

In this chapter, we survey the notions of propositions, real and complex numbers, sets, and mappings (functions) that are the basics of mathematics needed to study linear algebra. We will also learn how these concepts are expressed in Python. Let us learn mathematics in a practical way by using Python.

1.1 Propositional Logic

In mathematics, a statement that describes a “matter”,¹ which is subject to judgment as to whether it is *true* or *false*, is called a *proposition*. True/false is called the *truth value* of a proposition. The truth value of the proposition “2 is a prime number” is true, and the value of “6 is a prime number” is false.

Suppose that P and Q are propositions. Then, $\neg P$ (not P), $P \wedge Q$ (P and Q), $P \vee Q$ (P or Q), $P \rightarrow Q$ (P implies Q) and $P \leftrightarrow Q$ (P if and only if Q) are also propositions whose truth values are determined by tables shown in Table 1.1 called *truth tables*.

In the context of Python, a truth value is called a *Boolean value*, and the values true and false are expressed by literals² True and False, respectively. In Python $\neg P$, $P \wedge Q$ and $P \vee Q$ are expressed by not P, P and Q and P or Q, respectively. Python does not provide logical operators for \rightarrow and \leftrightarrow , but since True/False is represented by an integer 1/0 in Python, the inequality sign $<=$ and equality sign $==$ can be used for \rightarrow and \leftrightarrow , respectively. Python treats expressions in Boolean values (also Boolean values themselves) as objects.


We can confirm the truth value tables in Table 1.1 by the following simple Python codes:

Program: table1.py

```
In [1]: 1 | for P in [True, False]:
        2 |     print(P, not P)
```



```
True False
False True
```

In this book, the icon  represents the output by the print function in interactive mode.

¹ A matter is distinguished from a “thing” that is an element of a set stated in Sect. 1.3.

² A literal is a constant value written in Python code. In addition to True and False, the real number 3.14 and string ‘Python’ are literals, for example.

Table 1.1 The truth value tables

| P | $\neg P$ | P | Q | $P \wedge Q$ | $P \vee Q$ | $P \rightarrow Q$ | $P \leftrightarrow Q$ |
|-------|----------|-------|-------|--------------|------------|-------------------|-----------------------|
| true | false | true | true | true | true | true | true |
| true | false | true | false | false | true | false | false |
| false | true | false | true | false | true | true | false |
| false | true | false | false | false | false | true | true |

Program: table2.py

```
In [1]: 1 | for P in [True, False]:
        2 |     for Q in [True, False]:
        3 |         print(P, Q, P and Q, P or Q, P <= Q, P == Q)
```

```
True True True True True True
True False False True False False
False True False True True False
False False False False True True
```

If the truth value tables of P and Q coincide, we write $P \leftrightarrow Q$. In this case, we say P is *equivalent* to Q or P holds *if and only if* Q holds. For example, by looking into the table above, we see $P \wedge Q \Leftrightarrow Q \wedge P$, $P \vee Q \Leftrightarrow Q \vee P$, and $P \leftrightarrow Q \Leftrightarrow Q \leftrightarrow P$.

Exercise 1.1 Show the equivalences

$$P \rightarrow Q \Leftrightarrow \neg P \vee Q \quad \text{and} \quad P \leftrightarrow Q \Leftrightarrow (P \rightarrow Q) \wedge (Q \rightarrow P)$$

by making the truth value tables for them. Moreover, confirm the equivalences by Python.

Exercise 1.2 Prove the associative laws

$$(P \wedge Q) \wedge R \Leftrightarrow P \wedge (Q \wedge R), \quad (P \vee Q) \vee R \Leftrightarrow P \vee (Q \vee R)$$

and the distributive laws

$$P \wedge (Q \vee R) \Leftrightarrow (P \wedge Q) \vee (P \wedge R), \quad P \vee (Q \wedge R) \Leftrightarrow (P \vee Q) \wedge (P \vee R).$$

A proposition including a variable x , in which the truth value changes depending on the value of x such as “ x is an integer greater than 2” or “ x is a prime number”, is called a *propositional function*. Let $P(x)$ and $Q(x)$ be propositional functions. The expression

$$P(x) \Rightarrow Q(x)$$

means that the truth value of $Q(x)$ is true whenever the truth value of $P(x)$ is true regardless of the value of variable x , and we read it as $P(x)$ *implies* $Q(x)$. This also means that the propositional function $P(x) \rightarrow Q(x)$ is always true regardless of the value of x . For example, we use it to say

$$x \text{ is a multiple of } 6 \Rightarrow x \text{ is even.}$$

On the other hand, $P(x) \Leftrightarrow Q(x)$ means that the truth values of $P(x)$ and $Q(x)$ always match for any value of the variable x . This means that $P(x) \Leftrightarrow Q(x)$ is true for any value of x , or equivalently both $P(x) \Rightarrow Q(x)$ and $Q(x) \Rightarrow P(x)$ hold. For example,

$$x \text{ is even and a multiple of } 3 \Leftrightarrow x \text{ is a multiple of } 6.$$

Let $P(n)$ be a propositional function in natural number n . If $P(1)$ is true and $P(n) \Rightarrow P(n+1)$ holds, then we conclude that $P(n)$ is true for every natural number n . This reasoning is called a *mathematical induction*.

1.2 Numbers

The following algebraic properties for real numbers are well known and hold even for complex numbers³:

$$\begin{aligned} \text{R1. } x + y &= y + x, & \text{R2. } (x + y) + z &= x + (y + z), & \text{R3. } x + 0 &= x, \\ \text{R4. } x + (-x) &= 0, & \text{R5. } xy &= yx, & \text{R6. } (xy)z &= x(yz), \\ \text{R7. } 1x &= x, & \text{R8. } x \neq 0 \Rightarrow \frac{1}{x} \cdot x &= 1, & \text{R9. } x(y + z) &= xy + xz. \end{aligned}$$

Exercise 1.3 Deduce the equalities $0x = x0 = 0$ from properties R1–R9.

For a complex number $z = x + iy$ where i is the imaginary unit $\sqrt{-1}$ and x and y are real numbers, we call x and y the *real part* and the *imaginary part* of z and denote them by $x = \operatorname{Re} z$ and $y = \operatorname{Im} z$, respectively. The *absolute value* $|z|$ and the *complex conjugate* \bar{z} of z are defined by

$$|z| \stackrel{\text{def}}{=} \sqrt{x^2 + y^2} \quad \text{and} \quad \bar{z} \stackrel{\text{def}}{=} x - iy,$$

where the notation $\stackrel{\text{def}}{=}$ is used to mean that the left-hand side is defined by the expression of the right-hand side. With these notations the following properties are satisfied:

$$\begin{aligned} \text{Z1. } |z|^2 &= z\bar{z}, & \text{Z2. } |z_1 z_2| &= |z_1| |z_2|, & \text{Z3. } z = \bar{z} &\Leftrightarrow z \text{ is real,} \\ \text{Z4. } \overline{z_1 + z_2} &= \bar{z}_1 + \bar{z}_2, & \text{Z5. } \overline{z_1 \cdot z_2} &= \bar{z}_1 \cdot \bar{z}_2, & \text{Z6. } |z_1 + z_2| &\leq |z_1| + |z_2|. \end{aligned}$$

Exercise 1.4 Prove properties Z1–Z6 above from properties R1–R9 of real numbers.

Every complex number z has an expression

$$z = |z| (\cos \theta + i \sin \theta)$$

called the *polar representation*, where θ is a real number called the *argument* of z . When $z = 0$, we do not define its argument. We can find in a textbook of elementary analysis the Maclaurin expansions

³ We can deduce these properties from the Peano axioms of natural numbers. It is a long journey, starting with proving the properties of natural numbers and extending them to the cases of integers, rational numbers, real numbers, and complex numbers. We omit it here.

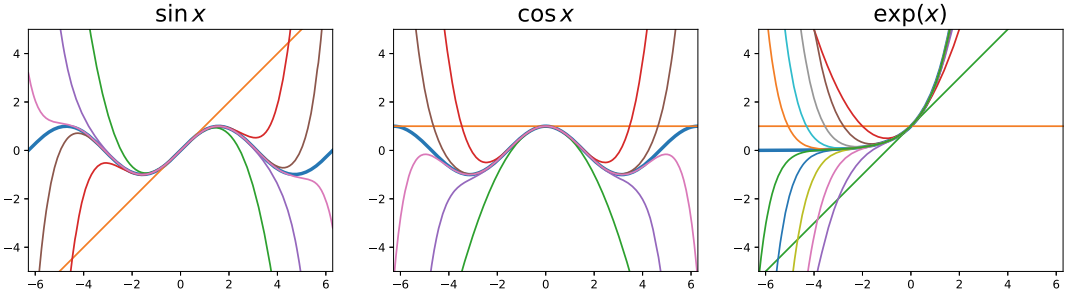


Fig. 1.1 Behaviors of convergence for Maclaurin's expansions of $\sin x$, $\cos x$, e^x

$$\begin{aligned}\sin x &= \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots, \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots, \\ e^x &= 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots\end{aligned}$$

of the functions $\sin x$, $\cos x$ and e^x (Fig. 1.1).⁴ Substitute x by $i\theta$, replace i^2 with -1 , rearrange the order of additions in the last expression, and compare to the first two expressions; then we obtain

$$e^{i\theta} = \cos \theta + i \sin \theta.$$

This is the well-known *Euler's formula*, and from this, the polar representation of z is expressed by

$$z = |z| e^{i\theta}.$$

Exercise 1.5 Prove the following formulas:

- (1) $e^{i(\theta_1+\theta_2)} = e^{i\theta_1} e^{i\theta_2}$ for any $\theta_1, \theta_2 \in \mathbb{R}$. (Hint: use the trigonometric addition theorem.)
- (2) $(e^{i\theta})^n = e^{in\theta}$ (**de Moivre's theorem**) for any natural number n . (Hint: use mathematical induction.)

Exercise 1.6 Using Python and its library Matplotlib, draw the graphs in Fig. 1.1.

Let $z = ae^{i\theta}$ with $a > 0$ and $2\pi > \theta \geq 0$. For a natural number n , let

$$w = \sqrt[n]{a} e^{\frac{i\theta}{n}},$$

then by de Moivre's theorem, we have $w^n = z$. We call this w the n -root of z and denote it by $\sqrt[n]{z}$.

Python can handle integers and real numbers. They are stored differently in the computer memory and are distinguished as an `int` class object and as a `float` class object. The ranges of numbers that can be handled are also limited. For now, we do not need to be so careful about these differences and limitations. We will explain these when there are cases where we should be careful.

⁴The reader is advised to see a textbook on analysis for a discussion of convergence of these series.

Complex numbers are also available in Python by default. Using `j`, the imaginary unit is expressed by `1j` or `1.0j`.⁵ The following is a dialogue to find the real part, the imaginary part, the absolute value, and the complex conjugate of $x = 1 + 2i$.

```
In [1]: x = 1 + 2j
        x.real, x.imag, abs(x), x.conjugate()
```

```
Out[1]: (1.0, 2.0, 2.23606797749979, (1-2j))
```

Moreover, putting $y = 3 + 4i$, let us calculate the sum $x + y$, the product xy , and the quotient x/y .

```
In [2]: y = 3 + 4j
        x + y, x * y, x / y
```

```
Out[2]: ((4+6j), (-5+10j), (0.44+0.08j))
```

Using $e = 2.718281828459045$ and $\pi = 3.141592653589793$ as approximations of the logarithmic constant and the circular constant respectively, let us calculate $e^{\pi i}$. We use the power operation `**` in Python.

```
In [3]: 2.718281828459045 ** 3.141592653589793j
```

```
Out[3]: (-1+1.2246467991473532e-16j)
```

Comparing to the real part, the imaginary part has a very small numerical error $1.2246467991473532 \times 10^{-16}$ that is almost 0. In the library NumPy of Python, names of the circular constant `pi` and the logarithmic constant `e` are defined, but these constants contain numerical errors and cause the same error.

```
In [4]: from numpy import pi, e
        e**(pi * 1j)
```

```
Out[4]: (-1+1.2246467991473532e-16j)
```

1.3 Sets

An object is a “thing” that appears in mathematics, such as numbers, points, etc. A collection of objects is also an object called a *set*. Each collected object is called an *element* of the set. A set must have a clear range of its elements. There are two standard ways to express the range. One way is to list all elements of a set. This is called the *extensional definition*. For example, the set of all prime numbers less than 10 is denoted by

$$\{2, 3, 5, 7\},$$

and the set of all planets of the solar system is

$$\{\text{Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus, Neptune}\}.$$

⁵ The use of the letter `j` for the imaginary unit comes from the practice of electrical engineers who use the letter `i` to represent electric current. Placing a literal of an `int` class object or a `float` class object just before the symbol `j`, it becomes a literal of a `complex` class object expressing a purely imaginary number.

The other way is called the *intensional definition* which uses a property that exactly characterizes the elements of a set. The two examples above are expressed in the intensional way as

$$\{x \mid x \text{ is a prime number less than } 10\}$$

and

$$\{x \mid x \text{ is a planet of the solar system}\},$$

respectively. As another example of an intensional definition,

$$\left\{ \frac{m}{n} \mid m \text{ is an integer and } n \text{ is a positive integer} \right\}$$

expresses the set of all fractions (rational numbers).

If x is an element of a set A , we denote this by $x \in A$ and say that x *belongs to* A . The negation of $x \in A$ is denoted by $x \notin A$. A set that has no element is also considered to be a set called the *empty set*, and it is represented by \emptyset or $\{\}$. A set consisting of at most a finite number of elements, including the empty set, is called a *finite set*, and a set that is not finite is said to be *infinite*. The first two examples above are both finite sets. The last example (the set of all rational numbers) is an infinite set.

Some sets which frequently appear in mathematics are marked with special symbols. An integer greater than or equal to 1 is called a *natural number*, and the set of all natural numbers is expressed by \mathbb{N} . A common but somewhat ambiguous expression⁶

$$\mathbb{N} = \{1, 2, 3, \dots, n, \dots\}.$$

is used. Whereas, \mathbb{R} and \mathbb{C} will stand for the set of all real numbers and the set of all complex numbers, respectively. It is not easy to give a rigorous definition of \mathbb{R} .⁷ As far as \mathbb{R} is defined, \mathbb{C} can be expressed by intentional definition as

$$\mathbb{C} = \{x + yi \mid x \in \mathbb{R} \text{ and } y \in \mathbb{R}\},$$

where i is the imaginary unit. All \mathbb{N} , \mathbb{R} and \mathbb{C} are infinite sets.

We can visualize \mathbb{R} and \mathbb{C} in Fig. 1.2 as the *real line* and the *complex plane*, respectively. The parts $\{z \mid |z| = 1\}$ and $\{z \mid |z| \leq 1\}$ of \mathbb{C} are called the *unit circle* and the *unit disk* of the complex plane, respectively.

Let A and B be arbitrary sets. When $x \in A \Rightarrow x \in B$,⁸ that is, all the elements of A are also elements of B , A is said to be a *subset* of B . We also say that B *contains* A and this situation is expressed as $A \subseteq B$. For example, the unit circle and disk above are subsets of \mathbb{C} . Always, $\emptyset \subseteq A$ holds for any set A . When $x \in A \Leftrightarrow x \in B$ holds, that is, both $A \subseteq B$ and $B \subseteq A$ hold, we write $A = B$ and say that A and B are equal. If $A \subseteq B$ and $A \neq B$, then A is called a *proper subset* of B and we denote this situation by $A \subsetneq B$.

The set $\{X \mid X \subseteq A\}$ of all subsets of A is called the *power set* of A and is denoted by 2^A . For example,

$$2^\emptyset = \{\emptyset\}, \quad 2^{\{1\}} = \{\emptyset, \{1\}\}, \quad 2^{\{1, 2\}} = \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}.$$

⁶ Undefined symbols “...” and variable n are used.

⁷ The reader should consult a book on real number theory.

⁸ The variable x without any mention before is called a *free variable* and includes the meaning of “any x ”.

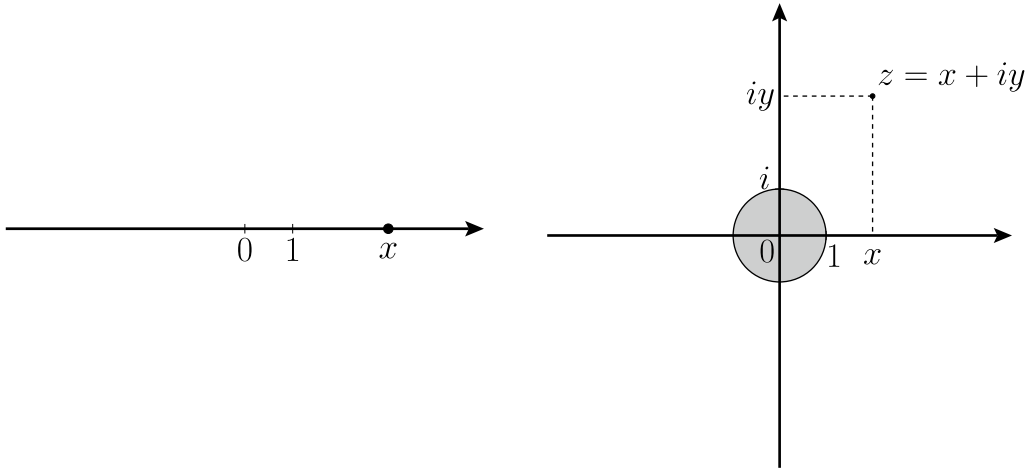


Fig. 1.2 The real line (left) and the complex plane (right) with the unit circle and disk

Exercise 1.7 Write down the extensional description of the power set $2^{\{1,2,3\}}$. Also, prove that the number of elements of the power set of a set with n elements is 2^n .

We define the following *set operations* for sets A and B :

$$\begin{aligned} A \cup B &\stackrel{\text{def}}{=} \{x \mid (x \in A) \vee (x \in B)\}, \\ A \cap B &\stackrel{\text{def}}{=} \{x \mid (x \in A) \wedge (x \in B)\}, \\ A \setminus B &\stackrel{\text{def}}{=} \{x \mid (x \in A) \wedge \neg(x \in B)\}. \end{aligned}$$

We call them the *union*, *intersection*, and *difference set*, respectively. These set operations are closely related to the operations of propositional logic. In fact,

$$\begin{aligned} x \in A \cup B &\Leftrightarrow (x \in A) \vee (x \in B), \\ x \in A \cap B &\Leftrightarrow (x \in A) \wedge (x \in B), \\ x \in A \setminus B &\Leftrightarrow (x \in A) \wedge \neg(x \in B). \end{aligned}$$

There are many elementary properties of set operations. For example, the equality

$$A \cup B = B \cup A$$

is shown as

$$x \in A \cup B \Leftrightarrow (x \in A) \vee (x \in B) \Leftrightarrow (x \in B) \vee (x \in A) \Leftrightarrow x \in B \cup A,$$

and the equality

$$(A \cup B) \cup C = A \cup (B \cup C)$$

follows from the associative law of \vee in propositional logic:

$$\begin{aligned} x \in (A \cup B) \cup C &\Leftrightarrow ((x \in A) \vee (x \in B)) \vee (x \in C) \\ &\Leftrightarrow (x \in A) \vee ((x \in B) \vee (x \in C)) \Leftrightarrow x \in A \cup (B \cup C). \end{aligned}$$

Exercise 1.8 Prove the equalities $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ and $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$. (Hint: use the distributive laws in Exercise 1.2).

When we only consider elements and subsets of some (large) set U in our mathematical discussions, U is called the *universal set*. For instance, \mathbb{N} can be the universal set when we discuss something about prime numbers, and so can be \mathbb{R} when we discuss rational and irrational numbers. If U is the universal set in some context and $P(x)$ is a propositional function, the set of $x \in U$ such that $P(x)$ is true is expressed by

$$\{x \in U \mid P(x)\}.$$

For a subset A of U we denote $U \setminus A$ by A^c and call it the *complement* of A . With a free variable x ranging over U , we have

$$x \in A^c \Leftrightarrow \neg(x \in A).$$

In Python, we can use the extensional definition of a set as in mathematics. Let us use it in interactive mode.

```
In [1]: A = {2, 3, 5, 7}; A
```

```
Out[1]: {2, 3, 5, 7}
```

```
In [2]: B = set([6, 9, 3]); B
```

```
Out[2]: {9, 3, 6}
```

```
In [3]: C = {3, 6, 9, 6, 3}; C
```

```
Out[3]: {9, 3, 6}
```

```
In [4]: set()
```

```
Out[4]: set()
```

The set $\{3, 6, 9\}$ is also expressed as `set([3, 6, 9])`. The duplicated elements in set C are removed. The empty set is expressed by `set()`. Note that `{}` has a different meaning in Python.⁹ In this book we use a semicolon `;` only for referring to a variable immediately after its definition as above in interactive mode. Sets are considered to be the same even if the orders of their elements are different.¹⁰

```
In [5]: C == {3, 6, 9}
```

```
Out[5]: True
```

⁹ For historical reasons, `{}` means the dictionary with empty entries. Dictionaries will be explained in Sect. 1.6.

¹⁰ Data are assigned to computer memory through hash functions, and so the order of assignment of elements in a set depends on them.

Use `in` for the symbol \in of belonging.

```
In [6]: 2 in A
```

```
Out[6]: True
```

```
In [7]: 2 in B
```

```
Out[7]: False
```

Use `|` and `&` for the union \cup and the intersection \cap , respectively.

```
In [8]: A | B
```

```
Out[8]: {2, 3, 5, 6, 7, 9}
```

```
In [9]: A & B
```

```
Out[9]: {3}
```

The set inclusion between sets X and Y is expressed as `X.issubset(Y)`.

```
In [10]: (A & B).issubset(A)
```

```
Out[10]: True
```

```
In [11]: A.issubset(A | B)
```

```
Out[11]: True
```

Here, `issubset` is said to be a *method*¹¹ of `set` class. Both union and intersection have also method expressions of `set` class.

```
In [12]: A | B == A.union(B)
```

```
Out[12]: True
```

```
In [13]: A & B == A.intersection(B)
```

```
Out[13]: True
```

A set is said to be an object of `set` class in Python. Objects of `set` class are limited to those with a finite number of elements. In mathematics, we can think of the set of all prime numbers, but it is impossible to express it as a `set` class object. As long as it is a finite set, Python has an expression of a `set` class object similar to the intensional definition of a set in mathematics.

```
In [14]: {x for x in range(2, 10) if all([x%n for n in range(2, x)])}
```

```
Out[14]: {2, 3, 5, 7}
```

¹¹ We will explain what methods are in Sect. 1.6.

This means the set of all integers x with $2 \leq x < 10$ not divisible by any integer n with $2 \leq n < x$ (that is, prime numbers). In Python, such an expression is called the *inline for sentence*.

There are subsets of \mathbb{R} called *intervals*:

$$\begin{aligned} [a, b] &\stackrel{\text{def}}{=} \{x \mid a \leq x \leq b\}, & (a, b) &\stackrel{\text{def}}{=} \{x \mid a < x < b\}, \\ (a, b] &\stackrel{\text{def}}{=} \{x \mid a < x \leq b\}, & [a, b) &\stackrel{\text{def}}{=} \{x \mid a \leq x < b\}, \end{aligned}$$

for $a, b \in \mathbb{R}$ with $a < b$. They are called a *closed*, *open*, *open–closed*, and *closed–open* interval, respectively. All of them are called *finite intervals*. *Infinite intervals* are

$$\begin{aligned} (-\infty, a) &\stackrel{\text{def}}{=} \{x \mid x < a\}, & [a, \infty) &\stackrel{\text{def}}{=} \{x \mid a \leq x\}, \\ (-\infty, a] &\stackrel{\text{def}}{=} \{x \mid x \leq a\}, & (a, \infty) &\stackrel{\text{def}}{=} \{x \mid a < x\}, \\ (-\infty, \infty) &\stackrel{\text{def}}{=} \mathbb{R} \end{aligned}$$

for $a \in \mathbb{R}$, here $-\infty$ and ∞ are symbols not representing any number.

1.4 Ordered Pairs and Tuples

A pair (x, y) coupling two objects x and y is called an *ordered pair*. An expression (x_1, x_2, \dots, x_n) with n items x_1, x_2, \dots, x_n aligned is called an *n -fold tuple*. An ordered pair is a two-fold tuple. Each of items x_1, x_2, \dots, x_n in an *n -fold tuple* (x_1, x_2, \dots, x_n) is called a *component*. The order and duplication of components are meaningful, unlike the extensional definition of a set. For example, all of $\{1, 2\}$, $\{2, 1\}$ and $\{1, 2, 2, 1\}$ represent the same set, but we consider $(1, 2)$, $(2, 1)$ and $(1, 2, 2, 1)$ are different objects as two ordered pairs and one four-fold tuple.

Now let X and Y be nonempty sets. The set of all ordered pairs (x, y) for $x \in X$ and $y \in Y$ is called the *direct product* or the *Cartesian product* of X and Y , and written as $X \times Y$. Similarly, for nonempty sets X_1, X_2, \dots, X_{n-1} and X_n we denote by $X_1 \times X_2 \times \dots \times X_n$ the set of all *n -fold tuples* (x_1, x_2, \dots, x_n) with $x_1 \in X_1, x_2 \in X_2, \dots, x_n \in X_n$. When $X_1 = X_2 = \dots = X_n = X$, X^n denotes the direct product $X_1 \times X_2 \times \dots \times X_n$. If X is a finite set and consisting of k elements, X^n has k^n elements.

The direct product $\mathbb{R} \times \mathbb{R} = \mathbb{R}^2$ is the xy -coordinate plane, which is the set of all points (x, y) with $x, y \in \mathbb{R}$, and $\mathbb{R} \times \mathbb{R} \times \mathbb{R} = \mathbb{R}^3$ is the xyz -coordinate space. Identifying (x) with x itself, \mathbb{R}^1 is the same as \mathbb{R} .

In Python, an *n -fold tuple* is a `tuple` class object which is expressed by components lined up in a row and enclosed in parentheses. In mathematics we distinguish between the terms “element” and “component”, but in Python, we are used to calling components of tuples their *elements* as in sets.

Let $x = (1, 2)$, $y = (2, 1)$, and $z = (1, 2, 1)$. We check whether $x = y$, $x = z$, and $y = z$ hold or not by Python.

```
In [1]: x = (1, 2); x
```

```
Out[1]: (1, 2)
```

```
In [2]: y = (2, 1); y
```

```
Out[2]: (2, 1)
```

```
In [3]: z = (1, 2, 1); z
```

```
Out[3]: (1, 2, 1)
```

```
In [4]: x == y, x == z, y == z
```

```
Out[4]: (False, False, False)
```

In [4] refers to the Boolean values of $x == y$, $x == z$, and $y == z$. In Out [4], a tuple of the Boolean values is returned.

In Python, we say that an object has some *data type*, which means the object is an instance of some *class*. A name of a data type plays the role of a function which converts a value of a given object equipped with some data type into a value of another specified data type. This mechanism is called the *type casting*. Let us cast tuples x , y , and z into `set` class objects.

```
In [5]: set(x) == set(y), set(x) == set(z), set(y) == set(z)
```

```
Out[5]: (True, True, True)
```

```
In [6]: set(x), set(y), set(z)
```

```
Out[6]: ({1, 2}, {1, 2}, {1, 2})
```

All of `set(x)`, `set(y)`, and `set(z)` express the same set $\{1, 2\}$. Sometimes, the casting function `set` is used to delete duplication of elements in a tuple t by writing `tuple(set(t))`.

We can access the elements of a tuple by indices (subscripts). Indices are integers, and the first (left-most) element has index 0, the second 1, the third 2, and so on.

```
In [1]: a = (2, 3, 5, 7); a
```

```
Out[1]: (2, 3, 5, 7)
```

```
In [2]: a[0], a[1], a[2], a[3]
```

```
Out[2]: (2, 3, 5, 7)
```

Negative integers $-1, -2, -3, \dots$ can be allowed as indices which start from the last (rightmost) element backward to the previous one.

```
In [3]: a[-1], a[-2], a[-3], a[-4]
```

```
Out[3]: (7, 5, 3, 2)
```

If a tuple has n elements, the maximum index is $n - 1$ and the minimum index is $-n$. Integers out of this range cause an indexing error.

We sometimes need a tuple with only one element. The expression `(1)` means just the same as integer 1 as stated above, and we must express such a tuple as `(1,)` in Python.

```
In [1]: x = (1,); x
```

```
Out[1]: (1,)
```

```
In [2]: x[0]
```

```
Out[2]: 1
```

1.5 Mappings and Functions

Let X and Y be nonempty sets. A *mapping* from X to Y assigns to each element of X an element of Y . A mapping is also an object in mathematics, and if necessary, we can name it as f, g, h , etc. We denote a mapping f from X to Y by $f : X \rightarrow Y$. The unique element $y \in Y$ that f assigns to $x \in X$ is denoted by $f(x)$.

The set of all mappings from X to Y is denoted by Y^X , that is,

$$Y^X \stackrel{\text{def}}{=} \{f \mid f : X \rightarrow Y\}.$$

This notation¹² may look awkward but it comes from the fact that the number of elements of Y^X just equals y^x for X with x elements and Y with y elements.

Exercise 1.9 Prove the above fact on the size of the power set Y^X .

In mathematics, the word *function* is a synonym for mapping, but it is usually used in cases where the target set Y is \mathbb{R} or \mathbb{C} . In Python, a mapping in mathematics is always called a function (an object of the function class). There are some definite functions in mathematics which cannot be defined by Python. One example is the function that assigns 1 to rational numbers and 0 to irrational numbers. On the other hand, Python's *function* contains a *procedure* (*subroutine*, or *subprogram*), which is a block of tasks and runs only when it is called. For example, the `print` function, if we call it in the format `print(x)`, prints out the value of argument `x` readable enough for us in the shell window. This function is a type of built-in function which can be used everywhere without importing its name. We do not need to know how it is implemented, and we only need to know the rule determining the relation between its argument (input) and its result (output). Such a rule is called the *specification* of the function. The casting functions `tuple` and `set` are also examples of built-in functions.

Let us compare the function definitions in mathematics and in Python, with some examples.

1. Definition by a simple expression: $f(x) \stackrel{\text{def}}{=} x^2 - 1$.

```
def f(x):
    return x**2 - 1
```

2. Definition by divided cases: $g(x) \stackrel{\text{def}}{=} \begin{cases} 0 & \text{if } x < 0, \\ 1 & \text{otherwise.} \end{cases}$

```
def g(x):
    if x < 0:
        return 0
```

¹² This is a comprehensive notation in the sense that the power set 2^A is considered to be the set of all mappings from A to $\{0, 1\}$, and the direct product X^n is considered to be the set of all mappings from $\{1, 2, \dots, n\}$ to X .

```
else:
    return 1
```

This is equivalent to the following description with the *ternary operator*.

```
def g(x):
    return 0 if x < 0 else 1
```

3. An anonymous function: $x \mapsto x^2 - 1$.

```
lambda x: x**2 - 1
```

Python's anonymous function is called the *lambda expression* whose idea and name come from λ -calculus in the theory of mathematical logic. Situations using a lambda expression will occur when the act of naming for reuse of the function is not necessary. In mathematics the function $f : \mathbb{R} \rightarrow \mathbb{R}$ defined by $f(x) \stackrel{\text{def}}{=} x^2 - 1$, for example, is sometimes written as $f : x \mapsto x^2 - 1$, which is more descriptive to express the relationship between x and $f(x)$. Python allows lambda expressions for such use. The function f defined by `def f(x): return x**2 - 1` can be written as `f = lambda x: x**2 - 1`.

The factorial function $f(n) = n!$ is defined using mathematical induction by

$$f(n) \stackrel{\text{def}}{=} \begin{cases} 1 & \text{if } n = 0, \\ n \cdot f(n-1) & \text{otherwise.} \end{cases}$$

In Python, such a function is called a *recursive* function. It calls itself during its execution and is defined as follows:

```
def f(x):
    if n == 0:
        return 1
    else:
        return n * f(n-1)
```

For a mapping (function) $f : X \rightarrow Y$, the subset

$$\{(x, y) \mid x \in X \text{ and } y = f(x)\}$$

of the direct product $X \times Y$ is called the *graph* of f . For two mappings f and g , if their graphs are equal as sets, they are the same mapping, and written $f = g$.

Consider a mapping $f : X \rightarrow Y$, then X is called the *domain* of f . For a subset A of X , the subset $f(A)$ of Y defined by

$$f(A) \stackrel{\text{def}}{=} \{f(x) \mid x \in A\} = \{y \mid \text{there exists } x \in A \text{ such that } y = f(x)\}$$

is called the *image* of A with respect to f . On the other hand, for a subset B of Y , the subset $f^{-1}(B)$ of X defined by

$$f^{-1}(B) \stackrel{\text{def}}{=} \{x \mid f(x) \in B\}$$

is called the *inverse image* of B with respect to f .

The image $f(X)$ of the domain X is called the *range* of f and is denoted by $\text{range}(f)$. If $\text{range}(f) = Y$, then f is said to be *surjective* or *onto*. If $f(x) \neq f(x')$ whenever $x \neq x'$ for $x, x' \in X$, then f is said to be *injective* or *one-to-one*. An injective and surjective (one-to-one onto) mapping is called a

bijective mapping or simply a *bijection*. If f is a bijection, for each $y \in Y$, there exists unique $x \in X$ such that $f(x) = y$, so we have the mapping $h : Y \rightarrow X$ which sends $y \in Y$ with $y = f(x)$ to $x \in X$. This h is called the *inverse mapping* of f and denoted by f^{-1} . We have $f^{-1}(f(x)) = x$.

Let $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ be mappings. The mapping $h : X \rightarrow Z$ defined by $h : x \mapsto g(f(x))$ is called the *composition* of f and g . This h is denoted by $g \circ f$. We have $(g \circ f)(x) = g(f(x))$.

Suppose both $f : X \rightarrow Y$ and $g : Y \rightarrow Z$ are bijections. Putting $z = (g \circ f)(x)$, we have

$$(f^{-1} \circ g^{-1})(z) = f^{-1}\left(g^{-1}\left(g(f(x))\right)\right) = f^{-1}(f(x)) = x,$$

Hence, $f^{-1} \circ g^{-1}$ is the inverse mapping of $g \circ f$, that is,

$$(g \circ f)^{-1} = f^{-1} \circ g^{-1}.$$

The mapping $\text{id}_X : X \rightarrow X$ that sends $x \in X$ to x is called the *identity mapping* on X . It is bijective and has the inverse, which is id_X itself. If $f : X \rightarrow Y$ is bijective, then $f^{-1} \circ f = \text{id}_X$, $f \circ f^{-1} = \text{id}_Y$ and $(f^{-1})^{-1} = f$ hold.

1.6 Classes and Objects in Python

Data that Python can handle are called *objects*. A *class* is an abstract concept that puts together all the objects with the same data structure. For example, the literal `3.14` is an object of `float` class representing a real number. The actual object is stored in computer memory as binary data in the form of floating-point data. An `int` class object has a different form in memory which represents an integer.

In Python, there are several types of objects which possess elements. Objects possessing elements are divided into two kinds, those in which elements are arranged in order such as tuples, and those in which the order of elements does not matter such as sets. *Strings* (`str` class objects) and *lists* (`list` class objects) are other examples of the former type. The elements of these objects are indexed by integers as so are the elements of a tuple.

A string and its use:

```
In [1]: A = 'Hello Python!'; A
```

```
Out[1]: 'Hello Python!'
```

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

```
Hello Python!
```

```
In [3]: print(A[0], A[1], A[2], A[3])
```

```
H e l l
```

```
In [4]: print(A[-1], A[-2], A[-3], A[-4])
```

```
! n o h
```

A literal that represents a string encloses the entire string with quotation marks `'` or double quotation marks `"`. The elements of a string can be referred to by subscripting them in the same way as a tuple.

A list and its use:

```
In [1]: B = ['Earth', 'Mars', 'Jupiter']; B
```

```
Out[1]: ['Earth', 'Mars', 'Jupiter']
```

```
In [2]: print(B[0], B[1], B[2])
```



```
Earth Mars Jupiter
```

```
In [3]: print(B[-1], B[-2], B[-3])
```



```
Jupiter Mars Earth
```

```
In [4]: B.append('Saturn'); B
```

```
Out[4]: ['Earth', 'Mars', 'Jupiter', 'Saturn']
```

A list encloses its elements in [and]. We can refer to an element of it by subscripts. Method `append` of `list` class adds a new element at the end of an object of `list` class, and the contents of the list are partially changed.

Dictionary (`dict` class object) consists of pairs of a key and a corresponding value. In the literal of a dictionary, each pair consists of a key and its value sandwiching a colon, and all pairs are separated by commas and enclosed in braces.

A dictionary and its use:

```
In [1]: C = {'Earth': '3rd', 'Mars': '4th', 'Jupiter': '5th'}; C
```

```
Out[1]: {'Earth': '3rd', 'Mars': '4th', 'Jupiter': '5th'}
```

```
In [2]: C['Earth']
```

```
Out[2]: '3rd'
```

```
In [3]: C['Saturn'] = '6th'; C
```

```
Out[3]: {'Earth': '3rd', 'Mars': '4th', 'Jupiter': '5th', 'Saturn': '6th'}
```

Lists and tuples use an integer as an index to refer to the value of their element, while a dictionary uses a key as an index to refer to the value paired with that key. By adding a new key and its value, the contents of a dictionary can be partially changed.

An object possessing elements can be checked by using `in` as to whether given data are contained in it. Also, using `in`, the elements of object can be values for a loop counter of a `for` statement.

Program: planet.py

```
In [1]: 1 C = {'Earth': '3rd', 'Mars': '4th', 'Jupiter': '5th'}
        2 for x in C:
        3     print(f'{x} is the {C[x]} planet in the solar system.')
        4 print()
        5 for x in sorted(C):
        6     print(f'{x} is the {C[x]} planet in the solar system.')
```

Line 2: Loop counter `x` walks around the keys of dictionary `C`.

Line 3: `f'{x}'` is the `{C[x]}` planet in the solar system. `'` is called an *f-string*, and it is the string obtained by substituting the literal of the value `x` and `C[x]`, respectively, into the position enclosed by `{ }`.

Line 4: The `print` function without no arguments sends only a newline code.

Line 5: The order of keys in a dictionary is decided by the convenience of Python (see footnote 10). If one wants to sort with a certain intention, use the `sorted` function. In this example, the lexicographical order is expected. We can choose another order by changing the argument of `sorted`.



```
Earth is the 3rd planet in the solar system.
Mars is the 4th planet in the solar system.
Jupiter is the 5th planet in the solar system.

Earth is the 3rd planet in the solar system.
Jupiter is the 5th planet in the solar system.
Mars is the 4th planet in the solar system.
```

Note that the f-string is a relatively recent notation added since Python 3.6. It is better also to know other equivalent notations used before.

```
In [1]: x, y, z = 1, 2, 3; x, y, z
```

```
Out[1]: (1, 2, 3)
```

```
In [2]: f'{x}+{y}+{z}'
```

```
Out[2]: '1+2+3'
```

```
In [3]: '{}+{}+{}'.format(x, y, z)
```

```
Out[3]: '1+2+3'
```

```
In [4]: '%s+%s+%s' % (x, y, z)
```

```
Out[4]: '1+2+3'
```

In[2] uses the f-string, In[3] uses the `format` method of `string` class, and In[4] uses the `format` operator `%`.

The `list` class has the `sort` method which is similar to but not the same as the `sorted` function.

```
In [5]: A = [z, y, x]; A
```

```
Out[5]: [3, 2, 1]
```

```
In [6]: sorted(A)
```

```
Out[6]: [1, 2, 3]
```

```
In [7]: A
```

```
Out[7]: [3, 2, 1]
```



```
In [8]: A.sort()
A
```

```
Out[8]: [1, 2, 3]
```

A function specialized for manipulating an object of a certain class is called a *method* of that class. The `sort` method is said to be *destructive*, for it rewrites the object to which the method subjects. The `append` method of `list` class is also destructive, because it modifies the contents of the list. Also, `C['Saturn'] = '6th'` rewrites the contents of dictionary `C`. Lists, sets, and dictionaries are called *mutable* objects in the sense that we can change some or all of them after they are created. Tuples and strings are *immutable* objects and their contents cannot be changed after the object is created. Objects of the classes `int`, `float`, and `complex` are also immutable. Immutable objects can be keys of a dictionary.

1.7 Lists, Arrays and Matrices

A sequence of several kinds of items is called a list in general. The number of the items is called the *length* or *size* of the list.

In mathematics, the use of the terms tuples and lists may be roughly described as follows. In the context where a tuple is used, its length is constant and each of their components is an element of a fixed particular set, that is, a tuple is an element of some Cartesian product set. A list, on the other hand, with variable length is used to simply enumerate objects and is not aware that it is an element of any Cartesian product set.

The difference between tuples and lists in Python is simple. Tuples are immutable and lists are mutable. As we have seen, tuples enclose their elements with (and), whereas lists enclose them with [and].

```
In [1]: A = (1, 2, 3); A
```

```
Out[1]: (1, 2, 3)
```

```
In [2]: B = [1, 2, 3]; B
```

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

```
In [3]: A[0], B[0]
```

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

```
In [4]: B[0] = 0; B
```

```
Out[4]: [0, 2, 3]
```

```
In [5]: A[0] = 0
```

```
Out[5]: Traceback (most recent call last):
  File "<pyshell#13>", line 1, in <module>
    A[0] = 0
TypeError: 'tuple' object does not support item assignment
```


a student, the second column is an integer that is the student ID number, and so on. In spite of this flexibility, using a list for a matrix, the calculation slows down as the size of the matrix increases, with the additional reason that Python's implementation is an interpreter.¹⁴

On the other hand, NumPy allows us to perform fast vector and matrix calculations by using the *ndarray* class (we will call it simply *array*¹⁵ in subsequent descriptions). This class provides convenient functions and methods for those calculations, and it makes our code easier to write and easier to read.

1.8 Preparation of Image Data

1.8.1 Binarization of Image Data with PIL and NumPy

Prepare a grayscale image file. If necessary, refer to Appendix A.6 and create a grayscale image file from your favorite photos or images. The image is expressed by a matrix whose element is the graylevel of each pixel. Here, it is assumed that the prepared grayscale image file is `mypict1.jpg` of Appendix A.4.

Program: `graph1.py`

In [1]:

```
1 import PIL.Image as Img
2 from numpy import array, zeros
3 import matplotlib.pyplot as plt
4
5 im = Img.open('mypict1.jpg')
6 A = array(im)
7 m, n = A.shape
8 avr = A.sum()/m/n
9 print((m, n), avr)
10
11 L = zeros(256)
12 for i in range(m):
13     for j in range(n):
14         L[A[i, j]] += 1
15
16 plt.plot(range(256), L)
17 plt.plot([avr, avr], [0, L.max()])
18 plt.show()
```

Line 1: Import module `PIL.Image` from `PIL` and name it `Img`.

Line 2: Import names `array` and `zeros` from `NumPy`.

Line 3: Import module `matplotlib.pyplot` and name it `plt`.

Line 5: Read the data from `mypict1.jpg`.

Line 6: Generate an array `A` from `im` which is an object of PIL's image data structure. The function `array` defined in NumPy is a data structure which is more convenient than a matrix expressed by a nested list.

¹⁴ A system that interprets code and executes it on a computer is called a language processing system. An *interpreter* is one in which a language processor is resident in memory and interprets and executes code line by line. On the other hand, a *compiler* converts the whole code into machine language that can be executed without the hand of the language processing system. By thinking of the former as an interpreter and the latter as a translator, we can understand the advantages and disadvantages of each.

¹⁵ Since the name `array` is already used in the built-in library, `nd` is added to distinguish it. The prefix `nd` means *n*-dimensional. We will not use built-in library `array` in this book.

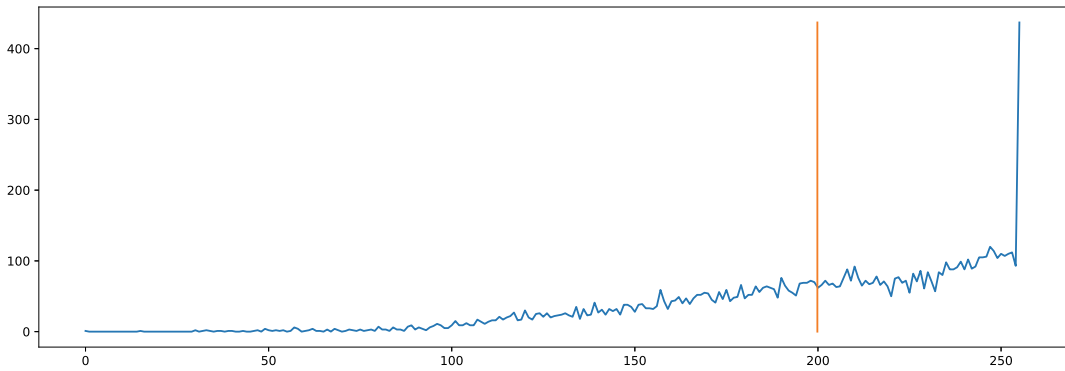


Fig. 1.3 Graph of graylevel distribution of pixels

Lines 5–9: A consists of $m \times n$ elements. The average `avr` of graylevels of the image is the sum `A.sum()` of all elements of A divided by the number of pixels $m \times n$.

Lines 11–14: Make up the array L whose k -th element is the number of all pixels with graylevel k for each integer k from 0 to 255.

Line 16: Draw the graph (Fig. 1.3) of L , which displays the distribution of graylevels of the image.

Line 17: Draw a vertical line at the value of `avr`.



```
(100, 88) 199.83977272727273
```

Let us look at the contents of A .

In [2]:

```
A
```

Out[2]:

```
array([[255, 253, 252, ..., 221, 197, 230],
       [245, 248, 250, ..., 149, 155, 212],
       [255, 255, 255, ..., 107, 168, 222],
       ...,
       [215, 222, 215, ..., 242, 255, 248],
       [243, 246, 243, ..., 244, 253, 252],
       [246, 235, 224, ..., 245, 249, 248]], dtype=uint8)
```

In [3]:

```
A[0, 0]
```

Out[3]:

```
255
```

Because it is too large to display all the contents of A in the shell window, they are truncated. The expression `dtype=uint8` means that the data type of each element is an 8-bit unsigned integer (an integer from 0 to 255) which represents the graylevel; 0 is pure black and 255 is pure white. In an array, its elements must be uniformly of the same type. The elements of array A are referenced with $A[i, j]$. Like lists and tuples, subscripts start with 0.

Program: mypict1.py

```
In [1]: 1 import PIL.Image as Img
2 from numpy import array
3
4 A = array(Img.open('mypict1.jpg'))
5 B = A < 200
6 m, n = B.shape
7 h = max(m, n)
8 x0, y0 = m/h, n/h
9
10 def f(i, j):
11     return (y0*(-1 + 2*j/(n - 1)), x0*(1 - 2*i/(m - 1)))
12
13 P = [f(i, j) for i in range(m) for j in range(n) if B[i, j]]
14 with open('mypict1.txt', 'w') as fd:
15     fd.write(repr(P))
```

Line 5: The expression $A < 200$ returns the array of Boolean class objects with the same shape as A , whose element is `True` if the corresponding element of A is less than 200, and `False` otherwise. Thus, 200 is the threshold of the graylevel. This array is assigned to B . You should choose another value for this threshold with reference to the average and distribution of the graylevels of your image. If this value is less (resp. greater) than the average, the number of pixels with the value `True` will be larger (resp. smaller) than the one with `False` in array B .

```
In [2]: B

Out[2]: array([[False, False, False, ..., False,  True, False],
 [False, False, False, ...,  True,  True, False],
 [False, False, False, ...,  True,  True, False],
 ...,
 [False, False, False, ..., False, False, False],
 [False, False, False, ..., False, False, False],
 [False, False, False, ..., False, False, False]])
```

Lines 7,8: Make x_0 or y_0 equal to 1 for the longer side of the image.

Lines 10,11: In order to make the pixel at the i -th row and the j -th column of the image correspond to the point (x, y) on the xy -coordinate plane, define the function f by

$$f : (i, j) \mapsto \left(x_0 \left(-1 + \frac{2j}{n-1} \right), y_0 \left(1 - \frac{2i}{m-1} \right) \right).$$

This f sends the four corners of the image to the corresponding corners of a rectangle in the xy -coordinate plane, precisely

$$\begin{aligned} (0, 0) &\mapsto (-x_0, y_0), \\ (0, n-1) &\mapsto (x_0, y_0), \\ (m-1, 0) &\mapsto (-x_0, -y_0), \\ (m-1, n-1) &\mapsto (x_0, -y_0). \end{aligned}$$

If the image is square, i.e. $m = n$, then $x_0 = y_0 = 1$, but otherwise the longer side just fits between -1 and 1 .

Line 13: Scan B to create the list P of the coordinates (x, y) such that $B[i, j]$ equals `True`.

Lines 14,15: Convert P to a string with the `repr` function and write it in the text file `mypict1.txt`. 'w' means to open the file for writing. There is also the `str` function to convert the object to a

string. It gives a human-readable string, but that string may not be evaluated as Python code. The string converted by the `repr` function can be restored by the `eval` function.

```
In [3]: A = array([1, 2, 3]); A
```

```
Out[3]: array([1, 2, 3])
```

```
In [4]: print(A)
```

```
Out[4]: [1 2 3]
```

```
In [5]: str(A)
```

```
Out[5]: '[1 2 3]'
```

```
In [6]: repr(A)
```

```
Out[6]: 'array([1, 2, 3])'
```

```
In [7]: eval('array([1, 2, 3])')
```

```
Out[7]: array([1, 2, 3])
```

The following program reads the text file `mypict1.txt` and displays the black-and-white image.

Program: `mypict2.py`

```
In [1]: 1 import matplotlib.pyplot as plt
        2
        3 with open('mypict1.txt', 'r') as fd:
        4     P = eval(fd.read())
        5     x, y = zip(*P)
        6     plt.scatter(x, y, s=3)
        7     plt.axis('scaled'), plt.xlim(-1, 1), plt.ylim(-1, 1), plt.show()
```

Lines 3,4: Open file `mypict1.txt` for reading, and `eval` converts the string data of the file created by `mypict1.py` into a list of data.

Line 5: `P` is the list $[(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)]$ of tuples (x_i, y_j) . Divide this into the list $[x_1, x_2, \dots, x_n]$ of the x -coordinates and the list $[y_1, y_2, \dots, y_n]$ of y -coordinates, and express them by `x` and `y` respectively.

Line 6: Create a scatter plot with Matplotlib. Keyword argument `s=3` of method `scatter` determines the size of the point plotted at the coordinates (x_i, y_i) .

Line 7: Argument `'scaled'` in `plt.axis` matches the aspect ratio of the graph to the ratio of the x -coordinate range to the y -coordinate range. In this way, the image is displayed on the left of Fig. 1.4. The quality of a black-and-white picture depends on the threshold you choose in `mypict1.py`. The right of Fig. 1.4 is a magnified view of the part specified by `plt.xlim(-0.25, 0.25)`, `plt.ylim(-0.25, 0.25)` in Line 7.

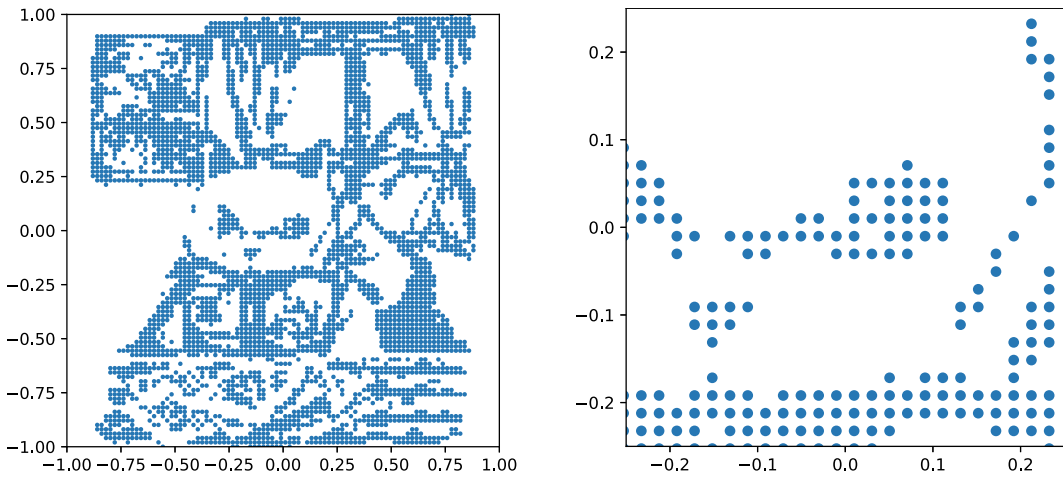


Fig. 1.4 Black-and-white image (left) and an enlarged view of part of it (right)

1.8.2 GUI for Creating Complex-Valued Data of Handwritten Characters

In Chaps. 6 and 10 we will deal with complex-valued data. In this chapter we make a GUI¹⁶ tool that creates line figures such as handwritten Chinese characters¹⁷ on a two-dimensional plane (Fig. 1.5), converts them into complex-valued data, and saves them into a file. Here, we use the built-in library *Tkinter*.

We think of a handwritten character as a sequence of points arranged in the order in which they are written on the square of $-1 \leq \operatorname{Re} z \leq 1$ and $-1 \leq \operatorname{Im} z \leq 1$ ($z \in \mathbb{C}$) in the complex plane. The sequence is expressed as a list of complex numbers and its representation by a string is stored in a text file.

Program: tablet.py

```
In [1]: 1 from tkinter import Tk, Button, Canvas
2
3 def point(x, y):
4     return C.create_oval(x - 1, y - 1, x + 1, y + 1, fill='black')
5
6 def PushButton(event):
7     x, y = event.x, event.y
8     Segs.append([(x, y), point(x, y)])
9
10 def DragMouse(event):
11     x, y = event.x, event.y
12     Segs[-1].append((x, y), point(x, y))
13
14 def Erase():
15     if Segs != []:
16         seg = Segs.pop()
17         for p in seg:
18             C.delete(p[1])
19
20 def Save():
21     if Segs != []:
22         L = []
```

¹⁶ An abbreviation for Graphical User Interface, a program that can interact with a computer by operating the screen with the mouse.

¹⁷ Kanji “ku-kan” meaning space.

In [1]:

```

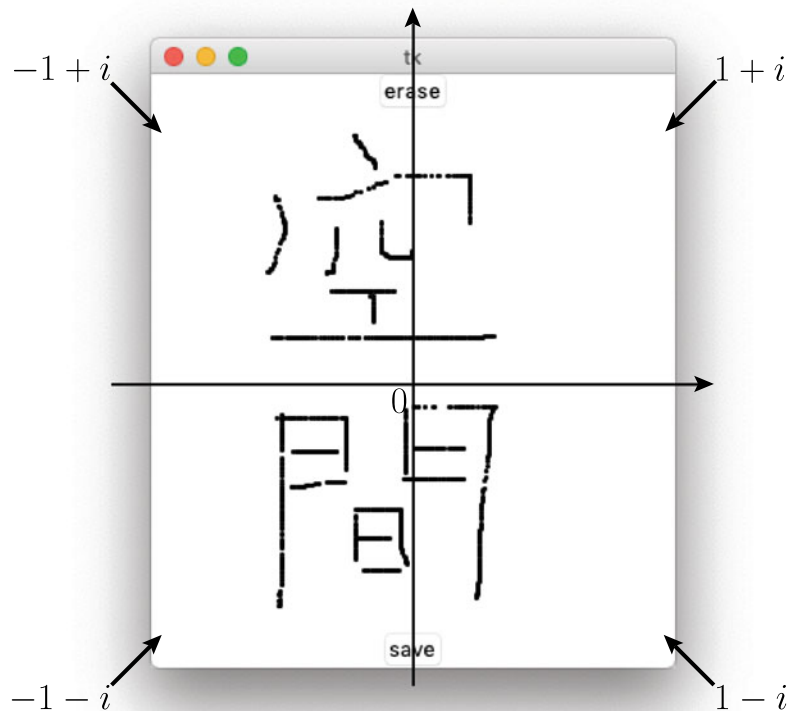
23     for seg in Segs:
24         for (x, y), _ in seg:
25             L.append((x - 160) / 160 + 1j * (160 - y) / 160)
26     with open(filename, 'w') as fd:
27         fd.write(repr(L))
28     print('saved!')
29
30 filename = 'tablet.txt'
31 Segs = []
32 tk = Tk()
33 Button(tk, text="erase", command=Erase).pack()
34 C = Canvas(tk, width=320, height=320)
35 C.pack()
36 C.bind("<Button-1>", PushButton)
37 C.bind("<B1-Motion>", DragMouse)
38 Button(tk, text="save", command=Save).pack()
39 tk.mainloop()

```

Line 1: Use the library tkinter for GUI tools. On a Tk class object of tkinter, place a Button class object that can be operated with mouse clicks and a Canvas class object on which we can draw line shapes with mouse drags.

Lines 3,4: Create an object that represents a point on the canvas. Since this function has only one line, it is possible to write the expression of the return value in Line 4 directly in the PushButton function in Line 6 and the DragMouse function in Line 10, but since one line will become long and hard to read, this is defined as a supplementary function. In programming, it is also important to write a code easier to read.

Fig. 1.5 GUI to create a line figure on the complex plane



Lines 6–28: Define functions that will be executed when mouse operations are detected. When the mouse button is clicked, a new line segment is drawn, and when the mouse is dragged, its trajectory becomes a line segment. Line segments are stored as two lists. One is a list of tuples that represent the position coordinates of points. The other is a list of point objects (small black circles) on the canvas. We use the former when saving the figure and the latter when erasing a line segment. We consider a list, which records changes in the pair of coordinates of the mouse pointer position and the point object generated on the canvas while dragging the mouse, as a line segment.

Line 30: `filename` is the name of file to save data in when the save button is clicked.

Line 31: `Segs` is a list of list objects expressing line segments.

Lines 32–38: Create a window on the screen, embed a canvas and buttons on it, and decide which function to call by operating the mouse. When the erase button is pressed, the line segment drawn last is removed. When the save button is clicked, the position coordinate data of `Segs` are converted into complex numbers and saved as a list in the file.

Line 39: Enter an infinite loop waiting for mouse activity. The program will not exit unless the created window is closed. It is called *event-driven programming* in the sense that we describe in a program what action to take when a certain event occurs and wait till an event occurs.

1.8.3 Data of Handwritten Letters with Grayscale

Character data may be treated as a grayscale plane figure instead of a line figure. Figure 1.6 shows some MNIST data of handwritten digits which are often used as learning and test data of machine learning systems.

Visit the home page <http://yann.lecun.com/exdb/mnist/> in which we will find the following tags:

```
train-images-idx3-ubyte.gz: training set images (9912422 bytes)
train-labels-idx1-ubyte.gz: training set labels (28881 bytes)
t10k-images-idx3-ubyte.gz: test set images (1648877 bytes)
t10k-labels-idx1-ubyte.gz: test set labels (4542 bytes).
```

Download these files, unzip¹⁸ and rename these files as follows respectively:

```
train-images.bin
train-labels.bin
test-images.bin
test-labels.bin.
```

Put them in the same folder where the program `mnist.py` below is located. The file `train-images.bin` (resp. `test-images.bin`) has a total of 60,000 (resp. 10,000) images. The label (the digit expressing the image) for each image can be found in `train-labels.bin` and `test-labels.bin`. Usually, `train-*` files are used as learning data for machine learning and `test-*` files are used for test data. All of these are binary files and their data formats are written at the end of the above home page.¹⁹

Let us look inside the file. We check the number of image data for each character and calculate the average of them.

¹⁸ Compressed files with `.gz` extension can be decompressed by double-clicking on macOS and Raspberry Pi. For Windows, we need to install free software such as 7-Zip or Lhaplus.

¹⁹ MNIST handwriting data is also available through scikit-learn and TensorFlow, Python libraries related to machine learning.



Fig. 1.6 Selected MNIST data

Program: mnist.py

```
In [1]: 1 import numpy as np
        2 import matplotlib.pyplot as plt
        3
        4 N = 10000
        5 with open('test-images.bin', 'rb') as f1:
        6     X = np.fromfile(f1, 'uint8', -1)[16:]
        7 X = X.reshape((N, 28, 28))
        8 with open('test-labels.bin', 'rb') as f2:
        9     Y = np.fromfile(f2, 'uint8', -1)[8:]
       10 D = {y: [] for y in set(Y)}
       11 for x, y in zip(X, Y):
       12     D[y].append(x)
       13 print([len(D[y]) for y in sorted(D)])
       14
```

```
In [1]: 15 fig, ax = plt.subplots(11, 10, figsize=(10, 10))
16 plt.subplots_adjust(wspace=0.1, hspace=0.1,
17                     left=0.01, right=0.99, bottom=0.01, top=0.99)
18 for y in D:
19     for k in range(10):
20         A = 255 - D[y][k]
21         ax[k][y].imshow(A, 'gray')
22         ax[k][y].tick_params(labelbottom=False, labelleft=False,
23                             color='white')
24     A = 255 - sum([x.astype('float') for x in D[y]]) / len(D[y])
25     ax[10][y].imshow(A, 'gray')
26     ax[10][y].tick_params(labelbottom=False, labelleft=False,
27                             color='white')
28 plt.show()
```

Line 4: N is the number of data.

Lines 5,6: Open the binary file of the character images and read it until the last as unsigned 8-bit integers. We can change the last argument `-1` of the `np.fromfile` function to a positive integer, which means the number of the character images to read. The first 15 bytes consist of the header, so read only the data from the 16th byte and set it to array X .

Line 7: X is a one-dimensional array with $N \times 28 \times 28$ elements, so it is converted into a three-dimensional array. We consider it as N matrices of shape $(28, 28)$ whose components are unsigned 8-bit integers revealing the values of the grayscale.

Lines 8,9: Read the binary file of labels. The first n -th byte counted from the 8th byte is an integer value from 0 to 9 which is the digit (label) for the n -th matrix in X . Put this list of labels in array Y .

Line 10: Create a dictionary whose key is the label k and value is a list of matrices each of which is a pattern handwritten as the digit k . The list is initially empty. `set(Y)` is the set of all elements in array Y and it should actually be $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.

Lines 11–13: `zip(X, Y)` is the list of tuples (x, y) that are created by sequentially extracting x from X and y from Y . Matrix x has label y , so add x to the list of items with label y in the dictionary. In this way, the dictionary D in which images are classified by labels is completed. Line 13 checks how many images there are in each label. The keys in the dictionary are sorted by the `sorted` function, and the number of the images is displayed in the form of a list in the order 0, 1, 2, ..., 9.

Lines 15–28: Display the first 10 patterns and the total average of images for each label. Line 15 prepares to embed multiple graphs in one diagram. Arrange the charts in 11 rows by 10 columns. The bottom row is for the total average of the images. Lines 16 and 17 adjust the margins of the graph. On line 20, the black-and-white inverted image is set as A . Line 21 draws each A in grayscale. Lines 22 and 23 set not to draw the scale of each axis of the graph. Lines 24–27 calculate the average by taking a simple average of the grayscales in pixels, and set it to A . Line 25 draws the average A . Line 28 actually displays Fig. 1.6.

```
[980, 1135, 1032, 1010, 982, 892, 958, 1028, 974, 1009]
```

The test data have 980, 1135, 1032, 1010, 982, 892, 958, 1028, 974, and 1009 images for each label.