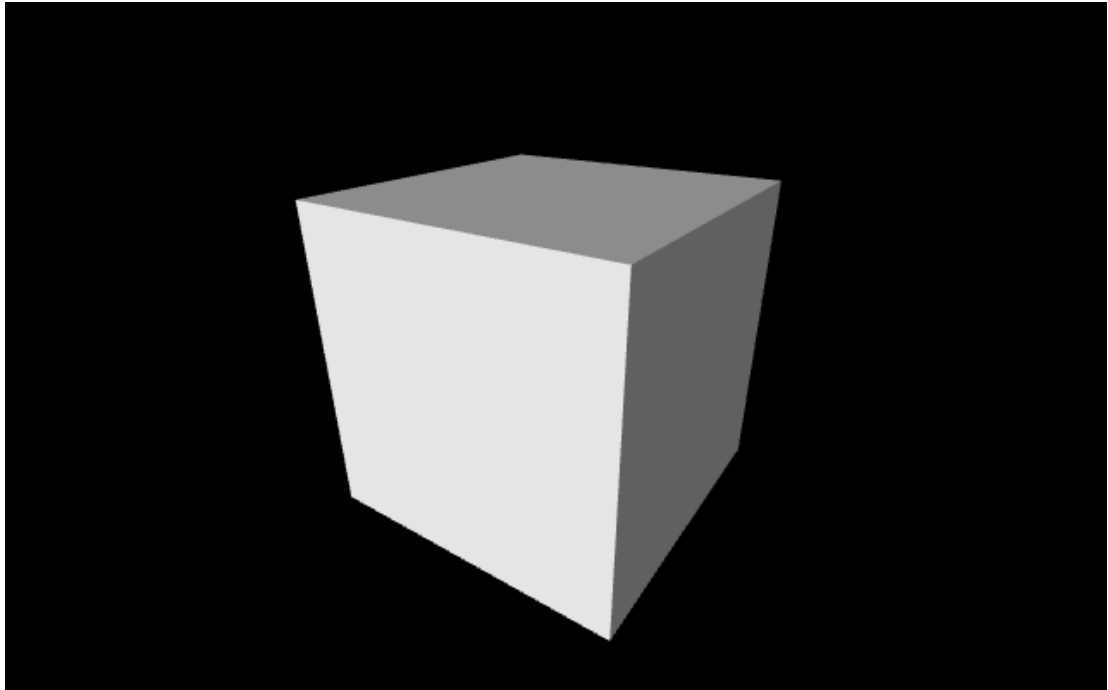
```
conda install -c vpython
```

Let us use this in an interactive mode. While using IDLE, restart it, and while using a Jupyter Notebook, open a new notebook. [vpython.ipynb](https://vpython.org/learn/intermediate/intermediate-tutorial-1)

```
In [1]: from vpython import *
```

If using IDLE, replace `In[*]` with the IDLE prompt `>>>`.

```
In [2]: B=box()
```



When `n` of `In[n]` is a consecutive number, it means that the interaction continues in the same shell or notebook. When using IDLE, the browser will open and the cube in 3D is displayed. For notebooks, it will be displayed on the same tab. We can resize it by left-clicking and dragging the sides or corners of the screen, change the viewing direction by moving the mouse while right-clicking on the screen, and move the viewpoint back and forth by clicking left and right at the same time on the screen, or by moving the mouse while holding down the center button.

```
In [3]: B.color=color.red
```

The color of the cube changes to red.

VPython also provides an environment for creating 3D animations. For example, see `newton.py` in Section 9.2.

A.7. Python Syntax

The function `f(N)` of the following program returns a list of prime numbers less than the given integer `N`. For each integer `n` with $2 \leq n < N$ we try to divide it in order by the numbers in the list `P` that stores the prime numbers found so far (the initial state of `P` is empty). If `n` is divided by some number, then it is not a prime number, otherwise, it is determined to be a prime number and appended to list `P`. This method of finding prime numbers is called the *sieve theory*.

Program: [prime.py](#) / [prime.ipynb](#)

```
In [1]: def f(N):
        P = []
        for n in range(2, N):
            q = 1
            for p in P:
```

```

        q = n % p
        if q == 0:
            break
    if q:
        P.append(n)
    return P

if __name__ == '__main__':
    P = f(20)
    print(P)

```

[2, 3, 5, 7, 11, 13, 17, 19]

We find prime numbers less than 20. White spaces at the beginning of a line are called *indentation*. Indentation in Python represents a nested block structure. The above program has a block structure as Figure.

IDLE and the Notebook will guess where indentation is needed when a line break occurs and move the cursor to the appropriate position. When we press the Enter/Return key on Line 8 and move to Line 9, the cursor automatically comes to the same level as `break` on Line 7. Then we use the BackSpace key to erase the white spaces and move the cursor to the desired position where `for` starts on Line 5.

Line 1: Up to Line 11 is one block for defining `f(N)`. This block is called a *def-block* and the first line is called a *def-statement*.

Line 2: `P` is a list that stores the prime numbers found so far, and is initially empty.

Lines 3 - 10: A block called *for-block* iterates the contents from Line 4 to Line 10. Line 3 is called *for-statement*, in which `n` is an integer called *loop counter* moving between 1 and 10.

Line 4: Set `q` to 1 initially, and if this value becomes 0 by the subsequent operation, `n` is determined not to be a prime number.

Lines 5 - 8: A block iterates the contents from Line 6 to Line 8, where `p` is a loop counter moving the elements of list `P`.

Line 6: Expression `n % p` calculates the remainder of `n` divided by `p`. Let this remainder be q .

Lines 7, 8: A block called an *if-block*. The expression to the right of `if` of *if-statement* is called a *conditional clause*, and if this condition is true, it executes the content on Line 8 of the block. The conditional clause `q == 0` means an equality and is distinct from the assignment statement `q = 0`.

We call `break` a *break statement* and it is an instruction to go out the innermost loop block in which the break statement exists. That is, break out of the loop block Line 5-8 when `q` that is the remainder of dividing `n` by `p` becomes 0.

Lines 9, 10: The if-block here is at the same indentation level as the block starting with the for-block Lines 5-8. We reach this block when we get out of the previous block either by the break statement or by the loop counter moved through all the values. In the

former case, the value of `q` is 0, and in the latter case, it is not zero. In Python, nonzero numbers have a Boolean value `True`. If the Boolean value of `q` is `True`, append the integer `n` as a new prime number to list `P`.

Line 11: This is called a *return statement*, and returns list `P` of prime numbers completed.

Lines 12, 13: We may include one or more blank lines to make the program code easier to read. In particular, when defining a function as in this example, the PEP8 coding convention recommends that we leave two lines apart, but this is not absolute. From here in this book, we leave only one line apart to save space.

Lines 14 - 16: The function is not executed just by defining it. What this program actually executes is the block starting with the if-statement on Line 14. This if statement is one of the idioms in Python. When this program is run as the main program rather than as a library (that is, instead of being imported into another program), the conditional clause of `__name__ == '__main__'` will be true and the contents of this block will be executed. There is no rule that the function definition part and the main program must be written separately in the program. If we remove Line 14 and also delete the indentation of both Line 15 and Line 16, these lines will be executed when we import this program as a module into another program. The expression `n` of `f(n)` in the def-statement is called a *parameter* (or *formal argument* of the function `f`). A parameter is a variable with the scope only in a def-block. On the other hand, `20` inside `f(20)` at Line 15, is called a *argument* of the function `f`. When `f(20)` is called, `20` is passed to the parameter `n` of `f`, and the contents of the def-block are executed for the first time. The value returned by the return statement is called a *return value*. The return value of `f(20)` is assigned to `P` on Line 15, and it is printed out on Line 16.

```
def f(N):
    P = []
    for n in range(3, N):
        q = 1
        for p in P:
            q = n % p
            if q == 0:
                break
        if q:
            P.append(n)
    return P

if __name__ == '__main__':
    P = f(20)
    print(P)
```

Let us continue to find prime numbers less than 50 in interactive mode.

In [2]: `f(50)`

Out[2]: `[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]`

Variable `P` on Line 15 is distinct from the variable `P` used in the def-block on Lines 1-11. The `P` on Line 15 has not changed even after `f(50)` was called.

```
In [3]: P
```

```
Out[3]: [2, 3, 5, 7, 11, 13, 17, 19]
```

In Python, telling the processor to use names defined in a library is called *importing*. Often the same name has different functionality depending on the library in which it is defined. There are several ways to import libraries.

The first is the following way which we used for Numpy in Program `example02`.

```
In [ ]: from numpy import linspace, pi, sin, cos, exp
```

Listing the names defined in the library `numpy`, we can use the names `linspace`, `pi`, `sin`, `cos` and `exp` listed there as they are. If it is annoying to list each name defined in the library that we want to use, here is the way we used in the VPython example in the last section.

```
In [ ]: from vpython import *
```

While useful, this can lead to unexpected rewriting of previously defined names and name battling when using several libraries at the same time. Here is a way that allows us to use all the names defined in the library avoiding such a risk. This is used in `example03` in the SymPy example above.

```
In [ ]: import sympy
```

We can use all the names defined in SymPy by prefixing them with the prefix `sympy` like `sympy.solve`. If the name of the library is too long, we can use a short cut as we did in `example02`.

```
In [ ]: import matplotlib.pyplot as plt
```

We can use `plt` for `matplotlib.pyplot`.

We can import the program `prime.py` in this section as a library. Try the following three ways explained above to import `prime` as a library.

- `from <library> import <name>1, <name>2, ..., <name>n`

```
In [ ]: from prime import f
print(f(50))
```

- `import <library>`

```
In [ ]: import prime
print(prime.f(50))
```

- `import <library> as <alias>`

```
In [ ]: import prime as pr
print(pr.f(50))
```

When we want to use `prime.ipynb` made with Jupyter Notebook as a library, convert it to `prime.py` by the following command.

```
jupyter nbconvert --to python prime.ipynb
```

Alternatively, we can open `prime.ipynb` in Jupyter Notebook and convert it with `File>Download as>Python (.py)` from the menu bar. The converted file `prime.py` will be in the Downloads folder.

A.8. JupyterLab (optional)

This section is for advanced users who are familiar with the computer environment, so can be skipped. We will not actually need JupyterLab to read this book. JupyterLab is an advanced system of Jupyter Notebook. From Anaconda Navigator, click the icon of JupyterLab, or from the command line execute the following to launch it.

```
jupyter lab
```

We may replace the above by `jupyter-lab`.

When using VPython on a notebook, better to open it from Jupyter Notebook. In order to use VPython in JupyterLab, it must be extended with Jupyter's `labextension` command. Before that, we need to have the stable version of Node.js installed from <https://nodejs.org>.

- **Windows or macOS:** Installers are available.
- **Raspberry Pi OS or Ubuntu 64-bit for Raspberry 4/400:** Binaries are available.

ARMv7 is for the former and ARMv8 the latter. Place the folder created by extracting the zip file downloaded in a suitable location we like. Change the current directory to the folder created using the `cd` command. We may delete all files there other than folders. Copy all the folders into `/usr/local/` recursively keeping the attributes by using the following command.

```
sudo cp -a * /usr/local/
Then, do labextension.
```

```
jupyter labextension install vpython
```

Depending on our environment, this command may need to be executed with the administrator privilege. Since the extension of JupyterLab by `labextension` is related to many libraries and may depend on each version of theirs, it may not work well on our environment. In this case, there is another way to install `jupyterlab-`

`python` using the `pip` command, which does not require Node.js or `labextension`. For more information, search [PyPi](#) of a repository of the Python programming language.

A.9. Other tools (supplementary)

LAT_{EX}

LAT_{EX} is a widely used system for typesetting documents containing mathematical formulas. In this book, LAT_{EX} codes may be used to make it easier to see the calculation results of SymPy. We will also use it to typeset automatically generated exercises in some chapters (e.g. Exercises 4.2 and 4.7).

Untitled.ipynb

```
In [1]: from sympy import solve, latex
        from sympy.abc import a, b, c, x
        A = solve([a*x**2+b*x+c], [x]); A
```

```
Out[1]: [(-b/(2*a) - sqrt(-4*a*c + b**2)/(2*a)),
        (-b/(2*a) + sqrt(-4*a*c + b**2)/(2*a))]
```

```
In [2]: print(latex(A[0][0]))
```

```
- \frac{b}{2 a} - \frac{\sqrt{- 4 a c + b^{2}}}{2 a}
```

When using a Jupyter Notebook, change the type of a new cell from code to Markdown. Next, copy and paste the above LAT_{EX} code into the cell and enclose it with `$` 's.

```
$- \frac{b}{2 a} - \frac{\sqrt{- 4 a c + b^{2}}}{2 a}$
```

After editing the cell, press Shift+Enter, then we have the typeset formula bellow on the cell.

$$-\frac{b}{2a} - \frac{\sqrt{-4ac + b^2}}{2a}$$

When using IDLE, we need a LAT_{EX} system installed.

- **Windows:** To use the LAT_{EX} system, install TeX Live. We can get the system from <https://www.tug.org/texlive/>.
- **macOS:** To use the LAT_{EX} system, install MacTex.

For that we recommend to use Homebrew, a package manager. For more information visit <https://brew.sh/>.

After installing Homebrew, install MacTex with the following commands.

```
brew install --cask mactex
sudo tlmgr update --self --all
```

- **Raspberry Pi OS or Ubuntu Desktop:** We can install it using the `apt` command as below.

```
sudo apt install texlive-science -y
sudo apt install texlive-latex-extra -y
sudo apt install texlive-font-utils -y
sudo apt install texworks -y
```

If installed, we can use an application called TeXworks, which is an integrated environment editing editing a LAT_EX source code and typesetting it.

```
\documentclass{standalone}
\usepackage{amssymb, amsmath}

\begin{document}

$- \frac{b}{2a} - \frac{\sqrt{-4ac + b^2}}{2a}$

\end{document}
```

Typesetting the above, we will have a pdf file of the cropped image of the formula $-\frac{b}{2a} - \frac{\sqrt{-4ac+b^2}}{2a}$. Use this as a template by replacing the code between `$` and `$` with another LAT_EX code.

You may want to upload a formula typeset in LAT_EX to your web home page. In that case, the easiest way is to convert the Jupyter Notebook page as it is to an HTML file. Alternatively, you can convert the pdf file into the other image format using GIMP described below and include it directly into the HTML

GIMP

The output of LAT_EX described above will be a pdf file, but if you like to convert it to another image format for inserting into another document, you may use GIMP. It is a cross-platform editor for image manipulation and converting the file formats. We can install it as follows depending on the environment.

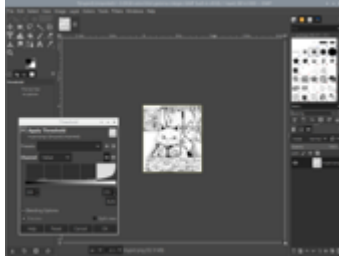
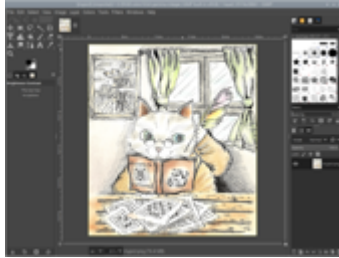
- **Windows:** Get the installer from <https://www.gimp.org/downloads/>.
- **macOS:**

```
brew install --cask gimp
```

- **Raspberry Pi OS or Ubuntu Desktop:**

```
sudo apt install gimp -y
```

By using GIMP, you can retouch your favorite photos or any images obtained from the internet. For example, we can get the same result through the GUI screen by grayscaling or reducing the image as done in Section A.4.



PyX

A Python library PyX allows us to draw a figure containing mathematical formulas typeset by *L^AT_EX*.

- **Windows:**

```
pip install pyx
```

- **macOS:**

```
python3 -m pip install pyx
```

- **Raspberry Pi OS or Ubuntu Desktop:**

```
sudo apt install python3-pyx -y
```

Alternatively, we can also install it by the `pip` command. Depending on our environment, add `sudo` before `pip`.

```
python3 -m pip install pyx
```

Some of the figures in this book are drawn by PyX. Here is the program to create Figures 1.2(right) of the complex plane in Section 1.3 for example.

Program: [comp.py](#)

```
In [1]: from pyx import *

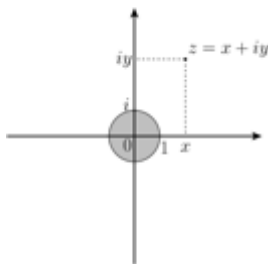
C = canvas.canvas()
text.set(text.LatexRunner)
text.preamble(r'\usepackage{amsmath}')
text.preamble(r'\usepackage{amsfonts}')
text.preamble(r'\usepackage{amssymb}')

C.stroke(path.circle(0, 0, 1),
```

```

        [style.linewidth.thick, deco.filled([color.gray(0.75)])])
C.stroke(path.line(-5, 0, 5, 0),
        [style.linewidth.THick, deco.earrow.Large])
C.stroke(path.line(0, -5, 0, 5),
        [style.linewidth.THick, deco.earrow.Large])
C.stroke(path.line(1, -0.1, 1, 0.1))
C.stroke(path.line(-0.1, 1, 0.1, 1))
C.text(-0.1, -0.1, r"\huge 0" ,
        [text.halign.right, text.valign.top])
C.text(1, -0.2, r"\huge 1" ,
        [text.halign.left, text.valign.top])
C.text(-0.2, 1, r"\huge $i$" ,
        [text.halign.right, text.valign.bottom])
C.stroke(path.circle(2, 3, 0.05),
        [deco.filled([color.grey.black])])
C.text(2.1, 3.1, r"\huge $z=x+iy$" ,
        [text.halign.left, text.valign.bottom])
C.stroke(path.line(2, 3, 2, 0),
        [style.linewidth.thick, style.linestyle.dashed])
C.stroke(path.line(2, 3, 0, 3),
        [style.linewidth.thick, style.linestyle.dashed])
C.text(2, -0.3, r"\huge $x$" ,
        [text.halign.center, text.valign.top])
C.text(-0.1, 3, r"\huge $iy$" ,
        [text.halign.right, text.valign.middle])
C.writePDFfile('comp.pdf')

```



Graphviz

Graphs showing the dependencies in some items can be created with a Python wrapper for an application called Graphviz.

- **Windows:**

```
conda install python-graphviz
```

- **macOS:**

```
brew install graphviz
python3 -m pip install graphviz
```

- **Raspberry Pi OS or Ubuntu Desktop:**

```
sudo apt install python3-graphviz -y
```

If we like to use the `pip` command, we must install Graphviz independently with the `apt` command. Depending on our environment, add `sudo` before `pip`.

```
sudo apt install graphviz -y
python3 -m pip install graphviz
```

This is the source code drawing the diagram in Preface.

Program: [book_diagram.py](#)

```
In [1]: from graphviz import Digraph

G = Digraph(format='pdf')
G.attr('node', shape='box')

G.node('Apx', 'Appendix', style='dashed', shape='oval')
G.node('ch1', 'Chapter 1: Mathematics and Python')
G.node('ch2', 'Chapter 2: Linear Space and Linear Mapping')
G.node('ch3', 'Chapter 3: Basis and Dimension')
G.node('ch4', 'Chapter 4: Matrix')
G.node('ch5', 'Chapter 5: Elementary Operation and Matrix Invariants')
G.node('ch6', 'Chapter 6: Inner Product and Fourier Expansion')
G.node('ch7', 'Chapter 7: Eigenvalue and Eigenvector')
G.node('ch8', 'Chapter 8: Jordan Normal Form and Spectrum')
G.node('ch9', 'Chapter 9: Dynamical System')
G.node('ch10', 'Chapter 10: Applications and Development of Linear Algebra')

G.edge('Apx', 'ch1')
G.edge('ch1', 'ch2')
G.edge('ch2', 'ch3')
G.edge('ch2', 'ch6')
G.edge('ch3', 'ch4')
G.edge('ch4', 'ch5')
G.edge('ch4', 'ch6')
G.edge('ch5', 'ch7')
G.edge('ch5', 'ch10')
G.edge('ch6', 'ch7')
G.edge('ch6', 'ch8')
G.edge('ch6', 'ch10')
G.edge('ch7', 'ch8')
G.edge('ch7', 'ch10')
G.edge('ch8', 'ch9')

G.render('diagram')
```

Out[1]: '.A.9/diagram.pdf'



Audacity

In Chapters 2 and 6, we will treat audio data.

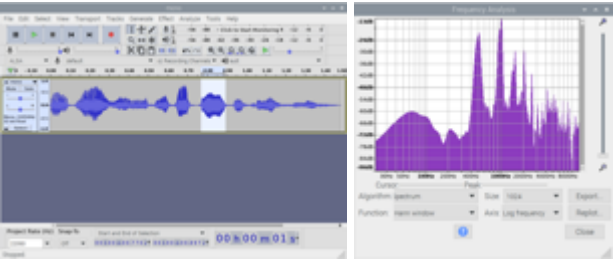
- **Windows:** Get the installer from <https://www.audacityteam.org/download/>.
- **macOS:**

```
brew install --cask audacity
```

- **Raspberry Pi OS or Ubuntu Desktop:**

```
sudo apt install audacity -y
```

Audacity is a GUI tool which can be used as recorder, editor and analyzer for audio data.



In []: