§2 Python programming for Physicists

Our first target is to learn how to write computer programs in the Python programming language. Python has a lot of facilities and features for performing diverse types of tasks. Here we are not going to learn all the features of Python. Instead, we will focus our attention to those that are useful for doing calculations in physics. For the information about other Python's features, read the online documentation on the official Python website at **https://www.python.org/**.

2.1 Getting Started with Python

To run Python programs on your computer, you first need to know how and where to enter them into the computer. By default, Python would be installed together with an **integrated development environment (IDE)** called IDLE which is a **graphical user interface (GUI)** for editing and running Python code. To help programmers writing programs, IDLE also includes features such as automatic indenting, color highlighting, and interactive development. You can also find a wide range of other IDEs for Python such as **Pycharm**, **Spyder**, and **Wing**, mostly for free, on the Internet. Here we consider how to perform Python programming with IDLE on Windows.

To launch IDLE, click the icon **IDLE (Python 3.7 64-bit)** under the Start menu on Windows. After starting IDLE, a window called **Python shell window** appears on the screen in which Python interpreter is running in an interactive mode called a **shell**. This shell allows user to enter python commands and then displays the result of execution. We can also access the Python shell from the command line by clicking on the icon **Python 3.7 (64-bit)** inside the **Python 3.7** folder in Start menu or typing **python** in the command prompt on Windows. When the Python shell is started in the command line, a greeting message like this would be shown on the screen:

```
Python 3.7.2 (tags/v3.7.2:9a3ffc0492, Dec 23 2018, 23:09:28) [MSC v.
1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This message displays the version of Python running on the computer and the system that the computer is working on followed by the symbol >>>. The >>> is a **Python prompt** which tells us that the computer is ready for us to type in commands. We can enter any Python command at this prompt through the keyboard and the computer will execute that command immediately. Below are the sample input and output for the Python shell:

```
>>> print("Hello, World!")
Hello, World!
>>> print(5 - 2)
3
>>> print("5 - 2 =", 5 - 2)
5 - 2 = 3
```

To exit the Python shell, type `exit()` or `quit()` at the Python prompt.

Running the Python interpreter in the shell mode is a useful way for trying individual Python commands. However, sometimes we want to type in an entire Python program at once and then run the whole program together. To do this, click on **File** and select **New File** from the menu bar of the Python shell window. An empty window called an **editor window** would pop up on the screen. In this window, you can type in a complete program and edit it just like in any text processor. The menu bar on the top of the editor window provides a number of word-processor features such as cut and paste. After finished writing the program, click **File** and select **Save** from the menu bar of the editor window to save it into a file with the **.py** extension. Such file is called a **module** or **script** which is simply a text file containing Python code. Python modules should have short names using only lowercase letters. You can also use underscores in module name to improve readability. As a trial, create the following module using the IDLE:

```
# firstprogram.py
# This is my first program which prints a welcome message.
# Last Update on 19 Nov 2020 by F K Chow

print("Welcome to Python!")
print("This is my first program!")
```

To run a Python program in the editor window, click **Run** and select **Run Module** from the menu bar of this window. You can also run it by pressing the **F5** function key which is a shortcut for this operation. When IDLE runs the program, control will shift over to the Python shell window. Here is the output of the module `firstprogram.py` in the shell window:

```
============ RESTART: D:/PythonCodes/firstprogram.py =============
Hello!
Welcome to Python!
```

The first line of the output indicates that the shell has restarted. IDLE does this each time we run a program so that the program runs in a new environment. The Python interpreter then executes the program line by line from top to bottom. The result would be just like that obtained by typing them one-by-one at the interactive Python prompt. Alternatively, we can also execute the program by typing **python firstprogram.py** at the command line in the directory where the program is located.

2.2 Basics Programming

This section introduces the basics concepts of the Python programming language including its syntax, structure, and data types.

2.2.1 Basic Elements of a Program

Let's begin by studying the program `tempconvert.py` with its sample input and output as shown below.

```
# tempconvert.py
# A program converts Fahrenheit temperature to Celsius.
# Last Update on 19 Nov 2020 by F K Chow

stopCon = "N"
print("This program converts Fahrenheit temperature to Celsius.")
while (stopCon != "Y"):
    fahrenheit = float(input("Enter the Fahrenheit temperature: "))
    celsius = (fahrenheit - 32)*5/9
    print("It is equal to {0:.2f} degrees Celsius.".format(celsius))
    stopCon = input("Enter Y to stop conversion: ")
print("Temperature conversion is stopped.")
```

```
=============== RESTART: D:/PythonCodes/tempconvert.py ===============
This program converts Fahrenheit temperature to Celsius.
```

```
Enter the Fahrenheit temperature: 100
It is equal to 37.78 degrees Celsius.
Enter Y to stop conversion: y
Enter the Fahrenheit temperature: 250
It is equal to 121.11 degrees Celsius.
Enter Y to stop conversion: Y
Temperature conversion is stopped.
```

From this program, we can observe the followings about the basic elements of a Python program.

1. A Python program contains a sequence of **statements** that specifies the commands to be executed by the Python interpreter. Individual statements do things such as performing arithmetic, getting input from the user of the program, and printing output to the screen. For example, the statement `stopCon = "N"` in the program `tempconvert.py` assigns the variable `stopCon` to be the letter N.

2. Comments are the notes in a program intended for human readers to understand the code. They are ignored by compilers and interpreters. *A Python program should begin with comments describing its purpose and include any further information explaining how its individual parts works.* It's also useful to include the filename, the date that the program was first written, and the note about any subsequent changes that have been made in comments. In Python, comments begins with a hash (`#`) symbol. The Python interpreter will ignore everything from the hash symbol to the end of that line. A comment may appear at the start of a line or following whitespace or code, but not inside quotes. A hash symbol inside quotes is just a hash symbol. Below are the examples illustrating the use of comments in Python:

   ```
   >>> print("Hello, World!") # This is a comment!
   Hello, World!
   >>> print("# This is not a comment!")
   # This is not a comment!
   ```

3. Just like in other computer languages, **variables** are used in Python to give names to **values** (e. g. `fahrenheit` and `celsius`) that can be referred again later in the program. In Python, variables don't require declaration to reserve memory space. However, we must initialize them before use. Besides, the value of a variable can be changed and a variable can even hold values of different types.

4. In Python, a **function** is a block of organized and reusable code used to perform a single task. (It is called a procedure or a subroutine in other languages.) Python has many **built-in functions** — functions that are provided by default. To call a built-in function, we type the name of the function followed by its **arguments** listed in parentheses. For example, the built-in functions **`input`** and **`print`** are used to input and output data in the program `tempconvert.py`. We will discuss the details about functions in subsection 2.2.6.

5. **Python identifiers** are user-defined names to identify variables, functions, and modules etc. Below are the rules for naming identifiers in Python.

   (a) *Identifier names are case-sensitive*, e. g. **`x`** and **`X`** are two different variables.

   (b) Identifier names can contain letters, numbers, and the underscore character (_). However, they must not start with a number.

   (c) There are some **reserved words** or **keywords** in Python that cannot be assigned as identifier names. The complete list of Python keywords is shown in Table 2.1.

   Table 2.1: Python keywords

`and`	`as`	`assert`	`break`	`class`	`continue`	`def`
`del`	`elif`	`else`	`except`	`finally`	`for`	`from`
`global`	`if`	`import`	`in`	`is`	`lambda`	`nonlocal`
`not`	`or`	`pass`	`raise`	`return`	`try`	`while`
`with`	`yield`	`False`	`None`	`True`		

   In general, programmers are free to choose any identifier name that conforms to the above rules. Good programmers always try to choose names that describe the thing being named.

6. A Python program is constructed from **code blocks** or **blocks**. A block is a group of Python statements that is executed as a single unit. Three common types of blocks are modules, definitions of **classes**, and bodies of functions. *Python uses indentation to indicate code blocks* which makes the code look neat and clean. In Python, all the statements belonging to the same block must be indented by the same distance to the right (i. e. inserting the same amount of leading whitespaces) so that the statements within a block line up vertically. A code block ends at a line less indented or the end of the file. We can place a block inside another block (i. e. create a **nested block**) simply by indenting the inner block further to the right. By convention, four whitespaces are used for each level of indentation. It is preferred to use whitespaces instead of tabs

for indentation although both can be used. For example, the four statements right after the statement `while (stopCon != 'Y'):` in the program `tempconvert.py` belong to the same block. These statements will be executed repeatedly if the condition in the `while` statement is true. Note that we must enter an empty physical line to terminate a compound statement when the Python interpreter is running in the shell mode. It's also good to separate groups of relevant statements in a Python program by blank lines to enhance readability of the program.

2.2.2 Expressions and Assignments

Computer programs operate on data. According to the Python data model in the official Python documentation, "**Objects** are Python's abstraction for data. All data in a Python program is represented by objects or by relations between objects. Every object has an identity, a type and a value.". It implies that *everything in Python is an object*. **Strings** are objects. **Lists** are objects. Functions are objects. Even modules are objects. It's because they have a unique identity, a type, and some content. An object is created from a "blueprint" called a class which dictates its behavior. After an object is created, we cannot change its identity and type. However, whether its value can be changed depends on its type. Objects like lists and classes are **mutable** which means that their content can be updated. Other objects like numbers and strings are **immutable** which means that their content cannot be updated but return new objects.

An **expression** in Python is a fragment of code that produces some values. The simplest kind of expression is a **literal** that generates a constant value of some built-in types. For instance, the `tempconvert.py` program contains the numeric literals 5, 9, and 32. This program also manipulates textual data which are referred to as **strings** by computer scientists. In Python, a string literal is a sequence of characters enclosed by single quotes (`''`) or double quotes (`""`). And a character literal is just a single character enclosed by single or double quotes. We can also use either triple-single quotes (`'''`) or triple-double quotes (`"""`) to enclose a string with more than one line in Python. In the `tempconvert.py` program, we can find the character literal `"N"` and several string literals such as `"Enter the Fahrenheit temperature: "`. Note that the quotes themselves are not part of the string. They are just used to tell Python to create a character or string literal. We will discuss the details about strings in section 2.4.

The process of turning an expression into the result is called **evaluation**. When we enter an expression into the Python shell, the shell evaluates the expression and prints out a textual representation of the result. Note that the shell displays the value of a string by enclosing the sequence of characters in single quotes. It is a way of indicating that the value is text but not other data type. For example, the expression `"18"` in the interactive Python session below produces a string instead of a number. In such case, Python is actually storing the characters 1 and 8 instead of a representation of the number 18.

```
>>> "Welcome"
'Welcome'
>>> 18
18
>>> "18"
'18'
```

An expression can be also a simple identifier. When an identifier appears as an expression, its value is retrieved to provide the result for the expression. To understand how to use variables as expressions, consider the following interaction with the Python interpreter:

```
>>> a = 4
>>> a
4
>>> b = a
>>> print(b)
4
>>> print(day)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'day' is not defined
```

In the second line of the interaction, we are asking Python to evaluate the expression `a`. Thus the Python shell prints out 4 which is the value that was just assigned to `a`. Of course, the same output is obtained if we use a print statement to explicitly ask Python to print `a`. The last line of the interaction shows that Python reports a `NameError` if we try to use a variable that has not been assigned a value. It is because *a variable must always be assigned a value before it can be used in an expression*.

Complex and interesting expressions can be constructed by combining simple expressions with **operators**. For instance, a **mathematical expression** can be formed by combining numbers with Python arithmetic operators such as `+`, `-`, `*`, and `/` for performing addition, subtraction, multiplication, and division. In the `tempconvert.py` program, we can find the mathematical expression `(fahrenheit - 32)*5/9`. We will discuss the details about Python operators for numbers in subsection 2.2.4.

After a data value has been produced or calculated in a Python program, we probably need to store the value in computer memory and label it by a variable for later retrieval. In many computer languages, a variable is a named storage location in computer memory, i. e. a box for holding values. If we change the value of a variable, the old value will be erased and a new value will be put into it. However, this "variable as a box" model is not valid in Python.

In Python, a variable is a symbolic name that is a reference or pointer to an object. Such "name-to-object" mapping allows us to access an object by a name that have been assigned to it. But the data itself is still contained within the object. To assign a value to a Python variable, we can use a simple **assignment statement** of the form

```
<var> = <expr>
```

where `<var>` is a variable and `<expr>` is an expression. It tells the computer to evaluate the expression on the right to produce a value and then point the variable on the left to a newly created object that holds the value. For example, the `tempconvert.py` program has the assignment statements `stopCon = "N"` and `celsius = (fahrenheit - 32)*5/9`.

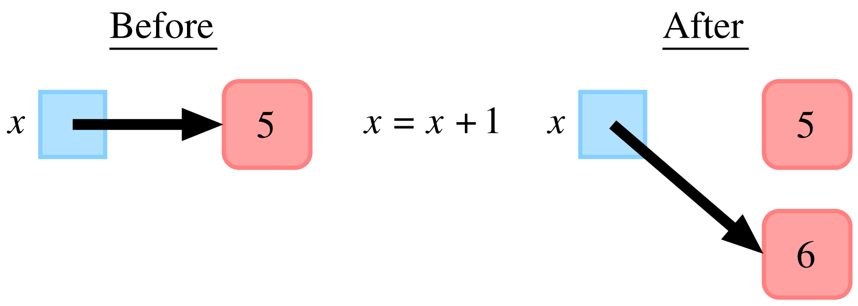

Figure 2.1: "Variable as a sticky note" model for assignment in Python.

Figure 2.1 is a picture showing how assignment works in Python in which an arrow is used to indicate the value referred by a variable. Suppose a variable is assigned a value so that it refers to the object holding the value. If we change its value, the old value doesn't get erased by the new one. The variable is simply "redirected" to refer to a newly created object holding the new value. It is just like moving a sticky note from one object to another. Notice that the original object would remain the same after the variable is changed. Below is an interactive Python session demonstrating this behavior:

```
>>> a = 5
>>> b = a
>>> id(a)
140713515082704
>>> id(b)
140713515082704
>>> a = 10
>>> id(a)
140713515082864
```

Here we use the identity function **id** to check the identity of an object.

In Python, an assignment statement doesn't directly cause the old value of a variable to be erased and overwritten. But we don't need to worry about computer memory getting filled up with the objects holding these "discarded" values. Obviously, an object is no longer useful when it is no longer referred to by any variables. Python will automatically clear these objects out of computer memory to free up the space for something else. This process of automatic memory management is called **garbage collection**.

We can change the value as well as the type of a variable as a Python program runs. For instance, we can assign an integer value to a variable and then assign a string to the same variable later. But we should not use the same variable as different types in a single program. It's because this makes the program much more difficult to follow and increases the chance for making a mistake in the program. Beware that it's always the value of the most recent assignment that would be retained if a variable is assigned multiple times. Below is an interaction with the Python interpreter illustrating this property:

```
>>> x = 1
>>> x
1
>>> x = 3
>>> x
3
>>> x = x + 2
>>> x
5
```

In the last assignment statement, the expression on the right computes the new value using the current value of the variable and then the variable points to a newly created object holding the new value.

Python also supports **simultaneous assignment**, i. e. assigning several variables at the same time. To assign a single value to multiple variables simultaneously, we use an assignment statement of the form

```
<var1> = <var2> = ... = <varN> = <expr>
```

This statement tells Python to evaluate the expression on the right and then assign it to all the variables on the left at the same time. In addition, to assign multiple values to multiple variables simultaneously, we use an assignment statement of the form

```
<var1>, <var2>, ..., <varN> = <expr1>, <expr2>, ..., <exprN>
```

This statement tells Python to evaluate all the expression on the right and then assign them to the corresponding variables on the left at the same time. Here is an interactive Python session showing how to use these assignment statements:

```
>>> a = b = c = 1
>>> print(a, b, c)
1 1 1
>>> a, b, c = 1, 2, 3
>>> print(a, b, c)
1 2 3
>>> a, b = b, a
>>> print(a, b, c)
2 1 3
```

Note that the values for the variables `a` and `b` are swapped by the last assignment statement. It's because the assignment is simultaneous so that none of the original values would be wiped out.

2.2.3 Numeric and Boolean Data Types

In Python, we can classify the data stored in a variable by its **data type**. The data type determines what values it can have and what operations can be performed on it. There are various data types in Python. Here we discuss **numeric data types** and **boolean data type** which are used frequently in physics programming. We delay the discussion of other data types such as strings and lists to later sections.

Numeric data types store numeric values including integers, floating point numbers, and complex numbers. Indeed, any number entered in Python will be regarded as a numeric value. Python 3 supports three different numeric data types:

1. `int`: They are integers which can be positive or negative whole numbers such as `3`, `−96`, `1234567812345678`. Of course, they cannot take non-integer values like 6.3. Without any prefix, they represent decimal numbers such as 123. They can also represent binary, octal, and hexadecimal numbers by placing an appropriate prefix before that number (see Table 2.2). `0B101001011`, `−0o62755`, and `0XAC1E2B0F54E` are examples of these numbers. Integer arithmetic is exact in Python. In Python 3, there is no limit to the length of an integer (but it's limited by the amount of computer memory available). Python considers any single number without a decimal (`.`) to be an integer.

Table 2.2: Prefixes for integers of different number systems in Python

Number System	Prefix
Binary	`0b` or `0B`
Octal	`0o` or `0O`
Hexadecimal	`0x` or `0X`

2. `float`: They are floating point numbers which represent real numbers such as `17.3`, `−10.8`, and `365.24`. Note that they can take integer values like 3.0. In general, they do not have the exact value of the real number that they represent, but are stored in memory to a certain precision. For example, the number 7/6 will be stored as 1.16666666666666740681... which is close to its exact value. Due to the limited precision, floating point arithmetic is not exact in Python but with care it's good enough for most scientific applications. Python also supports scientific notation using `e` or `E` to separate the mantissa from the exponent. For example, `1.67263e−7` represents the number 1.67263×10^{-7}. Python considers any single number with a decimal (`.`) or in scientific notation to be a floating point number.

3. `complex`: They are complex numbers expressed in the form $a+bi$ where $i=\sqrt{-1}$ is the pure imaginary number, and a and b are two real numbers known as the real and imaginary parts of that number. Thus complex number arithmetic is not exact but subject to the same finite precision as floating point numbers. In Python, the pure imaginary number i is denoted as `j` or `J` and both the real and imaginary parts of any complex numbers are always represented as floating point numbers. A complex number may be specified either by "adding" a real number to an imaginary one such as `2.8 + 5.3J` and `9.1 − 7.6j`. It can be also specified by separating the real and imaginary parts in a call to complex such as `complex(2.8, 5.3)` and `complex(9.1, −7.6)` which are the same as the previous two complex numbers.

Actually, we can represent both integers and floating point numbers as complex numbers. Why do we need different data types for numbers? The reason is having these different data types gives us great advantages. For instance, the memory required to store a complex number is double of that for a floating point number since the computer has to store both the real and imaginary parts. As a result, working on floating point numbers instead of complex numbers leads to significant reduction in the required memory space if our program has to deal with an enormous amount of numerical data which is a common case for physics calculation. Moreover, it takes much longer time to perform calculations with complex numbers than floating point numbers because the computer has to deal with both the real and imaginary parts. The computer still has to do the calculation for both parts even if the imaginary part is zero. It's very often that our physics program involve an enormous amount of numerical computation. The speed of individual computations has a significant effect in such case. So it's better for us to work on floating point numbers if we don't need to work with complex numbers.

Similar arguments also apply to floating point numbers and integers. If we know that the numbers that we are working with are integers, then using integers is usually faster and takes up less memory space. Moreover, sometimes it's actually more accurate to perform calculation with integers than floating point numbers. It's because floating point numbers are approximate representations of real numbers. The difference between the actual value and the approximate value is negligible in most cases. However, such difference could be crucially important in some special cases. Many bugs and problems in computer programs have arisen due to this issue. To avoid such trouble, we should store the number that we are working on as an integer if it is really an integer.

We can convert a value to different numeric data types by using the type conversion functions `int`, `float`, and `complex`. Here is an interaction with the Python interpreter showing how to use these functions:

```
>>> a = int(5.5)
>>> a
5
>>> b = float(a)
>>> b
5.0
>>> c = complex(a)
>>> c
```

```
(5+0j)
>>> type(c)
<class 'complex'>
```

From the above interaction, we can observe that the fractional part of a floating point number is simply discarded if it is converted to an integer. In other words, the value is truncated but not rounded. Besides, we can see that one can use the `type` function to check the type of a value.

Boolean data type store boolean values which are used to represent the truth values in logical operations. In Python, it is called `bool` which can have a value of either `True` or `False`. Indeed, `bool` is a subtype of integers. Boolean values `False` and `True` behave like the integers 0 and 1, respectively, in numeric contexts (such as used as the argument to an arithmetic operator). Many mathematical operations give us answers that evaluate to boolean values. Moreover, we can store a boolean value in a variable just like a number. Below is an interactive Python session illustrating these properties:

```
>>> 5 > 6
False
>>> 2 < 7
True
>>> x = (10 > 1)
>>> print(x)
True
```

Python numbers are objects and they have certain **attributes** including **data attributes** and **methods**. Data attributes tell us the properties of an object and they are accessed by the dot notation using the syntax <*object*>.<*attribute*>. For instance, complex numbers have the data attributes `real` and `imag` which yield the real and imaginary parts of the number:

```
>>> (5+6j).real
5.0
>>> (5+6j).imag
6.0
```

Methods are callable functions acting on objects in someway. They are **called** (i. e. executed) using the syntax <*object*>.<*methodname*>(<*arg1*>, <*arg2*>, ... <*argN*>) where <*arg1*>, <*arg2*>, ... <*argN*> are the "input" of the method. For example, complex numbers have the method `conjugate` which returns their complex conjugates:

```
>>> (5+6j).conjugate()
(5−6j)
```

Note that the parentheses after the method name should never be omitted. If we omit them like in `(5+6j).conjugate`, we are referring to the method itself without calling it. Another example is the method `bit_length` for integers which gives the number of bits in memory taken up by an integer:

```
>>> (51334848315466).bit_length()
46
```

2.2.4 Operators for Numeric Data Types

Operators are special symbols in Python that carry out operations on variables and values. For example, `3` and `4` are the operands and `+` is the operator in the expression `3 + 4`. The data type determines what operations can be performed on it. Here we introduce the operators for numeric data types in Python.

Table 2.3: Arithmetic operators in Python

Operator	**Meaning**	**Syntax**
+	Add the operands on both sides	`x + y`
−	Subtract right operand by left operand	`x − y`
∗	Multiply the operands on both sides	`x ∗ y`
/	Divide left operand by right operand	`x / y`
//	Take the floored quotient of right operand by left operand	`x // y`
%	Take the remainder of left operand divided by right operand	`x % y`
−	Negate the operand	`−x`
+	Indicate the operand is positive	`+x`
∗∗	Raise left operand to the power of right operand	`x ∗∗ y`

Python provides **arithmetic operators** to perform mathematical operations on numbers. Table 2.3 shows the list of these operators in increasing order of precedence. We should note the followings about these operators:

1. Python provides two different operators for division. The division operator `/` is used for ordinary division which always gives a result of type **`float`**. The **floor division** operator `//` gives the quotient of the division rounded down to the nearest integer. Although the result is a whole number, it is not necessary of type **`int`**.

2. The modulus operator `%` and the floor division operator `//` are not defined for complex numbers.

3. In Python, the power of a number can be also computed by the function **`pow`** where `pow(a, b)` is equivalent to `a ** b`. Just like in other computer languages, Python defines `pow(0, 0)` and `0 ** 0` to be `1`.

4. *Python chooses the appropriate underlying operation (integers, floating point numbers, or complex numbers) based on the operands.* For instance, if a binary arithmetic operator except the division operator `/` has operands of the same numeric type, then the result of the operation will also have the same numeric type. However, if a binary arithmetic operator has operands of different numeric types, then the result of the operation will be the more general of the two types of the operands. That is to say, if the two operands are **`int`** and **`float`**, then the result will be a **`float`**. If the two operands are **`float`** and **`complex`**, then the result will be a **`complex`**. Below is an interaction with the Python interpreter demonstrating these features:

```
>>> 3 + 7
10
>>> 3.0 + 7.0
10.0
>>> 7/3
2.3333333333333335
>>> 7//3
2
>>> 7.0//3.0
2.0
>>> 3**7
2187
>>> 3**7.0
2187.0
>>> 7.0 - (3 + 0J)
(4+0j)
```

5. We can combine several arithmetic operators together to make a complicated expression such as `2*a - b**3 + c/5`. In such case, we need to consider the **precedence** of the operators — their order of evaluation in an expression with multiple operators. The precedence order of the arithmetic operators in Python is similar to that in normal

algebra. Note that the precedence rules can be overridden by the use of parentheses. Moreover, operators of equal precedence are evaluated left to right with the exception of the exponent operator ∗∗, which is evaluated right to left. For example,

```
>>> 9/3/6
0.5
>>> 9/(3/6)
18.0
>>> 3**2**3
6561
>>> (3**2)**3
729
```

Notice that if the result of an arithmetic operation is equal to positive or negative infinity, then it will be displayed as `inf` or `−inf`. Moreover, if an arithmetic operation gives an undefined result, then the result will be displayed as `nan` (not a number).

Table 2.4: Comparison operators in Python

Operator	**Meaning**	**Syntax**
==	Equal	`x == y`
!=	Not equal	`x != y`
<	Less than	`x < y`
>	Greater than	`x > y`
<=	Less than or equal to	`x <= y`
>=	Greater than or equal to	`x >= y`

Table 2.4 is the list of the **comparison operators** provided by Python to compare two values. It returns either `True` or `False` according to the given condition. All comparison operators have the same order of precedence which is lower than that of the arithmetic operators. Typically, comparison operators are used in Boolean contexts like conditional and loop statements to control the flow of a program. Note that the result of comparison of different types of values using the the equality operator == is always `False`. Comparing two values of different types using the <, >, <=, and >= operators may also yield a `TypeError`. Besides, we should not use the equality operator == to determine whether two floating point numbers are equal. We should instead do this comparison by checking whether they are close to one another within some tolerance as follows:

```
>>> a = 2.22 + 3.33
>>> a == 5.55
```

```
False
>>> tolerance = 0.00001
>>> abs(a - 5.55) < tolerance
True
```

Note that the expression `abs(a - 5.55)` gives the absolute value of `a - 5.55`.

Table 2.5: Logical operators in Python

Operator	Meaning	Syntax
`or`	Return `True` if either of the operands is `True`	`x or y`
`and`	Return `True` if both the operands are `True`	`x and y`
`not`	Return `True` if the operand is `False`	`not x`

Python also provides **logical operators** to perform boolean operations. Table 2.5 shows theses operators sorted in ascending order of precedence. All logical operators have lower order of precedence than the comparisons operators. For example,

```
>>> x = 8
>>> (x < 3) and (x < 9)
False
>>> (x < 3) or (x < 9)
True
>>> not ((x < 3) and (x < 9))
True
```

Table 2.6: Common compound assignment operators in Python

Operator	Syntax	Equivalent to
`+=`	`x += 3`	`x = x + 3`
`-=`	`x -= 3`	`x = x - 3`
`*=`	`x *= 3`	`x = x * 3`
`/=`	`x /= 3`	`x = x / 3`
`//=`	`x //= 3`	`x = x // 3`
`%=`	`x %= 3`	`x = x % 3`
`**=`	`x **= 3`	`x = x ** 3`

We have already deal with the simple assignment operator =. For convenience, Python provides a number of **compound assignment operators** to assign values of variables. Table 2.6 shows the common compound assignment operators in Python. The use of these operators makes our code more compact. Some people also prefer using them because a statement like `x = x + 3` can be confusing as it looks like bad algebra. Below is an interactive Python session showing how these operators work:

```
>>> a = 2
>>> a += 3
>>> a
5
>>> a *= 3
>>> a
15
```

2.2.5 Multi-line Statements

Sometimes we may have a long statement in a program. Python does not have any restriction on the length of a statement. But we should break a long statement into multiple lines to enhance readability. One of the way to do this is implicit line continuation. In Python, any statement containing opening parentheses (`(`), brackets (`[`), or curly braces ({) is presumed to be incomplete until all matching parentheses, brackets, and braces have been encountered. Thus the statement can be implicitly continued across lines without raising an error before encountering the corresponding matching symbols. A long statement can be also split across multiple lines by wrapping it in grouping parentheses. For example,

```
>>> a = (1 + 2 + 3
... + 4 + 5 + 6
... + 7 + 8 + 9)
>>> print(a)
45
>>> colors = ["red",
... "blue",
... "green"]
>>> print(colors)
['red', 'blue', 'green']
```

Alternatively, we can split a long statement by explicit line continuation. Normally, the end of a statement is marked by a newline character (which we get by pressing Enter on our keyboard). If the statement is not complete by that point, Python will raise a

`SyntaxError` and the execution will stop. To indicate explicit line continuation, we can specify a backslash (\) character as the final character on the line. Then Python ignores the following newline character and the statement is effectively continued on next line:

```
>>> a = 1 + 2 + 3 \
... + 4 + 5 + 6 \
... + 7 + 8 + 9
>>> print(a)
45
```

Note that the backslash character must be the last character on the line. Not even whitespace is allowed after it. On the other hand, multiple statements can be placed on a single line by separating them with semicolons. For example,

```
>>> a = 1; b = 2; c = 3
>>> print(a, b, c)
1 2 3
```

In practice, your should place only one statement per line to make your program more readable.

2.2.6 Functions, Modules, and Packages

Recall that a function in Python is a block of organized and reusable code used to perform a single action. We can pass data known as arguments into a function and a function can return data as a result. Python has a collection of built-in functions that are always available. Two useful Python built-in functions for numbers are the mathematical functions **`abs`** and **`round`**. The function **`abs`** returns the absolute value of an integer or a floating point number or returns the magnitude of a complex numbers as follows:

```
>>> abs(−4)
4
>>> abs(−6.0)
6.0
>>> abs(2+3j)
3.605551275463989
```

The function **`round`** returns the nearest integer to a floating point number as follows:

```
>>> round(7.38)
7
```

```
>>> round(8.5)
8
>>> round(9.5)
10
```

Note that in Python 3, this function returns the nearest even integer if the number is midway between two integers.

Python also has a large number of functions defined in modules (files of Python definitions and statements) and packages (collections of relevant modules). They are *imported* if they are needed but are not loaded by default. Such practice minimizes the memory required to run a Python program and thus improves its performance. Table 2.7 lists some of the major, freely available Python modules and packages for numerical and scientific work. Some are installed with the core Python distribution, i. e. the **Standard Library** (see a complete list of its components at **https://docs.python.org/3/library/index.html**) and others can be downloaded and installed separately.

Table 2.7: Python modules and packages for numerical and scientific work

Module/Package	**Description**
`cmath`	Mathematical functions for complex numbers
`datetime`	Manipulation of dates and times
`decimal`	Decimal fixed point and floating point arithmetic
`fractions`	Rational number arithmetic
`math`	Mathematical functions
`random`	Random number generator
`statistics`	Mathematical statistics functions
Matplotlib (`matplotlib`)	Two-dimensional Plotting
NumPy (`numpy`)	Numerical and scientific computing
Pandas (`pandas`)	Data manipulation and analysis with table-like data structures
SciPy (`scipy`)	Scientific computing algorithms
SymPy (`sympy`)	Symbolic computation
VPython (`vpython`)	Three-dimensional visualization

We can use modules to split large programs into small, manageable, and organized files which makes the code easier to understand and use. Modules also support reusability of code. Each Python module has its own private **namespace** which serves as the global namespace for all the objects defined in the module. We can regard a namespace in

Python as a container for mapping each name that we have defined to the corresponding object (i. e. a table of defined symbols). Different namespaces may coexist at the same time but they are completely isolated. So it's possible for two different modules to have same names without any collision.

There are various ways to import the definitions inside a module to another module or the interactive Python session. For instance, we can import an entire module using a Python statement of the form `import <modname>` and access its content using the dot operator as follows:

```
>>> import math
>>> math.sqrt(144)
12.0
```

This form of import statement allows the name of the module to be placed into the caller's namespace. However, the objects that are defined in the module remain in the module's private namespace.

Alternatively, we can use a Python statement of the form `from <modname> import *` to import an entire module and access its content directly:

```
>>> from math import *
>>> exp(3.0)
20.085536923187668
```

We can also import individual objects from a module and access these objects directly using a Python statement of the form

```
from <modname> import <name1>, <name2>, ..., <nameN>
```

Here `<name1>, <name2>, ..., <nameN>` are the names of the definitions to be imported from the module `<modname>` so that any existing objects with these names will be overwritten. To call the imported module functions, one or more numbers need to be passed to them inside parentheses so that these numbers act as their arguments. For example,

```
>>> from math import log, factorial
>>> log(3.0)
1.0986122886681098
>>> factorial(6)
720
```

Although the last two approaches are convenient for accessing the content of a module, it is not recommended in Python programs. There is a danger of name conflicts and makes it difficult to know which function comes from which module. Importing with the `import` statement keeps the functions bound to their module's namespace. Thus the code would be much easier to understand and maintain even though it requires more typing.

Table 2.8: Commonly used mathematical functions provided by the `math` module

Syntax	**Meaning**
`sqrt(x)`	Square root of `x`
`exp(x)`	Exponential of `x`
`log(x)`	Natural logarithm of `x`
`log10(x)`	Logarithm of `x` to base 10
`sin(x),cos(x),tan(x)`	Sine, cosine, tangent of `x`
`asin(x),acos(x),atan(x)`	Arcsine, arccosine, arctangent of `x`
`sinh(x),cosh(x),tanh(x)`	Hyperbolic sine, cosine, tangent of `x`
`factorial(x)`	Factorial of `x`
`ceil(x)`	Nearest larger integer to `x`
`floor(x)`	Nearest smaller integer to `x`

One of the most useful Python modules is the `math` module[1] which contains all the standard mathematical functions. This module also contains the mathematical constants e and π which are denoted as `e` and `pi`. Note that it only provides the mathematical functions for floating point and integer operations (functions for complex numbers are provided by the `cmath` module). Table 2.8 is a list of the commonly used mathematical functions provided by the `math` module. Notice that all the trigonometric functions work with angles specified in radians instead of degrees. For example,

```
>>> import math
>>> math.tan(math.pi/3)
1.7320508075688767
>>> math.acos(0.5)
1.0471975511965979
```

Besides, the exponential and square root functions may seem redundant since both exponentials and square roots can be calculated by taking powers. For instance, `x**0.5` would give the square root of `x`. However, using these functions is usually much faster and more accurate due to the way that the computer calculates powers and roots.

The `random` module is another useful Python module for generating random numbers

[1]We may regard the `math` module as a package with a single module. The words package and module are often used interchangeably in such case.

Table 2.9: Commonly used functions provided by the `random` module

Syntax	**Meaning**
`random()`	Select a random floating point number from the semi-open interval $[0, 1)$
`uniform(a, b)`	Select a random floating point number in the closed interval [`a`, `b`]
`normalvariate(mu, sigma)`	Generate a number from the normal distribution with mean `mu` and standard deviation `sigma`
`randint(a, b)`	Select a random integer from the closed interval [`a`, `b`]
`choice(seq)`	Select an item at random from the sequence `seq`
`shuffle(seq)`	Randomly shuffles (permutes) the items of the sequence `seq` in place
`sample(population, k)`	Draw a list of `k` unique elements from the sequence or set `population` without replacement

and data. Roughly speaking, a random number is a number drawn by chance from a pool of numbers that has no discernible pattern for prediction. In computational physics, it's often necessary to generate random numbers from a given probability distribution[2] for solving problems. Like many other languages, Python implements a **pseudorandom number generator** (PRNG) which uses a deterministic formula to generate a sequence of numbers that approximates the properties of "truly" random numbers from a uniform distribution for which all possible values are equally probable. Such sequence is determined by the **seed** state and it's always the same following the same seed. Any PRNG would yield a sequence that eventually repeats and a good generator should have a long period. The Python `random` module uses a **Mersenne Twister** PRNG which is good enough for serious physics calculations. When this module is first imported, it's seeded with the current system time or a randomness source from your operating system if one is available. We can reset the seed value of the PRNG at any time with a call to `random.seed`. Table 2.9 shows the commonly used functions provided by the `random` module. Below is an interactive Python session demonstrating how to work on these functions:

```
>>> import random
>>> random.random()
```

[2]A random variable is a variable whose value is the outcome of a random event. A probability distribution of a random variable (e. g. a uniform distribution) is a function that gives a description of each possible value of the variable with its probability of occurrence.

```
0.9775988981697202
>>> random.seed(101)
>>> random.random()
0.5811521325045647
>>> random.random()
0.1947544955341367
>>> random.seed(101)
>>> random.random()
0.5811521325045647
>>> random.uniform(−3.,3.)
−1.8314730267951798
>>> random.uniform(−3.,3.)
2.7915066423666666
>>> random.normalvariate(10, 1)
11.364976704270461
>>> random.normalvariate(10, 1)
10.357029447652616
>>> random.randint(1, 8)
4
>>> random.randint(1, 8)
5
>>> seq = [2, "XL", −5.6, 72, 2.56, "Win"]
>>> random.choice(seq)
72
>>> random.choice(seq)
'Win'
>>> random.shuffle(seq)
>>> seq
[2.56, 'Win', 2, 72, −5.6, 'XL']
>>> numbers = range(1, 501)
>>> choices = random.sample(numbers, 5)
>>> choices
[99, 83, 46, 228, 395]
```

Packages allow for a hierarchical structuring of the Python's module namespace using dot notation. A package can contain multiple subpackages and modules. In the same way that modules help avoid collisions between global variable names, packages help avoid collisions between module names. We can import a package, a subpackage, and a module within a package like a "normal' module. For example, we can import the `fft` subpackage of the `numpy` package using the statement `from numpy import fft` and the `inv` function from the `linalg` subpackage of this package using the statement `from numpy.linalg import inv`.

2.2.7 Input and Output Statements

So far we have only considered the assignment statement in Python programs. Here, we are going to examine **input statement** and **output statement**, i. e. the statements for input and output of data. To collect the information from the user, we can use an input statement of the form

```
<variable> = input(<prompt>)
```

where <*prompt*> is a string used to prompt the user for input of <*variable*>. When Python encounters a call to `input`, it prints the prompt on the screen. It then pauses and waits for the user to input some text and press the **Enter** key. Whatever the user types is then stored as a string. Consider this simple interaction with the Python interpreter:

```
>>> jobtitle = input("Enter your job title: ")
Enter your job title: Assistant Manager
>>> jobtitle
'Assistant Manager'
```

In this interaction, the user has input `Assistant Manager`. So the variable `jobtitle` is assigned to the string `'Assistant Manager'`. Thus the evaluation of `jobtitle` yields the string of input characters. Besides, a blank space is added inside the quotes at the end of the prompt. Then the user input does not start right next to the prompt which makes the interaction easier to read and understand.

If the user input is a number, then we need an input statement of a slightly more complicated form:

```
<variable> = eval(input(<prompt>))
```

Here the `input` function is "wrapped around" by another built-in function `eval` which stands for evaluation. In this form, the text input by the user is evaluated as an expression to produce the value that is assigned to the variable. For example,

```
>>> salary = eval(input("Enter your salary: "))
Enter your salary: 10800
>>> salary
10800
```

In this example, `salary` points to the object holding the integer value `10800`. Remember that we need to **`eval`** the input whenever we want a number instead of a string.

In the previous example, the user is prompted to enter a number and the user just typed in a numeric literal. Indeed, any valid expression would be also acceptable. Consider the following interactive Python session:

```
>>> hours = eval(input("Enter an expression for hours: "))
Enter an expression for hours: 3*2 + 5
>>> print(hours)
11
```

After the user typed `3*2 + 5`, this expression is evaluated via **`eval`** and the resultant value is assigned to the variable `hours`. The print statement showed that `hours` has obtained the expected value `11`. The **`input`**—**`eval`** combination is like a delayed expression in the sense that it produced exactly the same result as if we had simply written `hours = 3*2 + 5`.

Beware that the **`eval`** function is very powerful but *also potentially dangerous*. It may lead to security risk since the user are essentially allowed to enter a portion of the program when the user input is evaluated. For example, the user could type an expression that captures private information or deletes files on the computer. As a result, the use of **`eval`** could be disastrous if the input to a program is coming from untrusted sources like users on the internet. Fortunately, we can use type conversion functions like **`int`** and **`float`** as safer alternatives to **`eval`**. For instance, the previous example can be rewritten as

```
>>> salary = int(input("Enter your salary: "))
Enter your salary: 10800
>>> salary
10800
```

We have already seen examples of using the Python's built-in function **`print`** to display information on the screen in textual form. The basic form of a **`print`** statement is like this:

```
print(<expr>)
```

where <*expr*> is the expression to be displayed on the screen. To display multiple expressions on screen, we use a `print` statement of the form:

```
print(<expr1>, <expr2>, ... , <exprN>)
```

where `<expr1>, <expr2>, ... , <exprN>` are the expressions to be displayed on the screen. Any supplied expressions are evaluated from left to right and the results are displayed on a line of output in a left-to-right fashion. By default, a single blank space character is placed between the displayed results. As an example, consider the following interaction with the Python interpreter:

```
>>> print(5*6)
30
>>> print(5, 6, 5*6)
5 6 30
>>> print("The answer is", 5*6)
The answer is 30
```

In fact, the `print` function automatically appends some ending text after all of the supplied expressions are printed. By default, the ending text is a special marker character (denoted as \n) that indicates the end of a line. So successive `print` statements normally display the results on separate lines of the screen. And a `print` without any arguments produces a blank line of output. To customize the ending text, we include an additional argument for the `print` function as follows:

```
print(<expr1, <expr2>, ... , <exprN>, end=<mc>)
```

where `<mc>` is the marker character specifying the ending text. For example, a Python program contains the following statements

```
print("The answer is", end=" ")
print(5*6)
```

produces the single line of output

```
The answer is 30
```

The first `print` statement produces an output ending with a space (`" "`). Thus the output from the second statement appears immediately following the space.

2.3 Control Structures

Up to now, we have only consider the case in which the statements in a Python program are executed sequentially one after the next. However, sometimes we may want to execute some statements only if a certain condition is met or execute a group of statements repeatedly. In Python, multiple statements can be grouped together into a **compound statement** to control their order of execution. A compound statement consists of one or more **clauses** aligned at the same indentation. Each clause has a **header** starting with a uniquely identifying keyword and ending with a colon (`:`), followed by a **body** which is a sequence of one or more statements. The statements in the body should be placed on separate lines after the header and aligned with the same further indentation. The clause ends when the indentation returns to that of its header. Let us discuss Python compound statements one by one.

Decision making is required when we want to execute a code only if a certain condition is satisfied. Python provides **`if`** statement for making **one-way** or **simple** decisions. The syntax of this statement is

```
if <condition>:
    <body>
```

Here <*`body`*> is a statement block and <*`condition`*> is a Boolean expression that produces a Boolean values when it is evaluated. When the Python interpreter encounters an **`if`** statement, it first evaluates <*`condition`*> in the header. If this expression evaluates to `True`, the statements in <*`body`*> are executed; otherwise, the statements in <*`body`*> are skipped. In either case, the Python interpreter then moves on to the next statement right after this statement. As an example, consider the following Python code:

```
x = int(input("Enter a whole number no greater than five: "))
if x > 5:
    print("You entered a number greater than five.")
    print("Let me fix that for you.")
    x = 5

print("Your number is", x)
```

It modifies the value of the input number `x` only if it is greater than five. Below are the sample input and output of this code:

```
Enter a whole number no greater than five: 8
You entered a number greater than five.
Let me fix that for you.
Your number is 5
Enter a whole number no greater than five: 3
```

```
Your number is 3
Enter a whole number no greater than five: -2
Your number is -2
```

In Python, Boolean expressions are constructed by using comparison and logical operators and they may compare either numbers or strings. When comparing strings, strings are put in alphabetic order according to the underlying Unicode values. So all uppercase Latin letters come before lowercase equivalents.

We can combine two boolean expressions in a single **if** statement using **and** or **or** operators such as:

```
if (x > 10) or (x < 1):
    print("Your number is less than 1 or greater than 10.")
if (x <= 10) and (x >= 1):
    print("Your number is between 1 and 10.")
```

We can combine any number of conditions in a single **if** statement in a similar way.

Python also provides **if–else** statements for making **two-way** decisions. The syntax of this statement is

```
if <condition>:
    <bodyif>
else:
    <bodyelse>
```

Here the Python interpreter again first evaluates the Boolean expression in the heading (i. e. <*condition*>). If this expression evaluates to `True`, then the statement block under the **if** (i. e. <*bodyif*>) is executed; otherwise, the statement block under the **else** (i. e. <*bodyelse*>) is executed. In either case, the interpreter then moves on to the next statement after the **if–else** statement. For example, the following Python routine:

```
x = int(input("Enter a whole number: "))
if x > 10:
    print("Your number is greater than ten.")
else:
    print("Your number is less than or equal to ten.")
```

prints different messages depending on whether `x` is greater than ten or not.

Furthermore, Python provides **if–elif–else** statement for making **multi-way** decisions. The syntax of this statement is

```
if <condition1>:
    <body1>
elif <condition2>:
    <body2>
⋮
else:
    <bodyE>
```

where `<condition1>, <condition2>, ...` are Boolean expressions. The keyword **`elif`** means "else if" which allows us to check for multiple conditions. An **`if`** statement can have multiple **`elif`** blocks but it can have only one **`else`** block. In an **`if`**–**`elif`**–**`else`** statement, the Python interpreter evaluates the condition in each clause one by one and the statement block under the condition that first evaluates to `True` are executed. If none of these conditions evaluates to true, the statement block under the **`else`** are executed. It's always only one of the statement blocks of this statement (i. e. `<body1>, <body2>, ..., <bodyE>`) that would be executed according to the conditions. After the execution of the statement block, the interpreter jumps to the next statement right after this statement. Below is the Python code illustrating how to use the **`if`**–**`elif`**–**`else`** statement:

```
x = float(input("Enter a number: "))
if x < 0:
    print("Your number is negative.")
elif x == 0:
    print("Your number is zero.")
else:
    print("Your number is positive.")
```

In Python, the body of a clause of a compound statement can contain any legal Python statements including other compound statements. Placing a compound statement inside another compound statement is called **nesting**. For instance, we can rewrite the previous Python code example by nesting two **`if`**–**`else`** statements as follows:

```
x = float(input("Enter a number: "))
if x <= 0:
    if x == 0:
        print("Your number is zero.")
    else:
        print("Your number is negative.")
```

```
else:
    print("Your number is positive.")
```

Sometimes we may want to execute a code in an **indefinite** loop that keeps iterating until certain conditions are met. In Python, an indefinite loop is implemented using a `while` statement. The syntax of this statement is

```
while <condition>:
    <body>
```

Here <*condition*> is a Boolean expression and <*body*> is a statement block just like that in an `if` statement. When the Python interpreter encounters a `while` statement, it executes <*body*> repeatedly as long as <*condition*> remains `True`. Beware that the `while` loop is a **pre-test loop** in which the condition is always tested at the top of the loop before the body of the loop is executed. If the loop condition is initially `False`, the loop body will not execute at all. As with an `if` statement, we can specify two or more conditions in a single `while` statement using `and` or `or` operators. Below is an example of a simple `while` loop that counts the even numbers from 0 to 50:

```
i = 0
while i <= 50:
    print(i)
    i = i + 2
```

In the above example, we must initialize `i` before the loop for correct counting.

The `while` statement is both powerful and dangerous due to its simplicity. Since it is less rigid, it is more versatile and can do more than just iterate through sequences. However, it is also easy for us to make errors in using the `while` statement. For example, if we forget to increment `i` at the bottom of the loop body in the previous example, then the loop will be executed endlessly, printing a zero each time. This is an example of an **infinite loop** which is usually a bad thing. Typically, we can break out of an infinite loop by pressing **⟨Ctrl⟩-c** (i. e. holding down the **⟨Ctrl⟩** key and pressing the button **c**). If this does not work, then we'll have to resort to more drastic means such as pressing **⟨Ctrl⟩-⟨Alt⟩-⟨Delete⟩** on a PC. If all else fails, we can always press the **reset** button on our computer. Of course, the best idea is to avoid writing infinite loops in the first place.

In some cases, we may prefer running a code in a **definite** loop that would be executed a definite number of times. In Python, a definite loop is implemented using a `for` statement. The syntax of this statement is

```
for <var> in <sequence>:
    <body>
```

Here <*body*> is a statement block, <*var*> is a variable called the **loop index**, and <*sequence*> is a sequence (e. g. list and string). When the Python interpreter encounters this statement, it executes <*body*> for <*var*> taking on successive values in <*sequence*>. Usually, <*sequence*> is a list of values. We will learn more about lists in section 2.5. For now, we only need to know that a simple list can be created by placing a sequence of expressions in square brackets separated by commas. Below is the interaction with the Python interpreter showing how the `for` statement works with a simple list:

```
>>> for i in [1, 2, 3, 4, 5]:
...     print(i)
...
1
2
3
4
5
>>> for fruit in ["apple", "banana", "melon", "orange"]:
...     print(fruit)
...
apple
banana
melon
orange
```

From the above examples, we can see that the length of the list determines the number of times that the loop executes.

Python provides a handy built-in function `range` for generating a sequence of numbers. Its simplest form is `range(n)` that produces a sequence of numbers starting with 0 and continuing up to, but not including `n`. With two arguments, `range(start, n)` produces a sequence of numbers starting with the value of `start` and continuing up to, but not including `n`. A third version `range(start, n, step)` is like the two-argument version, except that it uses `step` as the increment between numbers. To understand how the `range` function works, consider the following examples:

```
>>> list(range(10))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
>>> list(range(5, 10, 2))
[5, 7, 9]
```

Here **list** is another built-in function asking Python to turn a range into a plain old list.

In general, **range**(<*expr*>) produces a sequence of numbers that starts with 0 and goes up to, but does not include, the value of <*expr*>. Therefore, if we want to do something in our program for a certain number of times, use a **for** loop of the form

```
for <var> in range(<expr>):
    <body>
```

in which the value of <*expr*> determines how many times that the loop would be executed. This particular loop pattern is called a **counted loop** which is a very common use of definite loops. We are free to choose any name for the loop index of a counted loop. Just be sure to use an identifier that has not been used for any other purpose. Otherwise we might accidentally wipe out a value that we will need later. Indeed, programmers often use `i` or `j` as the name of the loop index for counted loops. As an example, below is a counted loop that computes the sum of integers from 1 to 50:

```
sum = 0
for i in range(51):
    sum = sum + i
print("The sum of the integers from 1 to 50 is", sum)
```

Python provides three further statements to be used with compound statements for controlling the flow of a program. The **break** statement asks Python to immediately exit the enclosing loop. It is often used to leave an infinite loop as follows:

```
x = 0
while True:
    x += 1
    if not ((x % 15) or (x % 25)):
        break

print(x, "is divisible by both 15 and 25")
```

The **while** loop condition here is always `True`. So the only escape from the loop occurs when the **break** statement is reached. This occurs only when the counter `x` is divisible by both 15 and 25. So the output of the above example is:

```
75 is divisible by both 15 and 25
```

Of course, we can use the **break** statement in a loop that does not executes endlessly. For example, the following Python routine finds the index of the first occurrence of a negative number in a list:

```
alist = [0, 4, 5, −2, 5, 10]
i = 0
negfound = False
for a in alist:
    i += 1
    if a < 0:
        negfound = True
        break

if negfound:
    print(a, "occurs at index", i)
else:
    print("No negative number is found")
```

Note that after escaping from the loop, the loop index `a` has the value that it had inside the loop at the **break** statement.

The **continue** statement acts in a similar way to **break** but instead of breaking out of the enclosing loop, it immediately forces the next iteration of the loop without completing the statement block for the current iteration. For example, the following Python code

```
for i in range(1, 11):
    if not (i % 2):
        continue
    print(i, "is odd!!")
```

produces the output:

```
1 is odd!!
3 is odd!!
5 is odd!!
7 is odd!!
```

```
9 is odd!!
```

In the above example, if `i` is an even number (and hence `i % 2` is equal to 0 which is equivalent to `False`), the **`print`** statement is skipped and the loop is continued with the next value of `i`.

A **`pass`** statement does nothing. It simply fills in the spot where Python is syntactically expecting a statement to be. Below is an interactive Python session showing how it works:

```
>>> for i in range(1, 20):
...     if i == 10:
...         pass
...     if not (i % 5):
...         print(i, "is divisible by 5.")
...

5 is divisible by 5.
10 is divisible by 5.
15 is divisible by 5.
```

If we replace the **`pass`** statement by a **`continue`** statement, the line `10 is divisible by 5.` would not be printed as the execution would return to the top of the loop for the next value of `i`.

A **`for`** or **`while`** loop may be followed by an **`else`** block of statements, which will be executed only if the loop finished "normally" (i. e. without the enccounter of a **`break`**). For **`for`** loops, it means that these statements will be executed after the loop has reached the end of the sequence it is iterating over; for **`while`** loops, they are executed when the while condition becomes `False`. For example, our Python routine for finding the first occurrence of a negative number in a list can be rewritten as follows:

```
alist = [0, 4, 5, 2, 5, 10]
i = 0
for a in alist:
    i += 1
    if a < 0:
        print(a, "occurs at index", i)
        break
else:
    print("No negative number is found")
```

Here the statement `print("No negative number is found")` would be executed only if the `break` statement inside the `for` loop has not been executed (i. e. `alist` does not contain any negative number).

Another example is the Python code for finding the largest factor of a number `x` $\geq$ `2`:

```
x = 2031
y = x − 1
while y != 1:
    if not (x % y):
        print("The largest factor of", x, "is", y)
        break
    y −= 1
else:
    print(x, "is a prime number")
```

If the largest factor of `x` is found in the `while` loop, Python will output this factor and then jump out from this loop. Otherwise, the `else` block of statements will be executed after the `while` loop is finished (in such case `x` is prime).

Finally, consider the following Python programs with their sample input and output that illustrate the application of compound statements in more complicated cases.

```
# quadratic.py
# This program finds the real roots to a quadratic equation with ita
# coefficients given.
# Last Update on 5 Jan 2022 by F K Chow

from math import sqrt

print("This program finds the real roots to a quadratic equation")
print("in the form of ax^2 + bx + c = 0.")

a = float(input("Enter the coefficient a: "))
b = float(input("Enter the coefficient b: "))
c = float(input("Enter the coefficient c: "))

# Find the real roots for different cases
```

```
discrim = b*b - 4*a*c
if discrim < 0:
    print("The equation has no real roots!")
elif discrim == 0:
    root = -b/(2*a)
    print("There is a double root at {0:f}.".format(root))
else:
    discRoot = sqrt(b*b - 4*a*c)
    root1 = (-b + discRoot)/(2*a)
    root2 = (-b - discRoot)/(2*a)
    print("The roots are {0:f} and {1:f}.".format(root1, root2))
```

```
================ RESTART: D:/PythonCodes/quadratic.py ================
This program finds the real roots to a quadratic equation
in the form of ax^2 + bx + c = 0.
Enter the coefficient a: 1
Enter the coefficient b: 6
Enter the coefficient c: 9
There is a double root at -3.000000.

================ RESTART: D:/PythonCodes/quadratic.py ================
This program finds the real roots to a quadratic equation
in the form of ax^2 + bx + c = 0.
Enter the coefficient a: 2
Enter the coefficient b: 5
Enter the coefficient c: 3
The roots are -1.000000 and -1.500000.
```

```
# piestimation2.py
# This program estimates the value of pi using random numbers. It
# stops when the estimated value of pi is close to the actual value.
# Last Update on 19 Nov 2020 by F K Chow

from math import pi
from math import sqrt
from random import random

epsilon = 1E-6 # Maximum error for the approximation to pi
nc = 0 # Number of points inside the quarter circle
ns = 0 # Number of points inside the unit square

print("{0:>6s} {1:>13s} {2:>10s}".format("Ns", "Estimated pi",
                                          "Error"))

while True:
    ns += 1
    # Generate a random point in the unit square
    x = random()
    y = random()
    # Decide whether it is inside the quarter circle
    if sqrt(x*x + y*y) <= 1:
        nc += 1
    # Compute the estimated value of pi
    piEst = 4*nc/ns
    if not ns % 1000:
        print("{0:6d} {1:13.6f} {2:10.6f}".format(ns, piEst, error))
    # Stop if the error of the estimated value is small
    error = pi - piEst
    if abs(error) <= epsilon:
        break

# Print the final estimated value of pi
print("After {0:d} iteration, we find the".format(ns), end=" ")
print("approximation to pi")
print("with error <= {0:f} to be {1:f}".format(epsilon, piEst))
```

```
============== RESTART: D:/PythonCodes/piEstimation2.py ==============
    Ns  Estimated pi      Error
  1000      3.128000   0.014466
  2000      3.110000   0.030037
  3000      3.126667   0.015217
  4000      3.128000   0.013811
  5000      3.137600   0.003365
  6000      3.138000   0.003070
  7000      3.143429  −0.001714
After 7451 iteration, we find the approximation to pi
with error <= 0.000001 to be 3.141592
```

2.4 Strings

Python strings are of type **str** which represents ordered, immutable sequences of characters. Recall that a single-line string literal is defined by enclosing text in single quotes (`''`) or double quotes (`""`). In addition, we can use triple-single quotes (`'''`) or triple-double quotes (`"""`) to define multi-line string literals as follows:

```
>>> paragraph = """This is the first line.
... This is the second line.
... This is the third line."""
>>> type(paragraph)
<class 'str'>
>>> print(paragraph)
This is the first line.
This is the second line.
This is the third line.
```

This is often used to create "docstrings" which document blocks of code in a program.

There is no restriction for the length of a line in Python. So a string literal can be defined in a single, quoted block of text. However, it is better to keep the lines of our program to a fixed maximum length for ease of reading (79 characters is recommended). To break up a string over two or more lines of code, we can use the line continuation character (\) or enclose the string literal in parentheses. For example,

```
>>> long_string = "Once upon a time, there was a beautiful princess"\
... " living in a castle in a far far away country ..."
>>> long_string = ("Once upon a time, there was a beautiful princess"
... " living in a castle in a far far away country ...")
```

This defines the variable `long_string` to hold a single line of text with no carriage return. The concatenation does not insert any spaces and so they need to be included explicitly if they are wanted.

Python provides the built-in function **`str`** to convert an object passed as its argument into a string according to a set of rules defined by the object itself:

```
>>> str(500)
'500'
>>> str(4/3)
'1.3333333333333333'
>>> str(1.8e6)
'1800000.0'
```

If we want to include the quote character inside a string literal, then we need to use the quote other than that enclosing the string itself as follows:

```
>>> sentence = "Peter said, 'I am hungry.'"
>>> print(sentence)
Peter said, 'I am hungry.'
```

However, if we want to include both quotes or more than one line in a string, then we should use **escape sequences** which are combinations of characters that have special meanings. In Python, escape sequences are indicated with a backslash (\). Table 2.10 shows the commonly used escape sequences in Python. Below is an interactive Python session demonstrating the use of these sequences.

```
>>> print("\\ is the backslash")
\ is the backslash
>>> print("\"Our Country\'s Good\"")
"Our Country's Good"
>>> print("Beautiful\ngirl")
Beautiful
girl
>>> print("Beautiful\tgirl")
Beautiful       girl
>>> print("Beautiful\rgirl")
girltiful
>>> print("Beautiful\bgirl")
Beautifugirl
```

Table 2.10: Commonly used escape sequences in Python

Escape Sequence	**Meaning**	**What it does**
`\\`	Backslash (\\)	Enter a backslash into the string
`\'`	Single quote (')	Enter a single quote into the string
`\"`	Double quote (")	Enter a double quote into the string
`\n`	ASCII Linefeed	End this line and move to a new one
`\t`	ASCII Horizontal Tab	Move to the right to the next tab stop
`\r`	ASCII Carriage Return	Return the printing position to the start of the line
`\b`	ASCII Backspace	Remove the previous character
`\a`	ASCII Bell	Sound the bell on the terminal
`\uxxxx`	16-bit hex value Unicode character	Enter a Unicode character with 16-bit hex value *`xxxx`* into the string
`\UXXXXXXXX`	32-bit hex value Unicode character	Enter a Unicode character with 32-bit hex value *`XXXXXXXX`* into the string
`\N{name}`	Unicode character in the Unicode database	Enter a Unicode character called *`name`* in the Unicode database into the string

On the other hand, if we want to define a string to include character sequences such as `\t` *without them being escaped*, then the string must be defined with a prefix `r`:

```
>>> mystring = r"The escape sequence for a horizontal tab is \t."
>>> print(mystring)
The escape sequence for a horizontal tab is \t.
```

In Python 3, strings are composed of **Unicode** characters. Unicode is a standard encoding for the text representation of the letters and symbols used with different languages and scripts. It works by assigning a unique number to every character and the numbers that make up a string are then encoded as a sequence of bytes. Note that Unicode uses the same codes as **ASCII** (American Standard Code for Information Interchange) for the 127 characters originally defined there. By default, Python 3 use the UTF-8 encoding which is the most widely used Unicode encoding standard today. If our editor does not allow us to enter a Unicode character directly into a string literal, we can use its 16- or 32-bit hex value or its Unicode character name as an escape sequence as follows:

```
>>> print("\u01A9")
Ʃ
>>> print("\U000001A9")
```

```
Σ
>>> print("\N{LATIN CAPITAL LETTER ESH}")
Σ
```

Python provides several built-in functions that allow us to switch back and forth between characters and the numeric values used to represent them in strings. The **ord** function returns the numeric ("ordinal") code of a single character string, while **chr** works in the reverse manner. For examples,

```
>>> ord("j")
106
>>> ord("J")
74
>>> chr(74)
'J'
>>> chr(106)
'j'
```

When we do Python programming, we may need to find the length of a string, i. e. how many characters are in the string. We can use the built-in function **len** to do so:

```
>>> a = "Chocolate"
>>> len(a)
9
```

Python strings can be putting together using either the concatenation operator + or by placing them next to each other on the same line. Below are the interaction with the Python interpreter demonstrating how to put strings together:

```
>>> "Peter" + " Jackson"
'Peter Jackson'
>>> "Apple " "and " "banana"
'Apple and banana'
```

If our string consists of a repetition of one or more characters, the repetition operator ∗ can be used to concatenate them the required number of times as follows:

```
>>> "dog"*4
'dogdogdogdog'
```

```
>>> ("dog"*3) + ("cat"*5)
'dogdogdogcatcatcatcatcat'
```

Note that the empty string is defined as `s = ''` or `s = ""`.

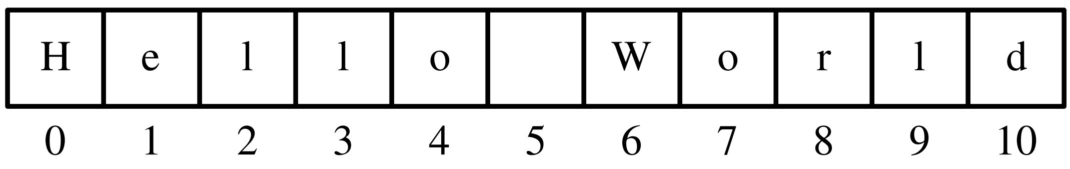

Figure 2.2: Indices of the string `Hello World` in Python

Indexing is used in strings to access a character at a specific location of the string. Like all sequences in Python, strings are indexed from left to right with the first character having the index 0. Figure 2.2 illustrates the indexing scheme for the string `Hello World`. The syntax for selecting a character from a string is <*string*>[<*expr*>] where the value of <*expr*> determines the character to be selected from <*string*>. Any attempt to index a string outside its length yields an `indexError`. Below is an interactive Python session demonstrating how to access characters in string by index:

```
>>> message = "Hello World"
>>> message[0]
'H'
>>> print(message[1], message[4], message[7])
e o o
>>> x = 10
>>> print(message[x-3])
o
>>> message[12]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

Note that the last character in a string of length n has the index $n-1$ since the indexes start at 0. We should also beware of the difference between string objects and the actual printed output. In the above example, the Python shell displays the value of a string by putting it in single quotes. However, when we **`print`** the string, Python does not put any quotes around the sequence of characters.

Python also allows indexing from the right end of a string using negative indexes where the final character has the index −1. This is particularly handy for getting the last character of a string. For example,

```
>>> message[-1]
'd'
>>> message[-5]
'W'
```

Indexing a string returns a single character at a given location. It is also possible to access a continuous sequence of characters or **substring** from a string. In Python, this can be done through an operation called **slicing** using the slice operator `:`. We may regard slicing as a way of indexing a range of positions in the string. The simple syntax for a slice of a string is <*string*>[<*start*>:<*end*>] where both <*start*> and <*end*> are integer-valued expressions. It produces a substring of <*string*> starting from the index <*start*> and running up to, but not including, the index <*end*>. If the first index is omitted, it takes on the default value 0; if the second one is omitted, it takes on the default value **len**(<*string*>) so that the string is sliced to its end. Unlike indexing, slicing a string outside its length does not raise any error. Continuing with our previous example, here are some simple examples of string slicing in Python:

```
>>> message[3:9]
'lo Wor'
>>> message[:5]
'Hello'
>>> message[5:]
' World'
>>> message[:]
'Hello World'
>>> message[12:]
''
```

The more general syntax for a slice of a string is <*string*>[<*start*>:<*end*>:<*step*>] where <*step*> is an optional argument giving an integer value that determines the step of the slicing. If <*step*> is omitted, it takes on the default value 1, i. e. returning every character in the requested range. To return every kth letter in a string, set <*step*> to `k`. Note that negative values of <*step*> reverses the string. In such case, the default values of the indices <*start*> and <*end*> are changed to −1 and −**len**(<*string*>)−1, respectively. Below are the examples for this form of string slicing in Python:

```
>>> s = "Queen Mary"
>>> s[::2]
'QenMr'
>>> s[1::2]
'ue ay'
>>> s[-1:4:-1]
'yraM '
>>> s[4::-1]
'neeuQ'
>>> s[:3:-1]
'yraM n'
```

Since Python strings are immutable objects, it is not allowed to change a string by assignment. However, new strings can be constructed from existing objects as new objects. For example,

```
>>> x = "Fragrance"
>>> x += " Free"
>>> print(x)
Fragrance Free
>>> y = "Rose " + x[:9]
>>> print(y)
Rose Fragrance
```

To test whether a string contains a given substring, use the **in** operator:

```
>>> "Water" in "Lemon water"
False
>>> "water" in "Lemon water"
True
```

Because strings are sequences of characters, we can iterate through the characters using a **for** loop in Python. For example,

```
>>> for ch in "Chocolate":
...     print(ch, end=" ")
...
C h o c o l a t e >>>
```

Table 2.11 summarizes the basic string operations in Python that we have discussed so far. To illustrate these string operations, the program `month.py` plus its sample input and output are also shown after the table.

Table 2.11: Basic string operations in Python

Operation	**Meaning**
+	Concatenation
*	Repetition
<*string*>[]	Indexing
<*string*>[:]	Slicing
len(<*string*>)	Check length
<*substring*> in <*string*>	Check substring
for <*var*> in <*string*>	Iteration through characters

```
# month.py
# This program prints the abbreviation of a month given its number.
# Last Update on 19 Nov 2020 by F K Chow

# We use months as a lookup table
months = "JanFebMarAprMayJunJulAugSepOctNovDec"
m = int(input("Enter the number of a month (1-12): "))

# Compute the starting position of month m in months
# Then slice the abbreviation for month m from months
pos = (m - 1)*3
monthAbbrev = months[pos:pos+3]

# Print the abbreviation for month m
print("The month abbreviation is", monthAbbrev + ".")
```

```
================== RESTART: D:/PythonCodes/month.py ==================
Enter the number of a month (1-12): 9
The month abbreviation is Sep.
```

In Python, string objects have a large number of methods for manipulating and transforming them. These are accessed using the usual dot notation that we have already used.

Some of the more useful ones are listed in Table 2.12. Below is the interaction with the Python interpreter demonstrating some possible manipulations using string methods:

```
>>> x = "amy betty carrie diana"
>>> x.isalpha()
False
>>> x.split(" ")
['amy', 'betty', 'carrie', 'diana']
>>> x.strip("a")
'my betty carrie dian'
>>> x.upper()
'AMY BETTY CARRIE DIANA'
>>> y = x.title()
>>> y
'Amy Betty Carrie Diana'
>>> z = y.replace(" ", "!\t")
>>> z
'Amy!\tBetty!\tCarrie!\tDiana'
>>> print(z)
Amy!    Betty!  Carrie! Diana
>>> z.index("Carrie")
12
>>> z[:14].endswith("Ca")
True
>>> " ".join(["Mary", "is", "very", "happy!"])
'Mary is very happy!'
>>> "***A happy day***".lstrip("*")
'A happy day***'
```

Note that each string method returns a new string. So these methods can be chained together as follows:

```
>>> string = "*^*Python Wrangling for Beginners"
>>> string.replace("Wrangling", "Programming").upper().lstrip("*^*")
'PYTHON PROGRAMMING FOR BEGINNERS'
```

Table 2.12: Common string methods in Python

Method	**Description**
`center(`*`width`*`)`	Return the string centered in the original string with total number of characters *`width`*.
`startswith(`*`prefix`*`)`	Return `True` if the string starts with the substring *`prefix`*.
`endswith(`*`suffix`*`)`	Return `True` if the string ends with the substring *`suffix`*.
`index(`*`substring`*`)`	Return the lowest index in the string containing *`substring`*.
`lstrip(`*`chars`*`)`	Return a copy of the string with any of the leading characters specified by *`chars`* removed. If *`chars`* is omitted, any leading white space is removed. The *`chars`* argument is not a prefix; rather, all combinations of its values are stripped.
`rstrip(`*`chars`*`)`	Return a copy of the string with any of the trailing characters specified by *`chars`* removed. If *`chars`* is omitted, any trailing white space is removed. The *`chars`* argument is not a suffix; rather, all combinations of its values are stripped.
`strip(`*`chars`*`)`	Return a copy of the string with leading and trailing characters specified by *`chars`* removed. If *`chars`* is omitted, any leading and trailing white space is removed. The *`chars`* argument is not a prefix or suffix; rather, all combinations of its values are stripped.
`upper()`	Return a copy of the string with all characters in uppercase.
`lower()`	Return a copy of the string with all characters in lowercase.
`title()`	Return a copy of the string with all words starting with capitals and other characters in lowercase.
`replace(`*`old, new`*`)`	Return a copy of the string with each substring *`old`* replaced by *`new`*.
`split(`*`sep`*`)`	Return a `list` of substrings from the original string which are separated by the string *`sep`*. If *`sep`* is not specified, the separator is taken by any amount of whitespace.
`join(`*`list`*`)`	Use the string as a separator in joining a `list` of strings specified by *`list`*.
`isalpha()`	Return `True` if all characters in the string are alphabetic and the string is not empty; otherwise return `False`.

2.5 Lists and Tuples

A Python list is an ordered, mutable sequences of objects which is of type **list**. It can contain arbitrary objects such as strings, any type of numbers, built-in constants, and even other lists. Note that the items in a Python list need not be all of the same type. Python lists can be constructed by placing a sequence of comma-separated objects inside square brackets `[]`. An empty list is simply written as `[]`. For example,

```
>>> lst1 = [3, "four", 3, 1.66, 5]
>>> lst1
[3, 'four', 3, 1.66, 5]
>>> a = "S"
>>> lst2 = [1, a, −0.6, lst1, False]
>>> lst2
[1, 'S', −0.6, [3, 'four', 3, 1.66, 5], False]
```

Python lists are typically used to represent an ordered collection of related data.

Recall that both Python strings and lists are sequences. Indeed, the operations in Table 2.11 are valid for any kind of sequences. Thus we can index, slice, and concatenate lists in the same manner as that for strings. For instance, if `lst1` and `lst2` have the same definition as in the previous example, then

```
>>> lst1 + lst2
[3, 'four', 3, 1.66, 5, 1, 'S', −0.6, [3, 'four', 3, 1.66, 5], False]
>>> lst1∗3
[3, 'four', 3, 1.66, 5, 3, 'four', 3, 1.66, 5, 3, 'four', 3, 1.66, 5]
>>> lst1[0]
3
>>> lst2[−1]
False
>>> lst2[3][1]
'four'
>>> lst2[1:3]
['S', −0.6]
>>> lst2[−2:1:−1]
[[3, 'four', 3, 1.66, 5], −0.6]
```

```
>>> 1.66 in lst2
False
>>> for ch in lst1:
...     print(ch)
...
3
four
3
1.66
5
```

Note that the expression `1.66 in lst2` evaluates to `False` since `lst2` does not contain the floating number 1.66 even though it includes `lst1` which contains this number.

The major difference between Python strings and lists is that lists are mutable whereas strings are immutable. Strings cannot be altered once defined. However, the value of an item in a list can be modified by an assignment statement. Here is an interaction with the Python interpreter illustrating one possible consequence of such property of lists:

```
>>> a = [1, 3, 5]
>>> b = a
>>> b is a
True
>>> a[1] = "omg"
>>> a
[1, 'omg', 5]
>>> b
[1, 'omg', 5]
```

Beware that the assignment statement `b = a` does not make an independent copy of the list `a` since lists are mutable. In fact, both `a` and `b` refer to the same list object after this statement. Thus `b` would be also changed if an element in `a` was changed. To make a copy of a list that does not refer to the same object, you can use the slice operator `:` as follows:

```
>>> a = [2, 4, 6]
>>> b = a[:]
>>> b
```

```
[2, 4, 6]
>>> b is a
False
```

On the other hand, integers are immutable and so the following code does not change the value of `b[2]`:

```
>>> a = 6
>>> b = [2, 4, a]
>>> a = 8
>>> b
[2, 4, 6]
```

The assignment statement `a = 8` creates a new integer object for the variable `a`. The original integer object for `a` stored in the list `b` is not changed by the assignment and so the list `b` has no change.

Table 2.13: Common list methods in Python

Method	**Description**
`append(`*`element`*`)`	Append ***`element`*** to the end of the list.
`extend(`*`list2`*`)`	Append the elements from ***`list2`*** to the end of the list.
`insert(`*`index, element`*`)`	Insert ***`element`*** at the index **`index`** of the list.
`count(`*`element`*`)`	Return the number of elements equal to ***`element`*** in the list.
`index(`*`element`*`)`	Return the lowest index of the list containing ***`element`***.
`remove(`*`element`*`)`	Return the first occurrence of ***`element`*** from the list.
`copy()`	Return a copy of the list.
`pop()`	Remove and return the last element from the list.
`reverse()`	Reverse the list in place.
`sort()`	Sort the list in place.

Just like strings, list objects come with a large number of useful methods in Python which are summarized in Table 2.13. Below is an interactive Python session demonstrating the use of these methods.

```
>>> lst = []
>>> lst.append(3)
>>> lst.extend([6, 9, 12])
>>> lst
[3, 6, 9, 12]
```

```
>>> lst.insert(2, 15)
>>> lst.remove(9)
>>> lst
[3, 6, 15, 12]
>>> lst.index(6)
1
>>> lst.sort()
>>> lst
[3, 6, 12, 15]
>>> lst.reverse()
>>> lst
[15, 12, 6, 3]
```

Note that list objects can grow or shrink without having to copy the contents to a new object since they are mutable.

Unlike Python 2, Python 3 does not allow direct comparisons between objects of different types. So it's an error to sort a list containing a mixture of different types of objects. For example,

```
>>> lst = [123, "Happy", 2.00, False]
>>> lst.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: '<' not supported between instances of 'str' and 'int'
```

If we want to have a sorted copy of a list without changing it, we can use the built-in function **sorted**:

```
>>> lst = [1, 8, 4, 2, 5, 9]
>>> sorted(lst)
[1, 2, 4, 5, 8, 9]
>>> sorted(lst, reverse=True)
[9, 8, 5, 4, 2, 1]
>>> lst
[1, 8, 4, 2, 5, 9]
```

By default, the functions **sort** and **sorted** sort the items in a list in ascending order. To sort the items in descending order, set the optional argument `reverse = True` for these functions.

As mentioned in Section 2.3, we can use the Python built-in function `range` to generate a list composed of a sequence of numbers. Python also provides a useful and powerful compact syntax called list comprehension for creating lists. Its general syntax is

```
<mylist> = [<expression> for <var> in <list> if <condition>]
```

which is equivalent to

```
<mylist> = []
for <var> in <list>:
    if <condition>:
        <mylist>.append(<expression>)
```

Here <*expression*> produces the elements of the list <*mylist*> for each value of the list <*list*> whenever the Boolean expression <*condition*> is `True`. Below is an interaction with the Python interpreter showing how to use this syntax:

```
>>> x = [i*i for i in range(5) if i*i % 2 == 0]
>>> x
[0, 4, 16]
>>> motto = "Have Courage and Be Kind".split()
>>> [i.upper() for i in motto]
['HAVE', 'COURAGE', 'AND', 'BE', 'KIND']
>>> coords = [(x, y) for x in range(3) for y in range(4)]
>>> coords
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2,
0), (2, 1), (2, 2), (2, 3)]
```

Note that list comprehension can be nested as shown in the last example.

Python also provides another sequence data type called `tuple`. Tuples are just like lists except that tuples are immutable. They can be constructed by placing a sequence of comma-separated objects inside round parentheses `()`:

```
>>> tup = (2, "four", 6)
>>> tup
(2, 'four', 6)
```

Tuples can be indexed and sliced in the same way as lists but, being immutable, they cannot be appended to, extended, or have elements removed from them:

```
>>> tup = ("one", 0.123, 101, False)
>>> tup[3]
False
>>> tup[0:2]
('one', 0.123)
>>> tup[0:3:2]
('one', 101)
>>> tup[1] = -0.522
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
>>> tup.append("five")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'tuple' object has no attribute 'append'
```

Although a tuple itself is immutable, it may contain references to mutable objects such as lists. For example,

```
>>> tup = (1, "A", ["x", "y", "z"], 2)
>>> tup[2][1] = "q"
>>> tup
(1, 'A', ['x', 'q', 'z'], 2)
```

An empty tuple is created with empty round parentheses `()`. To create a tuple containing only one item, we should enclose the item in round parentheses with a trailing comma to avoid confusion with other syntactical use of parentheses, e. g. `tup = ("two",)`.

In simple assignments and few other situations, the round parentheses around a tuple's items can be omitted:

```
>>> tup = 3, 2, 1
>>> tup
(3, 2, 1)
```

Such usage is an example of **tuple packing** in which the variables are packed together in a tuple. The reverse is called **tuple unpacking** in which the individual items of a tuple are extracted back into variables. It is a common way of assigning multiple variables in one line:

```
>>> x, y, z = 11, 12, 13
>>> z
13
```

This way of assigning multiple variables is commonly used in preference to separate assignment statements either on different lines or on a single line, separated by semicolons.

Tuples are useful for representing an ordered collection of related data that cannot be altered. In the previous example, the values `11`, `12` and `13` are packed into a tuple for the purpose of assigning them to the variables `x`, `y`, and `z`. But the tuple object itself is destroyed once the assignment was done. As another example, a function may return more than one object and the returned objects are packed into a tuple. Beware that program execution is faster when manipulating a tuple than it is for the equivalent list.

Strings, lists, and tuples are sequences of items which can be taken one at a time. Thus they are all examples of data structures that are **iterable** objects. Python provides the built-in constructor methods **`list`** and **`tuple`** which generates a list and a tuple respectively from any iterable object. For example,

```
>>> list("happy")
['h', 'a', 'p', 'p', 'y']
>>> tuple([2, "four", 6])
(2, 'four', 6)
```

We can also use these methods to convert a list to tuple or vice versa as follows:

```
>>> tup = tuple([6, 4, 2])
>>> tup
(6, 4, 2)
>>> lst = list(tup)
>>> lst
[6, 4, 2]
```

The data elements are *copied* in the construction of a new object using the **`list`** method. So it is another way of creating an independent **`list`** object from another one:

```
>>> x = [1, 3, 5, 7, 9]
>>> y = x
>>> y is x
True
>>> y = list(x)
>>> y is x
False
```

We can use the built-in functions **any** and **all** to test whether any or all of the items in an iterable object is `True`. Here is an interactive Python session demonstrating the use of these functions.

```
>>> x = [1, 0, 2, 0, 3]
>>> any(x), all(x)
(True, False)
>>> y = [False, [], 0.]
>>> any(y), all(y)
(False, False)
```

Sometimes we may want to call a function with arguments taken from a list or other sequence. The $*$ syntax used in a function call unpacks such a sequence into positional arguments to the function. For example, the `math.pow` function takes two arguments `a` and `b` and returns `a` raised to the power `b`. If the arguments we wish to use are in a list or tuple, then the following will fail:

```
>>> import math
>>> lst = [2, 5]
>>> math.pow(lst)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: pow() takes exactly 2 arguments (1 given)
```

It is an error to call this function with a single argument. We should instead index the list explicitly to retrieve the two values we need:

```
>>> math.pow(lst[0], lst[1])
32.0
```

But a more elegant method is to unpack the list into arguments to the function using the $*$ syntax:

```
>>> math.pow(*lst)
32.0
```

Recall that the Python built-in function **range** generates a sequence of numbers. In Python 3, the object created by **range** is not a **list**. Indeed, it is an iterable object that can produce integers on demand. Like other iterable objects, **range** objects can be indexed, cast into lists and tuples, and iterated over:

```
>>> a = range(0, 10, 2)
>>> a[1]
2
>>> list(a)
[0, 2, 4, 6, 8]
>>> for x in range(3):
...     print(x)
...
0
1
2
```

Since **range** objects can be used to produce a sequence of integers, it is tempting to use them to provide the indexes of lists or tuples when iterating over them in a **for** loop. For example,

```
>>> reptiles = ["crocodiles", "lizards", "snakes", "turtles"]
>>> for i in range(len(reptiles)):
...     print(i, ":", reptiles[i])
...
0 : crocodiles
1 : lizards
2 : snakes
3 : turtles
```

It's more natural to avoid the explicit construction of a **range** object which can be accomplished by using the Python built-in function **enumerate**. This function takes an iterable object and produces a tuple `(count, item)` for each `item` in turn where the tuple consists of a counting index and the item itself:

```
>>> reptiles = ["crocodiles", "lizards", "snakes", "turtles"]
>>> for i, reptile in enumerate(reptiles):
...     print(i, ":", reptile)
...
0 : crocodiles
1 : lizards
```

```
2 : snakes
3 : turtles
```

Note that each `(count, item)` tuple is unpacked into the variables `i` and `reptile` in the `for` loop. It is also possible to set the starting value of `count` to something other than 0. For instance, if `reptiles` has the same definition as in the previous example, then

```
>>> list(enumerate(reptiles, 3))
[(3, 'crocodiles'), (4, 'lizards'), (5, 'snakes'), (6, 'turtles')]
```

Python allow us to iterate over two or more sequences at the same time. The built-in function `zip` can be used for such purpose. It creates an iterator object in which each item is a tuple of items taken in turn from the sequences passed to it. For example,

```
>>> x = [1, 2, 3, 4]
>>> y = [1, 4, 9, 16]
>>> zip(x, y)
<zip object at 0x0000027C4F242388>
>>> for pair in zip(x, y):
...     print(pair)
...
(1, 1)
(2, 4)
(3, 9)
(4, 16)
>>> list(zip(x, y))
[(1, 1), (2, 4), (3, 9), (4, 16)]
```

A nice feature of `zip` is that it can be used to *unzip* sequences of tuples as well:

```
>>> z = zip(x, y)
>>> X, Y = zip(*z)
>>> print(X, Y)
(1, 2, 3, 4) (1, 4, 9, 16)
>>> list(X) == x, list(Y) == y
(True, True)
```

Since `zip` does not copy the items into a new object, working on it is memory-efficient and fast. However, it implies that we cannot index a `zip` object and can only iterate over the zipped items:

```
>>> z = zip([1, 2, 3], [2, 4, 6])
>>> z[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'zip' object is not subscriptable
>>> for pair in z:
...     a = 0 # Dummy operation for each iteration
...
>>> list(z)
[]
```

2.6 Formatted Input and Output

String operations are often required to handle the basic input and output tasks of a Python program. We can use basic string operations to build nicely formatted output. However, sometimes you may need to build a complex output through slicing and concatenation of smaller strings which can be tedious. Python supports several approaches to do string formatting that makes the job much easier.

Python supports a powerful string's method **format** for formatting string since version 2.6. In Python 3, this method is preferred over C-style formatting with `%` operator which is less powerful and error prone. Here we will only study its basics usage. You can read the official Python documentation for more details. The idea of the **format** method is that the string serves as a sort of template and the values supplied as arguments are interpolated into this template to form a new string. The general form of this method is:

`<template-string>.format(<values>)`

The `<template-string>` is a string containing one or more **replacement fields** surrounded by curly braces {}. Anything else that is not contained in curly braces will be printed without any changes. To insert the curly braces themselves into a formatted string, they must be doubled up, i. e. use {{ and }}. Each replacement field will be replaced by a formatted value of one of the arguments in `<values>`. The usual syntax of a replacement field is:

`{<index>:<format-spec>}`

The `<index>` indicates which argument is inserted into the replacement field. It can be numbered using a zero-based scheme or named, which helps with longer strings and allows the same value to be interpolated more than once. If it is omitted, the arguments are just filled into the replacement fields in a left-to-right fashion. For example,

```
>>> "{1} plus {0} equals {2}.".format(4, 3, "seven")
'3 plus 4 equals seven.'
>>> "{n2} plus {n3} equals {n1}.".format(n1="seven", n2=3, n3=4)
'3 plus 4 equals seven.'
>>> "{} plus {} equals {}.".format(3, 4, "seven")
'3 plus 4 equals seven.'
>>> "{0} plus {0} equals {1}.".format(3, 3+3)
'3 plus 3 equals 6.'
```

The code after the colon in a replacement field (i. e. <*format−spec*>) is the **format specifier** giving the specification on how the argument should be formatted. A typical format specifier has the form

<*fill*><*align*><*sign*><*width*>.<*precision*><*type*>

Below are the descriptions for the individual options in this form:

1. The <*align*> option specifies the alignment options of the field. Table 2.14 lists the possible alignment options.

   Table 2.14: Alignment options for the format specifier of the `str.format` method

Option	**Description**
`<`	Force the field to be left-aligned to the remaining space.
`^`	Force the field to be center-aligned to the remaining space.
`>`	Force the field to be right-aligned to the remaining space.
`=`	Forces zero padding after the sign (if any) but before the digits. This option is only valid for number types.

2. The <*fill*> option specifies the pad character for filling the field if a valid <*align*> value is specified. It can be any character and defaults to a space if omitted.
3. The <*sign*> option is only valid for number types. It specifies the behavior for displaying the sign of the number and can have one of the values shown in Table 2.15.
4. The <*width*> option specifies the minimum width of the field. If the field takes less space than the minimum width, then the extra space will be filled by the pad character. If the field is too long for the minimum field width, then it will take up as much space as needed to show the field.

Table 2.15: Sign options for the format specifier of the `str.format` method

Option	**Description**
+	A sign is used for both positive and negative numbers.
-	A sign is used only for negative numbers, which is the default behavior.
space	A leading space is used for positive numbers and a minus sign is used for negative numbers.

5. The <***precision***> option specifies the number of decimal places to print for floating point values formatted with `f` and `F`, or the number of significant digits to print for other floating point values. For non-number types, it specifies the maximum field width, i. e. the number of characters will be used from the field content. Note that this option is not allowed for integer values. Moreover, if this option is not specified, the preceding full stop `.` should be removed.

6. The <***type***> option specifies the presentation type, i. e. how the argument should be presented. Table 2.16 shows the commonly used type options.

Table 2.16: Common type options for the format specifier of the `str.format` method

Data Type	**Option**	**Description**
`string`	`s`	String
`int`	`d`	Decimal integer
	`b`	Binary integer
	`o`	Octal integer
	`x`	Hexadecimal integer with lowercase letters after 9
	`X`	Hexadecimal integer with uppercase letters after 9
`float`	`f`	Fixed point notation
	`F`	Same as `f` except switches `inf` and `nan` by `INF` and `NAN`
	`e`	Scientific notation using `e` to indicate the exponent
	`E`	Scientific notation using `E` to indicate the exponent
	`g`	General format which switches to `e` for very small and very large numbers
	`G`	General format which switches to `E` for very small and very large numbers
	`%`	Multiplies by 100 and displays as type `f`, followed by a percent sign

Below is an interaction with the Python interpreter demonstrating the use of the format specifier of the `format` method:

```
>>> "=== {0:<12} ===".format("Physics")
'=== Physics      ==='
>>> "=== {0:^12} ===".format("Physics")
'===   Physics    ==='
>>> "=== {0:*>12} ===".format("Physics")
'=== *****Physics ==='
>>> "=== {0:5} ===".format("Physics")
'=== Physics ==='
>>> a = 2046
>>> "a = {0:5d}".format(a)
'a =  2046'
>>> "a = {0:10b}".format(a)
'a = 11111111110'
>>> "a = {0:5o}".format(a)
'a =  3776'
>>> "a = {0:5x}".format(a)
'a =   7fe'
>>> "{0:>+4d}".format(1234)
'+1234'
>>> "{0:>-4d}".format(1234)
'1234'
>>> "{0:> 4d}".format(1234)
' 1234'
>>> a = -123
>>> b = 486
>>> print("{0:+6d}\n{1:+6d}\n={2:+5d}".format(a, b, a+b))
  -123
  +486
= +363
>>> a = 1.357913e-12
>>> "{:g}".format(a)
'1.35791e-12'
```

```
>>> "{:10.2E}".format(a)
'   1.36E−12'
>>> "{:15.13f}".format(a)
'0.0000000000014'
>>> "{:10f}".format(a)
'  0.000000'
>>> "Percentage gain = {:5.2%}".format(0.125)
'Percentage gain = 12.50%'
```

Python 3.6 introduced a concise and convenient way for formatting strings known as f-strings which are faster than both %-formatting and `str.format`. To create an f-string, prefix the string with the letter `f`. The string itself can be formatted in a similar manner as that for `str.format()`. Below are some simple examples showing how to use f-strings:

```
>>> a = 3
>>> b = 4
>>> f"{a} plus {b} equals {a + b}."
'3 plus 4 equals 7.'
>>> name = "Mary"
>>> mark = 64
>>> f"Hello, {name}! Your test mark is {mark}."
'Hello, Mary! Your test mark is 64.'
>>> value = 123.4567
>>> width = 10
>>> precision = 3
>>> f"value = {value:{width}.{precision}f}"
'value =    123.457'
>>> f"value = {value:{width}.{precision}e}"
'value =  1.235e+02'
>>> a = 4086
>>> f"a = {a:5d}"
'a =  4086'
>>> f"a = {a:5o}"
'a =  7766'
>>> f"a = {a:5x}"
```

```
'a =   ff6'
>>> student = "Peter"
>>> subject = "Physics"
>>> institution = "HKU"
>>> message = (
...     f"Hi {student}! "
...     f"You are studying {subject} in {institution}."
... )
>>> message
'Hi Peter! You are studying Physics in HKU.'
```

From the last example, we can see that an `f` need to be placed in front of each line of a multiline string. As usual, read the official Python documentation if you want to learn more about f-strings.

2.7 User-defined Functions

Python functions are sets of statements grouped into a single block that can be called more than once in a program. They are useful for avoiding code duplication and allow a program to be broken into separate parts where each part performs a specific task. So far we have only used pre-written Python functions such as built-in functions as well as functions and methods from modules and packages. In this section, we discuss how to design our own functions to make our programs more understandable and easier to maintain.

A function is a kind of a **subprogram** — a small program inside a program. The part of the program that defines a function is called a function definition. A function definition has the general form:

```
def <name> (<parm1>, <parm2>, ..., <parmN>):
    <body>
```

Here are the details for the components in the definition:

1. A function header begins with the **`def`** keyword followed by the name of the function and a list of **parameters** enclosed in parentheses and finally ends with a colon. The parameter list may be empty or contain any number of parameters separated from one another by commas. The parameters specifies the necessary information for the function to do its work.

2. The body of a function is made up of one or more Python statements which is written in an indented block following the function header. It is executed each time the function is called. A function body can contain one or more <***return***> statement which stops the execution of a function call and returns the result to the caller. The general syntax of this statement is

   ```
   return <expr1>, <expr2>, ..., <exprN>
   ```

   where <***expr1***>, <***expr2***>, ..., <***exprN***> are expressions. If a <***return***> statement is followed by an expression list, that expression list is evaluated and the value is returned to the caller after the function terminates. If no expression list is specified or the statement itself is not contained inside a function, then the function returns the special value `None`. Python functions can return any type of data such as integer, list, or array and those that return values can be used in expressions. A single function can even return various types of data. Note that if a <***return***> statement is reached during the execution of a function, the current function call is left at that point. Besides, if a single function has multiple <***return***> statements, then execution of the function call ends when one of these return statements is reached.

3. The first statement of a function body should be a **docstring** which is a documentation string of the function. It should be written as a triple-quoted string on a single line if the function is simple, or on multiple lines with an initial one-line summary for more detailed descriptions of complex functions. A docstring should provide details about how to use the function: which arguments to pass it and which objects it returns. Docstrings are also used to provide documentation for classes and modules.

Function definitions can appear anywhere in a Python program. Once a function is defined, it can be called multiple times from any places in a program. Functions can even be **nested**, i. e .a function can be defined inside another function. But a function defined inside another one is not directly accessible from outside that function. Below are some examples of Python function definitions:

```
def happy():
    """Print a birthday greeting."""
    print("Happy Birthday to you.")


def bdsong(person):
    """Print a birthday song for person."""
    happy()
```

```
    happy()
    print("Happy birthday to", person + "!")
    happy()

from math import sqrt
def roots(a, b, c):
    """Return the roots of ax^2 + bx + c."""
    d = b**2 - 4*a*c
    r1 = (-b + sqrt(d))/(2*a)
    r2 = (-b - sqrt(d))/(2*a)
    return r1, r2
```

A function is called by specifying its name and a list of values called **arguments** as follows:

```
<name> (<arg1>, <arg2>, ..., <argN>)
```

When Python comes to a function call, the calling program suspends execution at the point of the call. Then the arguments are assigned to the parameters of the function and the body of the function is executed. Finally, control returns to the point just after where the function was called. If the function returns any values, the returned value is used as the expression result. Below is an interactive Python session showing how to call the Python functions defined in the previous examples:

```
>>> happy()
Happy birthday to you!
>>> bdsong("Peter")
Happy birthday to you!
Happy birthday to you!
Happy birthday to Peter!
Happy birthday to you!
>>> print(roots(1., -3., -10.))
(5.0, -2.0)
>>> print(roots(1., -10., 24.))
(6.0, 4.0)
```

Notice that the memory occupied by the variables defined in a function is released when a call to the function finishes. So these variables do not retain any values from one function call to the next.

Since Python functions are objects, we can assign identifiers to them. They can be also passed as arguments to other functions and can even be returned from other functions. In addition, Python allows more than one identifiers to be assigned to the same function object. For example,

```
>>> from math import cos, pi
>>> def secant(x):
...     """Return the secant of x, secant(x) = 1/cos(x)."""
...     return 1./cos(x)
...
>>> secant
<function secant at 0x0000020B6E5F8EA0>
>>> secant(math.pi/3)
1.9999999999999996
>>> sec = secant
>>> sec
<function secant at 0x0000020B6E5F8EA0>
>>> sec(math.pi/3)
1.9999999999999996
```

We have considered the case in which arguments are passed to the function in the order in which they appear in the function definition. These arguments are known as **positional arguments**. It's also possible to pass the arguments in an arbitrary order by setting them explicitly as **keyword arguments** as follows:

```
>>> roots(a=1., c=-10., b=-3.)
(5.0, -2.0)
>>> roots(b=-3., a=1., c=-10.)
(5.0, -2.0)
```

If you mix positional and keyword arguments, the former must come first; otherwise Python won't know the positional argument corresponds to which variable. For instance,

```
>>> roots(1., c=-10., b=-3.)
(5.0, -2.0)
>>> roots(b=-3., 1., -10.)
  File "<stdin>", line 1
SyntaxError: positional argument follows keyword argument
```

Sometimes you may want to define a function that takes a **default argument**: That is to say, if the caller doesn't provide a value for this argument, a default value would be used. Here is an example demonstrating how to set default arguments in a Python function definition:

```
>>> def report_weight(value, units="kg"):
...     return "The weight is {:.2f} {}".format(value, units)
...
>>> report_weight(43.176, "lb")
'The weight is 43.18 lb'
>>> report_weight(21.5)
'The weight is 21.50 kg'
```

Default arguments are assigned *when the Python interpreter first encounters the function definition*. This can lead to some unexpected results, especially for mutable arguments. For example,

```
>>> def func(alist=[]):
...     alist.append(3)
...     return alist
...
>>> func()
[3]
>>> func()
[3, 3]
>>> func()
[3, 3, 3]
```

The default argument to the function `func` is an empty list. But it is the specific empty list assigned when the function is defined. So this specific list grows each time `func` is called. On the other hand, if a function is defined with an argument defaulting to the value of a variable, any subsequent change to that variable won't change the default. Below is an interaction with the Python interpreter illustrating this property:

```
>>> def_units = "kg"
>>> def report_weight(value, units=def_units):
...     return "The weight is {:.2f} {}".format(value, units)
...
>>> report_weight(35.8)
```

```
'The weight is 35.80 kg'
>>> def_units = "lb"
>>> report_weight(35.8)
'The weight is 35.80 kg'
```

Next we consider the **scope** of variables in Python function definitions. We have learned that namespaces can exist independently from each other and that they can contain same variable names on different hierarchy levels. A **scope** defines the hierarchy level in which Python searches namespaces for certain "name-to-object" mappings. It is a context in which names are stored and in which they can be looked up. Parameters and other variables defined in a function are **local** to that function, i. e. they are not accessible outside the function, even if they have the same names as other variables defined elsewhere in the program. Conversely, variables assigned outside all Python function definitions in a program are **global** and are accessible everywhere within the program. As an example, consider the following Python function:

```
>>> def func():
...     x = 2
...     print(x, y)
...
>>> y = 5
>>> func()
2 5
```

This function prints out two variables `x` and `y` but it only defines `x`. Since `y` isn't defined in the local scope of the function, Python looks into the global scope and finds `y = 5`, so that is what is printed. Python allows `y` to be undefined when the function is defined. But of course it must be defined before the function is called.

What happens if a function defines a variable with the same name as a global variable? In such case, the local variables of the function are searched first when look up the object associated with the variable name. Thus it is the object referred by the local variable name that is retrieved. For example,

```
>>> def func():
...     x = 3
...     print(x)
...
```

```
>>> x = 4
>>> func()
3
>>> print(x)
4
```

Beware that the local variable **x** exists only within the body of the function. It just happens to have the same name as the global variable **x**. It disappears after the function exits and doesn't overwrite the global **x**.

Python uses the **LEGB rule** to determine the order in which namespaces are to be searched for the "name-to-object" mapping of a variable. According to this rule, the scopes listed below are searched in the hierarchical order from the top to the bottom:

1. Local scope (L) refers to the names defined inside the current function.
2. Enclosing scope (E) refers to the names defined inside any and all enclosing functions, from inner to outer.
3. Global scope (G) refers to the names defined at the top-level of a module file. Each module brings with it a new global scope.
4. Built-in scope (B) refers to the predefined names in the built-in modules.

Notice that if a variable is given the same name as a built-in function, then that name refers to your variable either in local or global scope and not to the original built-in. So it is in general not a good idea to name your variables after built-in functions.

Python does not allow users to modify the variables defined in scopes other than the local scope. Below is an interaction with the Python interpreter demonstrating this restriction:

```
>>> def func1():
...     print(x)
...
>>> def func2():
...     x -= 2
...     print(x)
...
>>> x = 8
>>> func1()
```

```
8
>>> func2()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in func2
UnboundLocalError: local variable 'x' referenced before assignment
```

If you really want to change the variables defined outside the local scope, you must first tell Python to use these variables instead of the locally defined variables using the keywords **`global`** (for variables in global scope) and **`nonlocal`** (for variables in enclosing scope). For instance, we can modify the function **`func2`** in the previous example as follows:

```
>>> def func2():
...     global x
...     x -= 2
...     print(x)
...
>>> x = 8
>>> func2()
6
```

In practice, it is usually a bad idea to modify the variables defined outside the function scope since it's often the cause of confusing code and tricky errors that are hard to debug.

In many computer languages, it's important to distinguish whether arguments are **passed by reference** or **passed by value** to functions. If an argument is passed by reference, then the variable holding the argument is sent as a parameter to the function. Thus the function can change the value of the argument in the calling program by assigning a new value to the parameter. On the other hand, if an argument is passed by value, then the parameter of the function receives a copy of the value of the argument. Thus the function cannot change the value of the argument in the calling program. As a result, making changes to the function parameters will have a different effect on the calling program for the passed by reference and passed by value schemes.

However, the distinction of whether arguments are passed to functions by value or by reference does not fit well into the Python "name-to-object" model. Recall that everything in Python is an object, and the same object may have multiple identifiers. When an identifier is passed to a function, the value being passed is in fact the reference to the object that it points to. Whether the function can change the object or not in the caller depends on the object is mutable or immutable. Let's study some examples to help us understand this point.

We first consider the Python function `func1` that takes an integer argument:

```
>>> def func1(a):
...     print("func1: a = {}, id = {}".format(a, id(a)))
...     a = 7
...     print("func1: a = {}, id = {}".format(a, id(a)))
...
>>> a = 3
>>> print("global: a = {}, id = {}".format(a, id(a)))
global: a = 3, id = 140727007765392
>>> func1(a)
func1: a = 3, id = 140727007765392
func1: a = 7, id = 140727007765520
>>> print("global: a = {}, id = {}".format(a, id(a)))
global: a = 3, id = 140727007765392
```

This function receives a reference to that integer object for which it attaches a local name `a` (which may or may not be the same as the global name). It cannot change the integer object which is immutable. So any reassignment of the local name `a` simply points to a new object and the global name `a` still points to the original integer object.

Then we consider the Python function `func2` that takes a list argument (which is a mutable object):

```
>>> def func2(a):
...     print("func2: a = {}, id = {}".format(a, id(a)))
...     a.append(8)
...     print("func2: a = {}, id = {}".format(a, id(a)))
...
>>> b = [2, 4, 6]
>>> print("global: b = {}, id = {}".format(b, id(b)))
global: b = [2, 4, 6], id = 1808961829384
>>> func2(b)
func2: a = [2, 4, 6], id = 1808961829384
func2: a = [2, 4, 6, 8], id = 1808961829384
```

```
>>> print("global: b = {}, id = {}".format(b, id(b)))
global: b = [2, 4, 6, 8], id = 1808961829384
```

We can observe that a reassignment to the list in this function changes the original object and such change persists after the function call. Note that it doesn't matter what name is given to the list by the function: this name points to the same object, as you can see from its **id**.

So are Python arguments passed by value or by reference? The best answer is probably that arguments are passed by value, but that value is a reference to an object (which can be mutable or immutable). Moreover, if the argument is a variable whose value is a mutable object (like a list or graphics object), then any changes to the state of the object inside the function will be visible to the calling program.

Finally, we consider **recursive functions** which are functions that can call themselves. Recursion is not always necessary but can lead to elegant algorithms in some situations. For example, one way to calculate the factorial of an integer $n \geq 1$ is to define the following recursive function:

```
>>> def factorial(n):
...     if n == 1:
...         return 1
...     return n*factorial(n - 1)
...
>>> factorial(6)
720
```

This function implements a recursive algorithm which makes use of the fact that $n! = n \cdot (n-1)!$. As a result, a call to `factorial(n)` returns `n` times the returned value of `factorial(n-1)`, which returns `n - 1` times the returned value of `factorial(n - 2)`, and so on until `factorial(1)` which is 1 by definition. Whenever such recursive algorithms are implemented, we must make sure that they stop under certain conditions.

2.8 File Processing

Up to now, we have only studied Python programs in which input data is hard-coded or from the user while output data is directed to the terminal. In practice, we often need to input data from an external file and write data to an output file. Python provides **file** objects for such purpose.

In Python, a **`file`** object is created by opening a file which is done using the **`open`** function. A common syntax for assigning a file object to a variable is

```
<variable> = open(<name>, <mode>)
```

where <*`name`*> is a string that contains the filename and <*`mode`*> is a string that specifies the mode in which the file is opened. The filename may be given as an absolute path or as a path relative to the directory in which the program is being executed. In addition, the file mode argument is optional and the default mode is `r` — reading in text mode. Table 2.17 shows a list of common file modes in Python. Files are normally opened in text mode in Python, i. e. you read and write strings from and to the file. However, the file is opened in binary mode if `b` is appended to the file mode such as `rb` and `w+b`. In such case, the data is read and written in the form of bytes objects. We should use the binary mode for all files that do not contain any text.

Table 2.17: Common file modes in Python

Mode	Description
`r`	Open a file for reading only.
`w`	Open a file for writing only. If the file does not exist, a new file is created. Otherwise the file is truncated.
`a`	Open a file for appending at the end of the file. If the file does not exist, a new file is created.
`r+`	Open a file for both reading and writing.
`w+`	Open a file for both reading and writing. If the file does not exist, a new file is created. Otherwise the file is truncated.
`a+`	Open a file for both reading and appending. If the file does not exist, a new file is created.

As an example, we can open a file for writing in text mode using a Python statement like the following:

```
>>> outfile = open ("myfile.txt", "w")
```

Once a file has been opened, its contents can be accessed through the associated **`file`** object. We can also obtain various file information from the data attributes of the **`file`** object. For instance, if `outfile` is the **`file`** object defined in the previous example,

```
>>> print("Name of the file: {}".format(outfile.name))
Name of the file: myfile.txt
>>> print("Opening mode: {}".format(outfile.mode))
Opening mode: w
```

```
>>> print("Closed or not: {}".format(outfile.closed))
Closed or not: False
```

When we are finished with the operations of a file, the file should be closed. Closing a file makes sure that any bookkeeping that was necessary to maintain the correspondence between the file on disk and the associated `file` object is finished up. It will also free up the resources that were tied with the file. To close a file, we use the `close` method of the `file` object as follows:

```
>>> outfile.close()
```

Python closes any open file objects automatically when a program terminates. However, we should not rely on it to close a file.

Python `file` object provides three methods for reading information from a file:

1. <*file*>`.read()` returns the entire remaining contents of the file as a single (potentially large, multi-line) string while <*file*>`.read(n)` returns `n` bytes from the file.
2. <*file*>`.readline()` returns the next line of the file, i. e. all text up to and including the next newline character.
3. <*file*>`.readlines()` returns a list of the remaining lines in the file where each list element is a single line including the newline character at the end.

Here is an example Python program that prints the entire contents of a file to the screen using the `read` method:

```
# printfile.py
# This program prints a file to the screen.
# Last Update on 5 Jan 2022 by F K Chow

fname = input("Enter filename: ")
infile = open(fname, "r")
data = infile.read()
print(data)
infile.close()
```

In this program, the entire contents of the file is read as one large string and stored in the variable `data`. Printing `data` causes the contents to be displayed.

Note that successive calls to **readline** get successive lines from the file which is analogous to **input**. However, unlike **input** which discards the newline character, the string returned by **readline** will always end with a newline character. For instance, we can use the **readline** method to print out the first three lines of a file:

```
>>> infile = open("trial.txt", "r")
>>> for i in range(3):
...     line = infile.readline()
...     print(line[:-1])
...
1st line of the file
2nd line of the file
3rd line of the file
>>>
```

Here slicing is used to strip off the newline character at the end of the line. Since **print** automatically jumps to the next line after it outputs a newline character, printing with the explicit newline character at the end would put an extra blank line between the lines of the file. Alternatively, you can suppress these blank lines by telling **print** not to add its own newline character using the statement `print(line, end = "")` .

Sometimes, we may want to store each element on every line of a file as an element in a two-dimensional list. We can use the **readlines** method for such purpose as shown in the following code:

```
>>> infile = open("trial.txt", "r")
>>> mydata = []
>>> for line in infile.readlines():
...     row = []
...     for i in line.strip("\n").split(" "):
...         row.append(i)
...     mydata.append(row)
...
>>> print(mydata)
[['1st', 'line', 'of', 'the', 'file'], ['2nd', 'line', 'of', 'the',
'file'], ['3rd', 'line', 'of', 'the', 'file']]
```

For each line, the newline character is stripped using the string method **strip** and the elements are split into a list with spaces as the separation character using the method `split`. This approach is both simple and general, and can be used for any kind of file.

Besides, the `readlines` method can be also used to loop through the entire contents of a file as follows:

```
>>> infile = open("trial.txt", "r")
>>> for line in infile.readlines():
...     print(line, end="")
...
1st line of the file
2nd line of the file
3rd line of the file
>>>
```

In this example, the lines of the file are stored as individual items of a list and then the list is looped through using a **for** loop. However, this approach has a potential drawback that reading the file into a list all at once may take up too much memory if the file is very large. Fortunately, there is an alternative way for looping over a file. Python **file** objects are iterable objects and the file itself is regarded as a sequence of lines. So looping through the entire contents of a file can be done directly like this:

```
>>> infile = open("trial.txt", "r")
>>> for line in infile:
...     print(line, end="")
...
1st line of the file
2nd line of the file
3rd line of the file
>>>
```

Unless you really want to store every line in memory, you should use this approach as it is memory efficient, fast, and yields simpler code.

Python **file** object also provides the **write** method to write information into a file. This method has the form:

```
<file>.write(<string>)
```

where <*string*> is the string be written to the file. You can replace this string by a formatted string in the form discussed in section 2.7 if you want to output formatted data to the file. For example, below is a Python program that writes the first three powers of the integers between 1 and 100 to a file using the `write` method:

```
# printpowers1.py
# This program writes the first three powers of the integers between
# 1 and 100 in comma—separated fields to the file powers.txt using
# the write method of a file object.
# Last Update on 19 Nov 2020 by F K Chow

outfile = open("powers.txt", "w")
for i in range(1, 101):
    outfile.write("{}, {}, {}\n".format(i, i**2, i**3))
outfile.close()
```

The contents of the generated file are:

```
1, 1, 1
2, 4, 8
3, 9, 27
⋮
99, 9801, 970299
100, 10000, 1000000
```

Indeed, an easier way to write information into a file is using the Python built-in function **`print`**. To print into a file, we just need to add an extra keyword argument **`file`** to specify the file:

```
print(..., file=<file>)
```

It behaves just like a normal **`print`** except that the result is sent to the file associated with the **`file`** object <*`file`*> instead of being displayed on the screen. As an example, here is a Python program that uses the **`print`** function to generate the same file as `printPowers1.py`:

```
# printpowers2.py
# This program writes the first three powers of the integers between
# 1 and 100 in comma—separated fields to the file powers.txt using
# the built—in function print.
# Last Update on 19 Nov 2020 by F K Chow
```

```
outfile = open("powers.txt", "w")
for i in range(1, 101):
    print("{}, {}, {}".format(i, i**2, i**3), file=outfile)
outfile.close()
```

In a complicated program that reads information from numerous files, the files may not be closed "quickly enough" even with the `close` statements and then leads to a "too many files open" error from the operating system. In such case, we should instead use the `with` statement to access the files using the syntax:

```
with open(<name>, <mode>) as <file>:
    <body>
```

where *<file>* is the variable assigned to the `file` object and *<body>* is a statement block accessing the data of the file. For example,

```
>>> with open("trial.txt", "r") as infile:
...     for line in infile:
...         print(line, end="")
...
1st line of the file
2nd line of the file
3rd line of the file
>>>
```

Within the `with` block, the file is accessed and finally the file is closed immediately on exiting the block. So there is no need to explicitly close the file. Moreover, the file will be closed after the `with` block even if something goes wrong in this block. Using a `with` block is now considered the preferred method of accessing a file.

2.9 Arrays

As we have seen, a Python list is an ordered sequence of values which can be used to perform numerical calculations. However, a `list` object is generally not the most con venient data type for such work. What we really want is an **numerical array** — a grid of numbers which are all of the same type. The NumPy package provides an array class that suits our needs. NumPy arrays are similar to Python lists. But there are some important differences between them:

1. The number of elements in an array is fixed and it must be known when the array is created. We cannot add/remove elements to/from an array once it is created.
2. Each element in an NumPy array occupies a fixed amount of memory and must have the same data type such as **`int`** or **`float`**. We cannot mix elements of different types in the same array or change the type of the elements of an array once it is created.
3. Arrays can be two-dimensional, like matrices in algebra. That is to say, we can have a grid of elements rather then just a one-dimensional row of them. In principle, arrays can have any number of dimensions. By contrast, lists are always just one-dimensional.
4. Arrays behave roughly like vectors or matrices. Arithmetic operations are performed *elementwise* and a new array is created to store the result. This is not true with lists. If we attempt to perform arithmetic with a list, we will either get an error message or an unexpected result.
5. With NumPy, a wide range of mathematical operations can be done directly on the entire arrays, thereby removing the need for loops over array elements. This is commonly called **vectorization** and may cause a dramatic speed-up of Python programs.

All the functions of NumPy are found in the package `numpy`. To use it, it is strongly recommended to import the package with

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

and then refer to its attributes with the prefix `np`. (Here we assume that `numpy` has been imported as `np` whenever it is "directly" used.) The use of qualified name (e. g. `np.array`) helps to avoid collisions of the definitions in the current namespaces and also helps to make clear where the definitions are coming from. The NumPy array class is called `ndarray`, which represents a multidimensional table of elements indexed by a tuple of integers. The dimensions of an array are called **axes** and the number of axes of an array is called its **rank**. Note that an array is printed out in a similar way to a nested list but with the elements separated by spaces instead of commas.

2.9.1 Creation of Arrays

NumPy provides many functions for creating NumPy arrays in different ways. For instance, there are several NumPy methods that can be used to declare a one-dimensional array filled with default or arbitrary values. The function `empty` creates an array without initializing its elements (they are usually near-zero random numbers as they are not initialized) whereas the functions `zeros` and `ones` create an array with all its elements equal to zero or one. Here are some examples for creating arrays using these methods:

```
>>> a = np.zeros(5)
>>> a
array([0., 0., 0., 0., 0.])
>>> print(a)
[0. 0. 0. 0. 0.]
>>> b = np.ones(6)
>>> print(b)
[1. 1. 1. 1. 1. 1.]
>>> c = np.empty(3)
>>> print(c)
[5.73040903e-272 1.75528625e-276 1.91709371e-151]
```

These functions require one argument which is an integer specifying the shape of the array. We can also use these functions to create two-dimensional arrays:

```
>>> a = np.zeros((2, 3))
>>> print(a)
[[0. 0. 0.]
 [0. 0. 0.]]
>>> b = np.ones((3, 5))
>>> print(b)
[[1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]
 [1. 1. 1. 1. 1.]]
>>> c = np.empty((2, 2))
>>> print(c)
[[0.00000000e+000 4.73383003e-313]
 [7.57412805e-312 5.88981152e-310]]
```

In such case, the argument of these functions is a tuple of integers specifying the shape of the array where its first element is the number of rows and its second element is the number of columns. We can create higher-dimensional arrays using these functions in a similar manner.

NumPy also provides other functions to generate some common special arrays. For example, the `eye` function creates a two-dimensional array with ones on the diagonal and zeros elsewhere:

```
>>> a = np.eye(3)
>>> print(a)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
>>> b = np.eye(2, 5)
>>> print(b)
[[1. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0.]]
```

The first and second arguments of this function specify the number of rows and columns of the array. If the second argument is omitted, then the number of columns is assumed to be equal to the number of rows. Another example is the `random.random` function which creates a two-dimensional array filled with random numbers in the half-open interval $[0.0, 1.0)$:

```
>>> a = np.random.random(3)
>>> print(a)
[0.42920573 0.28488559 0.32049076]
>>> b = np.random.random((3, 2))
>>> print(b)
[[0.40410184 0.8812502 ]
 [0.24940026 0.80784184]
 [0.40345588 0.7208537 ]]
```

This functions take on one argument which is an integer or a tuple of integers specifying the array's shape.

By default, the NumPy's functions `zeros`, `ones`, `eye`, and `random.random` will create an array filled with floating point values. But you can specify them to be other data types using the optional `dtype` argument (see section 2.9.2 for more details). For example,

```
>>> np.zeros(4, dtype=complex)
array([0.+0.j, 0.+0.j, 0.+0.j, 0.+0.j])
>>> np.eye(4, dtype=int)
array([[1, 0, 0, 0],
       [0, 1, 0, 0],
       [0, 0, 1, 0],
       [0, 0, 0, 1]])
```

Another way to create a NumPy array is to take a list or tuple of values and convert it to an array using the function `array`:

```
>>> a = np.array((11, 12, 13, 14))
>>> print(a)
[11 12 13 14]
>>> b = np.array([[1, 2, 3], [4, 5, 6]])
>>> print(b)
[[1 2 3]
 [4 5 6]]
```

Note that passing a list of lists creates a two-dimensional array (and similarly for higher dimensions).

The data type of the elements of an array created by the function `array` is deduced from the type of the elements in the sequence and "upcast" to the most general type if they are of mixed but compatible types. That is to say, if the sequence contains a mixture of integers and floating point numbers, then all the entries will be converted to floating point numbers. Similarly, if the sequence contains a mixture of floating point numbers and complex numbers, then all the entries will be converted to complex numbers. Below is an interaction with the Python interpreter illustrating such property:

```
>>> np.array([1, 3., 5])
array([1., 3., 5.])
>>> np.array([2, 4 + 4j, 6])
array([2.+0.j, 4.+4.j, 6.+0.j])
```

Sometimes we may want to create an array composing of a sequence of numbers. We can use the NumPy functions `arange` and `linspace` to do so. The `arange` function is the NumPy equivalent of `range` except that it can generate floating point sequences. It also allocates the memory for the elements in an array instead of returning a generator-like object. For example,

```
>>> np.arange(1, 6)
array([1, 2, 3, 4, 5])
>>> np.arange(0, 1.0, 0.2)
array([0. , 0.2, 0.4, 0.6, 0.8])
```

However, `arange` has the drawback that it's not always possible to determine the number of elements will be created due to the finite precision of floating point arithmetic. It is better for us to use the `linspace` function that takes an argument specifying the number of elements to return instead of the desired step size. Unlike `range` and `arange`, the `linspace` function fills the array to include both end points in the sequence. To omit the final point in the sequence, we should set the optional argument `endpoint` to `False`. Here is an interactive Python session demonstrating how to use the `linspace` function:

```
>>> np.linspace(0, 10, 5)
array([ 0. ,  2.5,  5. ,  7.5, 10. ])
>>> np.linspace(0, 10, 5, endpoint=False)
array([0., 2., 4., 6., 8.])
```

As shown in the last example, the array generated by `linspace` has the data type of floating point numbers even if the sequence generates integers. Notice that the functions `array`, `arange`, and `linspace` also take the optional `dtype` argument.

On the other hand, we can use the NumPy function `fromfunction` to create an array initialized with values calculated by a function. The `fromfunction` function takes another function and a tuple representing the shape of the desired array as its arguments. The function in its argument should take the same number of arguments as the rank of the array where these arguments index each element at which the function returns a value. To illustrate, consider the following interactive Python session:

```
>>> def func(i, j):
...     return i**2 + j**2
...
>>> np.fromfunction(func, (2, 3))
array([[0., 1., 4.],
       [1., 2., 5.]])
>>> np.fromfunction(func, (5, 3))
array([[ 0.,  1.,  4.],
       [ 1.,  2.,  5.],
       [ 4.,  5.,  8.],
       [ 9., 10., 13.],
       [16., 17., 20.]])
```

The function `func` is called for every index in the specified shape and the values it returns are used to initialize the corresponding elements.

2.9.2 Basic Data Attributes of Arrays

Since a NumPy array is a `ndarray` object, it has many data attributes which can be used to determine the properties of the array. The basic `ndarray` data attributes are summarized in Table 2.18. Below is an interactive Python session illustrating how to find the data attributes of a NumPy array:

```
>>> a = np.array([[8, 9, 0], [5, 6, 7], [1, 2, 3]])
>>> a.data
<memory at 0x000002DB1716A630>
>>> a.dtype
dtype('int32')
>>> a.itemsize
4
>>> a.ndim
2
>>> a.shape
(3, 3)
>>> a.size
9
```

Table 2.18: Basic `ndarray` data attributes

Attribute	Description
`data`	"Buffer" in memory containing the actual elements of the array
`dtype`	Data type of the elements of the array
`itemsize`	Size in bytes of each element of the array
`ndim`	Number of axes (dimensions) of the array where `ndim` = `len(shape)`
`shape`	The size of the array along each of its axes, returned as a tuple of integers
`size`	The total number of elements in the array, equal to the product of the elements of shape

Note that the `dtype` data attribute of a `ndarray` object is a data type (`dtype`) object which gives us the information about how the elements are represented in the array such as the type and size of the data. To describe the type of scalar data, there is a number of scalar types defined in NumPy. A full list of these types is given in NumPy

Table 2.19: Common scalar types in NumPy

Scalar Type	**Description**
`int_`	Default integer type (same as C's `long`, platform-dependent)
`int8`	Integer in a single byte (-128 to 127)
`int16`	Integer in 2 bytes (-32768 to 32767)
`int32`	Integer in 4 bytes (-2147483648 to 2147483647)
`int64`	Integer in 8 bytes (-2^{63} to $2^{63}-1$)
`uint8`	Unsigned integer in a single byte (0 to 255)
`uint16`	Unsigned integer in 2 bytes (0 to 65535)
`uint32`	Unsigned integer in 4 bytes (0 to 4294967295)
`uint64`	Unsigned integer in 8 bytes (0 to $2^{64}-1$)
`float_`	Default floating point number type (shorthand for `float64`)
`float32`	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
`float64`	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
`complex_`	Default complex number type (shorthand for `complex128`)
`complex64`	Complex number represented by two 32-bit floats (real and imaginary components)
`complex128`	Complex number represented by two 64-bit floats (real and imaginary components)
`bool_`	Default boolean type stored as a single byte

documentation, but the most common ones are listed in Table 2.19. They are not `dtype` objects although they can be used in place of one whenever a data type specification is needed in Numpy. If `numpy` has been imported as `np` as usual, then all scalar types can be referred with the prefix `np` such as `np.int32`. The scalar types that get created by default when using the native Python numerical types are those with a trailing underscore: `np.int_`, `np.float_`, `np.complex_`, and `np.bool_`.

2.9.3 Indexing and Slicing Arrays

We can access rows, columns, and individual elements in a NumPy array in the same way as Python lists using indexing and slicing. In a two-dimensional NumPy array, the rows are indicated as **axis 0** and the columns are indicated as **axis 1**. To uniquely specify an element in the array, we can provide a tuple of two integers where the first one specifies the index along axis 0 and the second one specifies the index along axis 1 using a zero-based indexing scheme. In general, a N-dimensional NumPy array have N axes from axis 0 to axis $N-1$. We can specify an element in the array by providing a tuple of N integers which are the indexes for individual axes. Below is an interactive Python session illustrating how to index and slice NumPy arrays:

```
>>> a = np.array([1, 2, 3, 4, 5, 6])
>>> a[2]
3
>>> a[3:]
array([4, 5, 6])
>>> a[1:5:2]
array([2, 4])
>>> b = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> b
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
>>> b[0]
array([1, 2, 3])
>>> b[1, 2]
6
>>> b[:, 1]
array([2, 5, 8])
>>> b[1, ::-1]
array([6, 5, 4])
>>> c = np.array([[[1, 2, 3], [4, 5, 6]]])
>>> c.shape
(1, 2, 3)
>>> c = np.array([[[1, 2, 3]], [[4, 5, 6]]])
>>> c[1, 0, 2]
6
>>> c.shape
(2, 1, 3)
```

Observe that replacing the index of an array axis by a slice operator `:` results in the selection of every item of the array along that axis.

Just like Python lists, negative indices are valid for NumPy arrays. A negative index `i` is interpreted as `n + i` where `n` is the number of elements in the corresponding dimension. For instance, if `a` and `b` are the arrays defined in the previous example, then

```
>>> a[:-2]
array([1, 2, 3, 4])
>>> b[-1,:]
array([7, 8, 9])
```

Further examples of NumPy array slicing are illustrated in Figure 2.3.

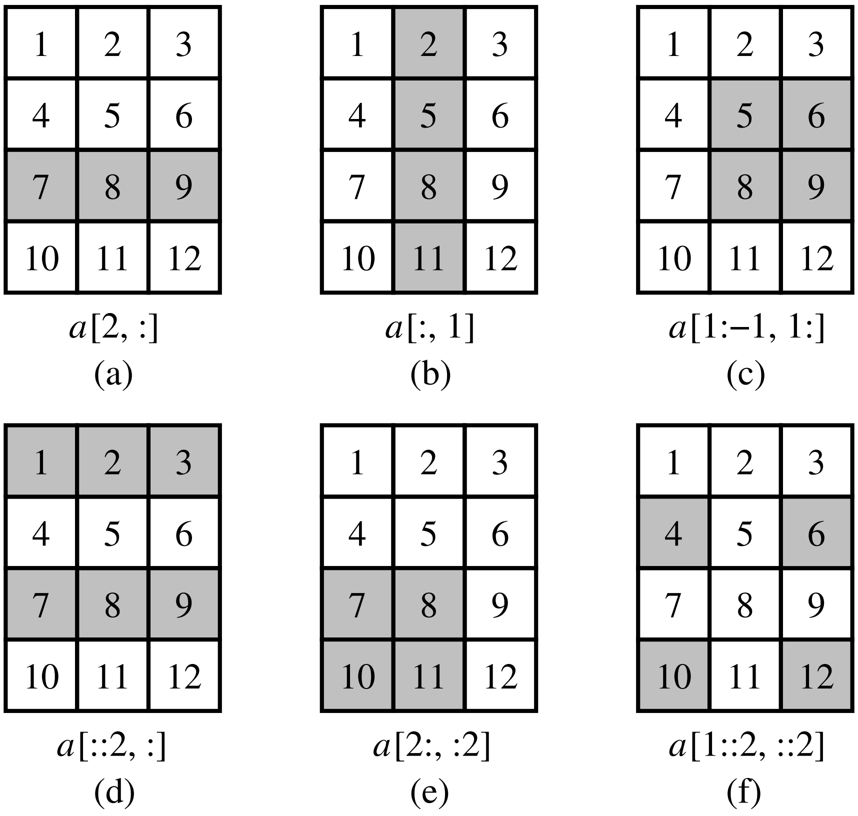

Figure 2.3: Various ways to slice a NumPy array.

If we supply only one index to a multidimensional NumPy array in indexing or slicing, no error will be raised. It is because NumPy will automatically assume a slice operator `:` for any subsequent dimensions. On the other hand, ellipsis `(...)` can also be used along with slicing to make a selection tuple of the same length as the dimension of a NumPy array. For example,

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

So far we have only discussed the rules for **basic slicing** of arrays, i. e. indexing and slicing arrays using simple tuples of integers. All arrays produced by basic slicing are **views** rather than copies of the original array. A view of a NumPy array is a new array which does not "own" its data elements but "shares" the elements of another array. Thus a reassignment of an element of one array affects the other. To give you an idea, consider the following interaction with the Python interpreter:

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

NumPy arrays can also be indexed by sequences that aren't simple tuples of integers, including other lists, arrays of integers and tuples of tuples. Such approach of indexing is called **advanced indexing**. All arrays produced by advanced indexing are new arrays with their own copies of the data rather than views of the original array. As an illustration, consider the indexing of a one-dimensional array by using a list or another array:

```
>>> a = np.linspace(0, 0.5, 6)
>>> i = [1, 4, 5]
>>> b = a[i]
>>> b[1] = 0.8
>>> print(a)
```

```
[0.  0.1 0.2 0.3 0.4 0.5]
>>> print(b)
[0.1 0.8 0.5]
>>> j = np.array([[1, 2], [3, 4], [3, 1]])
>>> c = a[j]
>>> print(c)
[[0.1 0.2]
 [0.3 0.4]
 [0.3 0.1]]
>>> c[1, 0] = 0.9
>>> print(a)
[0.  0.1 0.2 0.3 0.4 0.5]
>>> print(c)
[[0.1 0.2]
 [0.9 0.4]
 [0.3 0.1]]
```

One can even index a multidimensional array with multidimensional arrays of indexes, picking off individual elements at will to build an array of a specified shape. Below is an example of indexing a two-dimensional array in this way:

```
>>> a = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]])
>>> i = np.array([[1, 0], [2, 1]])
>>> j = np.array([[0, 1], [3, 2]])
>>> print(a[i, j])
[[ 5  2]
 [12  7]]
```

Here we make a 2×2 array (the shape of the index arrays) whose elements are `a[1, 0]`, `a[0, 1]` on the top row and `a[2, 3]`, `a[1, 2]` on the bottom row.

Instead of indexing an array with a sequence of integers, it is also possible to use an array of boolean values for such purpose. The `True` elements of this indexing array identify those elements in the target array to be returned:

```
>>> a = np.array([-3, -2, -1, 0, 1, 2])
>>> i = np.array([True, True, False, True, False, False])
>>> print(a[i])
[-3 -2  0]
```

2.9.4 Copying and Sorting Arrays

Suppose `a` is a NumPy array. Just like copying lists, the statement `b = a` makes `a` and `b` point to the same array object since NumPy arrays are mutable. Changing `b` will then also affect `a`:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = a
>>> id(a)
3140008848800
>>> id(b)
3140008848800
>>> b[0, 1] = -2
>>> a
array([[ 1, -2],
       [ 3,  4]])
>>> b
array([[ 1, -2],
       [ 3,  4]])
```

To make an independent copy of a NumPy array, we can use the `ndarray` class's method `copy` or NumPy function `array`. For example,

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = a.copy()
>>> b[0, 1] = -1
>>> a
array([[1, 2],
       [3, 4]])
>>> b
array([[ 1, -1],
       [ 3,  4]])
>>> c = np.array(a)
>>> c[0, 1] = -1
>>> a
array([[1, 2],
       [3, 4]])
```

```
>>> c
array([[ 1, -1],
       [ 3,  4]])
```

Notice that these methods can be also used to copy the data in an array as follows:

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

On the other hand, we can use the `ndarray` class's method `view` to create a view of an array:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> b = a.view()
>>> id(a)
3140008848800
>>> id(b)
3140010177760
>>> b[0, 1] = -2
>>> a
array([[ 1, -2],
       [ 3,  4]])
>>> b
array([[ 1, -2],
       [ 3,  4]])
```

Beware that changing the dimensions of an array view won't change the dimensions of the original array.

NumPy arrays can be sorted in several different ways with the `ndarray` class's method `sort`, which orders the numbers in an array *in place* (i. e. it changes the array and doesn't return anything). By default, this method sorts multidimensional arrays along their *last axis*. To sort an array along some other axis, set the `axis` argument of the `sort` method. For example,

```
>>> a = np.array([7, 1, 5, 5, 6, 0])
>>> a.sort()
>>> print(a)
[0 1 5 5 6 7]
>>> b = np.array([[9, 6, 4], [4, 8, 5], [9, 2, 9]])
>>> b.sort()
>>> print(b)
[[4 6 9]
 [4 5 8]
 [2 9 9]]
```

The command `b.sort()` is the same as `b.sort(axis = 1)`, i. e. order the numbers by column for each row. To sort the numbers in each column, i. e. order the numbers by row for each column, set `axis = 0`:

```
>>> b = np.array([[9, 6, 4], [4, 8, 5], [9, 2, 9]])
>>> b.sort(axis = 0)
>>> print(b)
[[4 2 4]
 [9 6 5]
 [9 8 9]]
```

By default, the sorting algorithm used by the `sort` method is the **quicksort** algorithm, which is a good choice for general purpose. We can select the sorting algorithm used by this method by setting the optional `kind` argument.

Two other sorting functions are worth mentioning. The NumPy function `argsort` returns the indexes that would sort an array instead of the sorted elements themselves:

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

```
array([2, 0, 3, 4, 5, 1], dtype=int64)
>>> a[np.argsort(a)]
array([0, 1, 2, 3, 3, 4])
```

The `argsort` function also takes the `axis` and `kind` arguments as described previously.

In addition, the NumPy function `searchsorted` takes a sorted array with one or more values, and returns the indexes in the array such that the values should be inserted before the indexes to maintain the order of the array:

```
>>> a = np.arange(1, 6)
>>> np.searchsorted(a, 4.5)
4
>>> np.searchsorted(a, (4.5, 0, 2.2))
array([4, 0, 2], dtype=int64)
```

2.9.5 Shape Manipulation of Arrays

No matter what's the rank of a NumPy array, its elements are stored in sequential memory locations. NumPy arrays are stored in memory in C-style — row-major order, i. e. with the elements of the last (rightmost) index stored contiguously. For example, in a two-dimensional array `a`, the element `a[0,0]` is followed by `a[0,1]`. So the following array

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

is stored in memory as the sequential elements `[1, 2, 3, 4]`.

Suppose you want to "flatten" a multidimensional array onto a single axis. You can use the `ndarray` class's methods `flatten` and `ravel` to do this. Both methods flatten the array into its internal (row-major) ordering, as described earlier. Note that `flatten` returns an independent copy of the flattened array whereas `ravel` returns a view of the flattened array. In general, `flatten` is slower than `ravel`. Below is an interaction with the Python interpreter showing how these methods work:

```
>>> a = np.array([[1, 2, 3], [4, 5, 6]])
>>> b = a.ravel()
>>> b[2] = 10
>>> c = a.flatten()
>>> c[2] = 20
>>> a
array([[ 1,  2, 10],
       [ 4,  5,  6]])
>>> b
array([ 1,  2, 10,  4,  5,  6])
>>> c
array([ 1,  2, 20,  4,  5,  6])
```

The **ndarray** class also provides the method **resize** for changing the shape and size of an array in place to a compatible shape. If the array doesn't reference the data of another array and doesn't have references to it, then resizing to a smaller shape is allowed and the array is truncated while resizing to a larger shape pads with zeros. Recall that array references are created when, say, one array is a view on another or simply by assignment. In contrast to **resize**, the method **reshape** change the shape of the array to a compatible size without changing its element. This method creates a new view object if possible; otherwise it creates a copy. Below is an interactive Python session demonstrating how to use these methods:

```
>>> a = np.arange(6)
>>> a.resize(4, 2)
>>> print(a)
[[0 1]
 [2 3]
 [4 5]
 [0 0]]
>>> a.resize(3, 3)
>>> print(a)
[[0 1 2]
 [3 4 5]
 [0 0 0]]
>>> b = np.arange(8)
>>> c = b.reshape(4, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
ValueError: cannot reshape array of size 8 into shape (4,3)
>>> c = b.reshape(2, 4)
>>> print(b)
[0 1 2 3 4 5 6 7]
>>> print(c)
[[0 1 2 3]
 [4 5 6 7]]
>>> b.resize(3, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot resize an array that references or is referenced
by another array in this way.
Use the np.resize function or refcheck=False
>>> c.resize(3, 3)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot resize this array: it does not own its data
```

Another useful `ndarray` class's method is `transpose` which returns a view of an array with the axes transposed (also available as the `T` attribute of the array). For a two-dimensional array, this is simply the usual matrix transpose:

```
>>> a = np.array([[1, 2], [3, 4], [5, 6], [7, 8]])
>>> b = a.transpose()
>>> b[1, 1] = 16
>>> a
array([[ 1,  2],
       [ 3, 16],
       [ 5,  6],
       [ 7,  8]])
>>> b
array([[ 1,  3,  5,  7],
       [ 2, 16,  6,  8]])
```

However, transposing a one-dimensional array returns the array unchanged:

```
>>> a = np.array([11, 12, 13, 14, 15, 16])
>>> a.transpose()
```

```
array([11, 12, 13, 14, 15, 16])
```

A clutch of methods of the `ndarray` class merge and split arrays in different ways. The methods `vstack`, `hstack`, and `dstack` stack arrays vertically (in sequential rows), horizontally (in sequential columns), and depthwise (along a third axis). Here is an interactive Python session illustrating how these methods work:

```
>>> a = np.array([1, 1, 1, 1])
>>> b = np.array([2, 2, 2, 2])
>>> c = np.array([3, 3, 3, 3])
>>> np.vstack((a, b, c))
array([[1, 1, 1, 1],
       [2, 2, 2, 2],
       [3, 3, 3, 3]])
>>> np.hstack((a, b, c))
array([1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3])
>>> np.dstack((a, b, c))
array([[[1, 2, 3],
        [1, 2, 3],
        [1, 2, 3],
        [1, 2, 3]]])
```

Note that the array created by these methods contains an *independent copy* of the data from the original arrays.

On the other hand, the inverse operations, `vsplit`, `hsplit`, and `dsplit`, split a single array into multiple arrays by rows, columns, or depth. In addition to the array to be split, these methods require an argument indicating how to split the array. If this argument is a single integer, then the array is split into that number of equal-sized arrays along the appropriate axis. For example,

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

Here a list of array objects is returned. If the second argument is instead a sequence of integer indexes, then the array is split on those indexes:

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

```
[array([1]), array([2, 3]), array([4, 5]), array([6])]
```

The result is the same as the list `[a[:1], a[1:3], a[3:5], a[5:]]`. Unlike with `hstack` and so on, the arrays returned are *views* on the original data.

NumPy also provides the function `newaxis` for adding an axis (i. e. dimension) to an array in the desired position:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> print(a)
[[1 2]
 [3 4]]
>>> a.shape
(2, 2)
>>> b = a[:, np.newaxis, :]
>>> print(b)
[[[1 2]]

 [[3 4]]]
>>> b.shape
(2, 1, 2)
```

2.9.6 Arithmetic Operations of Arrays

Arithmetic operations of NumPy arrays are performed elementwise and a new array is created to store the result. Below is an interaction with the Python interpreter demonstrating theses operations of NumPy arrays:

```
>>> a = np.arange(10, 30, 4)
>>> b = np.arange(5)
>>> print("a = {0}; b = {1}".format(a, b))
a = [10 14 18 22 26]; b = [0 1 2 3 4]
>>> a + b
array([10, 15, 20, 25, 30])
>>> a*b
array([  0,  14,  36,  66, 104])
>>> b/a
array([0.        , 0.07142857, 0.11111111, 0.13636364, 0.15384615])
```

```
>>> b**3
array([ 0,  1,  8, 27, 64], dtype=int32)
>>> a < 20
array([ True,  True,  True, False, False])
```

Like other arithmetic operators, the * operator operates elementwise for NumPy arrays. Indeed, matrix multiplication is implemented by the `dot` function of NumPy. For instance,

```
>>> a = np.array([[2, 4], [6, 8]])
>>> b = a + 3
>>> a*b
array([[10, 28],
       [54, 88]])
>>> np.dot(a, b)
array([[ 46,  58],
       [102, 130]])
```

If the input arguments are instead one-dimensional arrays, the `dot` function returns the standard dot product of the vectors. Similarly, the function `cross` returns the standard cross product of the vectors. Here is an example of working on these functions:

```
>>> a = np.array([1, 3, 5])
>>> b = np.array([2, 4, 6])
>>> np.dot(a, b)
44
>>> np.cross(a, b)
array([-2,  4, -2])
```

Notice that NumPy defines two special values to represent the outcome of calculations, which are not mathematically defined or not finite. The value `nan` ("not a number", NaN) represents the outcome of a calculation that is not a well-defined mathematical operation (e. g. 0/0) and `inf` represents infinity. For instance,

```
>>> np.arange(5)/0
__main__:1: RuntimeWarning: divide by zero encountered in true_divide
__main__:1: RuntimeWarning: invalid value encountered in true_divide
array([nan, inf, inf, inf, inf])
```

Don't test `nan` for equality as `np.nan == np.nan` always returns `False`. Moreover, `nan` is neither finite nor infinite. Indeed, NumPy provides the methods `isnan`, `isinf` and `isfinite` for checking NaNs, infinity, and finite values:

```
>>> np.isnan(np.arange(5)/0)
array([ True, False, False, False, False])
>>> np.isinf(np.arange(5)/0)
array([False,  True,  True,  True,  True])
>>> np.isfinite(np.arange(5)/0)
array([False, False, False, False, False])
```

2.9.7 Universal Functions

NumPy implements elementwise arithmetics for arrays of any dimension. Moreover, NumPy provides its own versions of mathematical functions which are implemented as so-called **universal functions** that act on each element of an array, producing an array in return without the need for an explicit loop. Below is an interactive Python session illustrating the use of these functions:

```
>>> a = np.linspace(0, 5, 6)
>>> np.cos(a*pi/5)
array([ 1.        ,  0.80901699,  0.30901699, -0.30901699, -0.80901699,
       -1.        ])
>>> np.sqrt(a*4)
array([0.        , 2.        , 2.82842712, 3.46410162, 4.        ,
       4.47213595])
>>> np.exp(a)
array([  1.        ,   2.71828183,   7.3890561 ,  20.08553692,
        54.59815003, 148.4131591 ])
```

With the use of universal functions, we can replace a loop for computing each element of an array by a vector/array expression like `exp(-a**2)*sin(a) + 3 + a**2`. Replacing a loop in this way is known as vectorization which promotes clean, efficient and easy-to-maintain code. The loop version is often referred to as **scalar code**. For example, the following Python code

```
import numpy as np
import math
N = 50
x = np.zeros(N)
y = np.zeros(N)
```

```
dx = 2.0/(N - 1)
for i in range(N):
    x[i] = -1 + dx*i
    y[i] = x[i]*math.exp(-x[i]**2)
```

is a scalar code which produces the same output as the vectorized code

```
>>> N = 50
>>> x = np.linspace(-1, 1, N)
>>> y = x*np.exp(-x**2)
```

Note that the requirement of vectorized code is that there are no explicit Python `for` loops. The loops required to compute each array element are instead performed in fast C or Fortran code in the NumPy package. Thus the list comprehensions

```
>>> x = np.array([-1 + dx*i for i in range(N)])
>>> y = np.array([xi*math.exp(-xi**2) for xi in x])
```

result in scalar codes since there are Python `for` loops operating on scalar quantities.

2.9.8 Logic and Comparisons of Arrays

In NumPy, there are many functions for comparing and performing elementwise logical operations on arrays. The most common of these functions are summarized in Table 2.20.

Table 2.20: Common NumPy functions for comparing and performing elementwise logical operations on arrays

Function	**Description**
`np.all(a)`	Determine whether *all* array elements of `a` evaluate to `True`
`np.any(a)`	Determine whether *any* array elements of `a` evaluate to `True`
`np.isreal(a)`	Determine whether each element of array `a` is a real number
`np.iscomplex(a)`	Determine whether each element of array `a` is a complex number
`np.isclose(a, b)`	Return a boolean array of the comparison between arrays `a` and `b` within some tolerance
`np.allclose(a, b)`	Return a `True` if *all* the elements in the arrays `a` and `b` are equal within some tolerance

Note that the NumPy functions **all** and **any** work in the same way as Python's built-in functions with the same name:

```
>>> a = np.array([1, 0, 2, 1, 0, 1, 1])
>>> np.any(a)
True
>>> np.all(a)
False
```

Moreover, the NumPy functions `isreal` and `iscomplex` return boolean arrays:

```
>>> a = np.array([2.0j, -3.2, 4.5-3.0j, 2, 10.8])
>>> np.isreal(a)
array([False,  True, False,  True,  True])
>>> np.iscomplex(a)
array([ True, False,  True, False, False])
```

Because the representation of floating point numbers is not exact in Python, comparing two **float** or **complex** arrays with the == operator is not always reliable and thus is not recommended. Instead, the best we can do is checking if two values are "close" to one another within some (typically small) absolute or relative tolerance. NumPy provides the function `isclose` for elementwise comparisons of two arrays `a` and `b` in this way. That is to say, it returns `True` for elements satisfying

```
abs(a - b) <= (atol + rtol*abs(b))
```

with absolute tolerance `atol` and relative tolerance `rtol` which are 10^{-8} and 10^{-5} respectively by default but can be changed by setting the corresponding arguments. An additional argument `equal_nan` has a default value of `False`, meaning that `nan` values in corresponding positions in the two arrays are treated as different; we should set `equal_nan = True` to treat such elements as equal. For example,

```
>>> a = np.array([4.13e-18, 2.89e-25, 7.54e-36, 6.02e23, np.nan])
>>> b = np.array([4.13e-18, 2.63e-22, 3.68e-28, 6.00e23, np.nan])
>>> np.isclose(a, b)
array([ True,  True,  True, False, False])
>>> np.isclose(a, b, equal_nan = True)
array([ True,  True,  True, False,  True])
```

Note that small numbers would be regarded as equal even though they may differ by many orders of magnitude. To correct this, set `atol = 0` to compare numbers within relative tolerance only. If `a` and `b` are the arrays defined in the previous example, then

```
>>> np.isclose(a, b, atol = 0)
array([ True, False, False, False, False])
```

Finally, the NumPy function `allclose` returns a single value: `True` only if every element in one array is equal to the corresponding element in the other array (within the tolerance defined by `atol` and `rtol`), and otherwise `False`:

```
>>> a = np.linspace(0, np.pi, 100)
>>> np.allclose(np.sin(a)**2, 1 - np.cos(a)**2)
True
```

2.9.9 Statistical Analysis of Arrays

NumPy provides a set of functions for performing statistical analysis, either on an entire array or an axis of it. For instance, we can find the sum of array elements using the NumPy function `sum`:

```
>>> a = np.linspace(0, 5, 11)
>>> a
array([0. , 0.5, 1. , 1.5, 2. , 2.5, 3. , 3.5, 4. , 4.5, 5. ])
>>> np.sum(a)
27.5
```

By default, all the elements of an array are summed by this function. If we want to find the sums along a given axis, use the `axis` argument:

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.sum(a)
10
>>> np.sum(a, axis = 0)
array([4, 6])
>>> np.sum(a, axis = 1)
array([3, 7])
```

We can also use the NumPy functions `min` and `max` to find the minimum and maximum values in an array. By default, the minimum and maximum values are selected over multiple axes of an array. If the `axis` argument is specified, then the maximum and minimum array elements along a given axis would be found. Notice that the minimum and maximum values found by these functions would be `nan` if the array contains one or more NaN values. To ignore NaN values instead, use the NumPy functions `nanmin` and `nanmax`. To illustrate how to use these functions, consider the following interaction with the Python interpreter:

```
>>> a = np.sqrt(np.linspace(-3, 3, 6))
__main__:1: RuntimeWarning: invalid value encountered in sqrt
>>> a
array([        nan,         nan,         nan, 0.77459667, 1.34164079,
        1.73205081])
>>> np.min(a), np.max(a)
(nan, nan)
>>> np.nanmin(a), np.nanmax(a)
(0.7745966692414832, 1.7320508075688772)
>>> b = np.array([[1, 5, 2], [4, 3, 6]])
>>> np.min(b)
1
>>> np.max(b)
6
>>> np.min(b, axis = 0)
array([1, 3, 2])
>>> np.min(b, axis = 1)
array([1, 3])
>>> np.max(b, axis = 0)
array([4, 5, 6])
>>> np.max(b, axis = 1)
array([5, 6])
```

Sometimes, we may want to look for the index of the maximum or minimum value of an array rather than the maximum or minimum value itself. This is what the NumPy functions `argmin` and `argmax` do. They have `nanargmin` and `nanargmax` variants too. By default, the index is into the flattened array. However, the index of the maximum or minimum value along a given axis of the input array would be given if the `axis` argument is specified. Note that in case of multiple occurrences of the minimum or maximum values, the indexes corresponding to the first occurrence are returned. For example,

```
>>> a = np.array([np.nan, 1.34164079, np.nan, np.nan, 1.73205081,
... 0.77459667])
>>> np.argmin(a), np.argmax(a)
(0, 0)
>>> np.nanargmin(a), np.nanargmax(a)
(5, 4)
>>> b = np.array([[1, 5, 2], [4, 3, 6]])
>>> np.argmax(b)
5
>>> np.argmax(b, axis = 0)
array([1, 0, 1], dtype=int64)
>>> np.argmax(b, axis = 1)
array([1, 2], dtype=int64)
```

On the other hand, NumPy provides the functions for computing the mean, median, weighted average, standard deviation, and variance of the values in an array. By default, all these functions perform the calculation over all the elements of the input array unless the `axis` argument is specified. In NumPy, the mean and median of array elements are calculated by the functions `mean` and `median`, respectively. Moreover, the weighted average of array elements is calculated by the function `average` as

$$\overline{x}_w = \sum_i^N w_i x_i \Big/ \sum_i^N w_i$$

where the weights w_i are supplied as a sequence having the same length as the array with elements x_i. As an illustration, consider the following interactive Python session:

```
>>> a = np.array([1, 4, 9, 16, 25])
>>> np.mean(a)
11.0
>>> np.median(a)
9.0
>>> np.average(a, weights=[1, 3, 2, 1, 2])
10.777777777777779
```

If you want the sum of the weights and the weighted average, set the `returned` argument to `True`. In the following example, we find these values for each row of an array:

```
>>> a = np.array([[1, 8, 27, 64], [2, 4, 6, 8]])
>>> av, sw = np.average(a, weights=[1, 0, 0.5, 1], axis=1,
... returned=True)
```

```
>>> print(av)
[31.4  5.2]
>>> print(sw)
[2.5 2.5]
```

The averages are therefore $(1 \times 1 + 0.5 \times 27 + 1 \times 64)/2.5 = 31.4$ and $(1 \times 2 + 0.5 \times 6 + 1 \times 8)/2.5 = 5.2$ where 2.5 is the sum of the weights.

By default, the NumPy function `std` computes the **uncorrected sample standard deviation**:

$$\sigma_N = \sqrt{\frac{1}{N}\sum_i^N (x_i - \overline{x})^2}$$

where x_i are the N observed values in the array and $\overline{x}$ is their mean. To calculate the **corrected sample standard deviation**

$$\sigma = \sqrt{\frac{1}{(N-\delta)}\sum_i^N (x_i - \overline{x})^2},$$

pass the value of δ to the argument `ddof` such that $N - \delta$ is the number of degrees of freedom in the sample. For instance, if the mean $\overline{x}$ is calculated from N sample values drawn from the population independently with replacement, there are $N - 1$ degrees of freedom in the vector of residuals used to calculate σ: $(x_1 - \overline{x}, x_2 - \overline{x}, \dots, x_N - \overline{x})$ and thus $\delta = 1$. To give you an idea, consider the following example for such case:

```
>>> a = np.array([2, 4, 6, 8, 10, 12])
>>> np.std(a)
3.415650255319866
>>> np.std(a, ddof=1)
3.7416573867739413
```

The function `nanstd` calculates the standard deviation in which `nan` values are ignored (so that N is the number of non-NaN values in the array). NumPy also has the functions `var` and `nanvar` for calculating the **variance** of the values in an array.

2.10 Dictionaries

Python provides a number of built-in data types for storing collections of related data. Besides lists, Python dictionaries of type **`dict`** is probably the most widely used one for such purpose. Other computer languages provide similar data structures called **hashes** or **associative arrays**. Roughly speaking, a Python dictionary is a list where the index can be a text. It can contain any objects as its **values** and each of its items is indexed by a unique **key**, which may be any immutable object. So the dictionary exists as a number of **key-value** pairs, which do not have any particular order. Notice that dictionaries themselves are mutable objects.

Python dictionaries can be created by placing a comma-separated list of key:value pairs inside curly braces {}. An empty dictionary is simply written as {}. For example,

```
>>> price = {"Apple Juice": 10.5, "Fruit Punch": 25.9,
...          "Melon Soda": 13.9, "Apple Juice": 19.9,
...          "Fresh Milk": 8.5, "Coffee": 7.9}
>>> price
{'Apple Juice': 19.9, 'Fruit Punch': 25.9, 'Melon Soda': 13.9, 'Fresh
Milk': 8.5, 'Coffee': 7.9}
```

Printing a dictionary will yield the same output as entering its name into the Python shell. If the same key is attached to different values (like `"Apple Juice"` here), only the most recent value survives since the keys in a dictionary are unique.

An individual item can be retrieved by indexing it with its key, either as a literal such as `"Melon Soda"` or with a variable equal to the key:

```
>>> price["Melon Soda"]
13.9
>>> drink = "Fresh Milk"
>>> price[drink]
8.5
```

Additional key-value pairs can be also added to a dictionary by indexing it in this way:

```
>>> price["Orange Juice"] = 8.5
>>> price["Grape Soda"] = 12.8
>>> print(price)
{'Apple Juice': 19.9, 'Fruit Punch': 25.9, 'Melon Soda': 13.9, 'Fresh
Milk': 8.5, 'Coffee': 7.9, 'Orange Juice': 8.5, 'Grape Soda': 12.8}
```

An alternative way of defining a dictionary is passing a sequence of (key, value) pairs to the constructor method **`dict`**. If the keys are simple strings just like those could be used as variable names, then the pairs can be also specified as keyword arguments to this constructor:

```
>>> marbles = dict([("Red", 34), ("Green", 30), ("Blue", 29)])
>>> marbles["Blue"]
29
>>> temps = dict(Bangkok=31, Dubai=42, Rome=33, Seoul=25)
>>> temps["Seoul"]
25
```

A `for`-loop iteration over a dictionary returns the dictionary keys:

```
>>> for t in temps:
...     print(t, temps[t])
...
Bangkok 31
Dubai 42
Rome 33
Seoul 25
```

If we index a dictionary with a key that does not exist, it will yield a `KeyError`:

```
>>> temps["Paris"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Paris'
```

Nevertheless, we can use the built-in method `get` to retrieve the value of a key in a dictionary if it exists, or some default value if it does not. If no default value is specified, then `None` is returned. For example,

```
>>> print(temps.get("Paris"))
None
>>> temps.get("Paris", -273)
-273
```

Python also provides three useful built-in methods `keys`, `values`, and `items` for dictionaries which respectively return a dictionary's keys, values, and key-value pairs (as tuples). In Python 2, each of these were returned in a list, but this is wasteful of memory for most purposes. For instance, calling `keys` will require all of the dictionary's keys to be copied as a list, which is simply iterated over in most cases. In other words, storing a whole new copy of the dictionary's keys is usually not necessary. Python 3 solves this problem by returning an iterable object, which accesses the dictionary's keys one by one, without copying them to a list. This saves running time and memory which is important for very large dictionaries. For example,

```
>>> locations = temps.keys()
>>> print(locations)
dict_keys(['Bangkok', 'Dubai', 'Rome', 'Seoul'])
>>> for location in locations:
...     print(location, temps[location])
...
Bangkok 31
Dubai 42
Rome 33
Seoul 25
```

A `dict_keys` object can be iterated over any number of times, but it is not a list and cannot be indexed or assigned:

```
>>> locations = temps.keys()
>>> locations[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'dict_keys' object does not support indexing
```

If you really want a list of the dictionary's keys, simply pass the `dict_keys` object to the `list` constructor:

```
>>> location_list = list(temps.keys())
>>> location_list[0]
'Bangkok'
>>> location_list[2] = "Honolulu"
>>> location_list
['Bangkok', 'Dubai', 'Honolulu', 'Seoul']
```

Beware that the last assignment only changes the `location_list` list. It does not alter the original dictionary's keys.

We can use the methods `values` and `items` in a similar fashion to retrieve a dictionary's values and items (i. e. key-value pairs). The objects returned by these methods are `dict_values` and `dict_items`. For example,

```
>>> temps.values()
dict_values([31, 42, 33, 25])
>>> temps.items()
dict_items([('Bangkok', 31), ('Dubai', 42), ('Rome', 33), ('Seoul',
25)])
>>> for temp_data in temps.items():
...     print(temp_data)
...
('Bangkok', 31)
('Dubai', 42)
('Rome', 33)
('Seoul', 25)
```

As mentioned before, any Python object whose contents cannot be changed can be used as keys in dictionaries. So the keys in a dictionary are not restricted to be strings. In particular, we may use integers as keys in a dictionary. This is a handy way of representing polynomials, which will be explained below.

We can regard the data associated with a polynomial as a set of power-coefficient pairs. To give you an idea, consider the polynomial

$$p(x) = 2 + 5x^3 - x^8.$$

In such case, the coefficient 2 belongs to power 0, the coefficient 5 belongs to power 3, and the coefficient -1 belongs to power 8. Then a dictionary can be used to represent a polynomial by mapping a power to each coefficient:

```
p = {0: 2, 3: 5, 8: -1}
```

Of course, a list can be also used to represent a polynomial by such mapping. But in this case we must fill in all the zero coefficients too since the index must match the power:

```
p = [2, 0, 0, 5, 0, 0, 0, 0, -1]
```

A major advantage of using a dictionary is that we need to store only the nonzero coefficients. For the polynomial $1 + x^{500}$, the dictionary holds two elements while the list holds 501 elements. The following function evaluates a polynomial which is represented as a dictionary:

```
def poly1(data, x):
    sum = 0.0
    for power in data:
        sum += data[power]*x**power
    return sum
```

The `data` argument must be a dictionary where `data[power]` holds the coefficient associated with the term `x**power`. A more compact implementation can make use of Python's sum function to sum the elements of a list:

```
def poly1(data, x):
    return sum([data[power]*x**power for power in data])
```

In this implementation, we first make a list of the terms in the polynomial using a list comprehension and then this list is feed to the `sum` function. Note that the name `sum` has different meaning in the two implementations. In the first one, `sum` is a `float` object; but `sum` is a function in the second one. When we set `sum = 0.0` in the first implementation, we bind the name `sum` to a new `float` object. Then the built-in Python summation function is no longer associated with the name `sum` inside the `poly1` function. Nevertheless, `sum` will be the summation function outside the `poly1` function (unless we have bound the global name `sum` to another object somewhere else in the main program).

With a list instead of a dictionary used to represent a polynomial, a slightly different evaluation function for the polynomial is required:

```
def poly2(data, x):
    sum = 0.0
    for power in range(len(data)):
        sum += data[power]*x**power
    return sum
```

If there are many zeros in the `data` list, `poly2` must perform many multiplications with the zeros while `poly1` computes with the non-zero coefficients only and is hence more efficient.

Another major advantage of using a dictionary rather than a list to represent a polynomial is that negative powers are easily allowed, e. g.,

```
p = {-2: 0.25, 3: -2}
```

can represent the polynomial $\frac{1}{4}x^{-2} - 2x^3$. With a list representation, negative powers require much more book-keeping. For example, we may set

```
p = [0.25, 0, 0, 0, 0, -2]
```

and remember that `p[0]` is the coefficient associated with the power -2. Notice that the `poly2` function will no longer work for such lists. But the `poly1` function still works for dictionaries with negative keys (powers).

Python 2.7 and 3.0 introduces a dictionary counterpart to list comprehensions called **dictionary comprehensions** for quickly generating parameterized key-value pairs with a **`for`** loop. Such construction is convenient to generate the coefficients in a polynomial. For instance, we can use dictionary comprehensions to create a dictionary `d` which contains the power-coefficient pairs of the Taylor polynomial of degree 20 for $\cos x$:

```
from math import factorial
d = {2*k: (-1)**k/factorial(2*k) for k in range(11)}
```

Indeed, a Python dictionary can also act as a kind of simple database. As an illustration, the following program `planetdict.py` stores the information about some astronomical bodies in a dictionary of tuples, keyed by the name of the planets, and manipulates them to produce a list of planet densities.

```
# planetdict.py
# This program manipulates the information about some astronomical
# bodies in a dictionary to produce a list of planet densities.
# Last Update on 19 Nov 2020 by F K Chow

import math

# Mass (in kg) and radius (in km) for some astronomical bodies
body = {"Sun": (1.988e30, 6.955e5),
        "Mercury": (3.301e23, 2440.),
        "Venus": (4.867e+24, 6052.),
        "Earth": (5.972e24, 6371.),
        "Mars": (6.417e23, 3390.),
        "Jupiter": (1.899e27, 69911.),
        "Saturn": (5.685e26, 58232.),
        "Uranus": (8.682e25, 25362.),
        "Neptune": (1.024e26, 24622.)}

# Remove the item for the Sun since it isn't a planet!
body.pop("Sun")
```

```
def calc_density(m, r):
    """ Returns the density of a sphere with mass m and radius r. """
    return m/(4/3*math.pi*r**3)

rho = {}
for planet in body.keys():
    m, r = body[planet]
    # Calculate the planet density in g/cm^3
    rho[planet] = calc_density(m*1000, r*1.e5)

for planet, density in sorted(rho.items()):
    print("The density of {0} is {1:3.2f} g/cm^3.".format(planet,
                                                            density))
```

Observe that two other built-in methods for dictionaries are used in this program. (Read the official Python documentation for more details about these methods). The command `body.pop("Sun")` removes the `body` dictionary's key-value pair with key `"Sun"` while the command `sorted(rho.items())` returns a `list` of the `rho` dictionary's key-value pairs, sorted by key. Here is the output of this program:

```
The density of Earth is 5.51 g/cm^3.
The density of Jupiter is 1.33 g/cm^3.
The density of Mars is 3.93 g/cm^3.
The density of Mercury is 5.42 g/cm^3.
The density of Neptune is 1.64 g/cm^3.
The density of Saturn is 0.69 g/cm^3.
The density of Uranus is 1.27 g/cm^3.
The density of Venus is 5.24 g/cm^3.
```

In section 2.7, we have discussed the syntax for passing arguments to functions. In that discussion, it was assumed that the function would always know the number of arguments could be passed to it and these were listed in the function definition such as

```
def func1(a, b, c):
        ⋮
```

Python provides a number of useful tools for handling the case where the number of arguments passed to a function is unknown. Including *args after any “formally defined” parameters places any additional positional argument into a **tuple**, args, as illustrated by the following code:

```
>>> def func1(a, b, *args):
...     print(args)
...
>>> func1(1, 2, 3, 4, 5, "ABC")
(3, 4, 5, 'ABC')
```

Inside the func function, in addition to the formal arguments a = 1 and b = 2, the arguments 3, 4, 5, and "ABC" are available as the items of the tuple args which can be arbitrarily long. Indeed, Python’s own built-in function **print** works in a similar way. It takes an arbitrary number of arguments to output as a string, followed by some optional keyword arguments:

```
def print(*args, sep=' ', end='\n', file=None):
          ⋮
```

It is also possible to collect arbitrary keyword arguments (see Section 2.7) for a function inside a dictionary by using the **kwargs syntax in the function definition. Using this syntax, Python takes any keyword arguments not specified in the function definition and packs them into the dictionary kwargs. For instance,

```
>>> def func2(a, b, **kwargs):
...     for k in kwargs:
...         print(k, "=", kwargs[k])
...
>>> func2(1, b=3, c=6, d=9, s="ABC")
c = 6
d = 9
s = ABC
```

One can also use *args and **kwargs when calling a function. This can be convenient such as with functions that take a large number of arguments:

```
>>> def func3(a, b, c, x, y, z):
...     print(a, b, c)
...     print(x, y, z)
...
```

```
>>> args = [1, 2, 3]
>>> kwargs = {"x": 4, "y": 5, "z": "ABC"}
>>> func3(*args, **kwargs)
1 2 3
4 5 ABC
```

2.11 Good Programming Style

Programmers may have different styles for writing computer programs. There are well written programs and poorly written ones. A well written program should have a simple structure, be easy to read and understand, and run fast. Note that it is important to make programs easy to read since an easy-to-read program makes it easier to find problems, easier to modify the code, and easier for other people to understand how it works.

To a certain extent, good programming is a matter of experience. And you will quickly get familiar of it after starting to write programs. But here are a few general guidelines that may help.

1. *Include comments in your programs.* Leave comments in the code to remind yourself the meaning of the variables, the calculations performed in different sections of the code, the arguments required by functions, and so forth. When you come back to a program written earlier, you may forgot how the program works. And you won't get into any trouble if you have included comments in the program. Comments are even more important if you are writing programs that other people will have to read and understand.

2. *Use meaningful variable names.* Give your variables names that help you remember what they represent. The names don't have to be long. In fact, very long names are usually harder to read. However, you should choose the names sensibly. For instance, use `E` for energy and `t` for time. Sometimes it's appropriate to use full words or even pairs of words to spell out what a variable represents, e. g. `force` and `angular_frequency`. In addition, if you are writing a program to calculate the value of a mathematical formula, give your variables the same names as in the formula.

3. *Use the right types of variables.* Use integer variables to represent quantities that actually are integers such as vector indices or quantum numbers. Use floats and complex variables for quantities that are really real or complex numbers.

4. *Import functions first.* If you are importing functions from packages, put your import statements at the start of your program. This makes them easy to find if you need to check them or add to them. It also ensures that you have imported functions before the first time they are used.

5. *Give your constants names.* If there are constants in your program like the charge q of a particle or the volume V of a gas, create similarly named variables at the beginning of your program to represent these quantities and use those variables wherever those quantities appear in your program. This makes formulas easier to read and understand. It also allows you to change the values of constants later by changing only a single line at the beginning of the program, even if the constant appears many times throughout your calculations. A physics program often has a section near the beginning (usually right after the import statements) that defines all the constants and parameters of the program, making them easy to find when you need to change their values.

6. *Employ user-defined functions wherever appropriate.* Use user-defined functions to represent repeated operations, especially complicated operations. Functions can make your program well-organized and greatly increase its legibility. However, you should not use functions for simple operations that can be represented by just one or two line of code because it's easier to follow the flow of the calculation if these operations are left in the main body. Normally you should put your function definitions at the start of your program (probably after imports and constant definitions). This ensures that each definition appears before the first use of the function it defines and the definitions can be easily found and modified whenever necessary.

7. *Print out partial results and updates throughout your program.* Large computational physics calculations can take a long time — minutes, hours, or even days. It will be helpful to include print statements in your program that print updates about where the program has got to or partial results from the calculations so that you know how the program is progressing. It's difficult to tell whether a calculation is working correctly if the computer simply sits silently for hours on end.

8. *Lay out your programs clearly.* Inside a Python program, you can add spaces or blank lines in most places without changing the operation of the program and doing so can improve readability. Make use of blank lines to split code into logical blocks and spaces to divide up complicated algebraic expressions or particularly long program lines. You can also split long program lines into more than one line if necessary.

9. *Don't make your programs unnecessarily complicated.* A short simple program is enor-

mously preferable to a long involved one. If the job can be done in ten or twenty lines, then it's probably worth doing it that way so that the code will be easier to understand. And if there are mistakes in the program, it will be easier to find where they lie.

Like good science, good programming is a matter of creativity and technical skill. As you gain more experience with programming, you will develop your own programming style and learn to write code in a way that makes sense to you and others.