An\_Introduction\_Python\_Computer\_Programming\_c03

p47

Chapter 3

The First Python Program

This chapter shows how to turn the computation power of the previous chapter into stand-alone Python programs, which can be executed on their own, independent of IDLE. To achieve such independence, the most important additional functionality that is required is input and output—or communication with users.

Este capítulo muestra cómo convertir el poder de cómputo del capítulo anterior en programas independientes de Python, que se pueden ejecutar por sí solos, independientemente de IDLE. **Para lograr tal independencia, la funcionalidad adicional más importante que se requiere es la entrada y salida, o la comunicación con los usuarios.**

3.1 Text Input and Output Using Strings

Text input and output (I/O) used to be the dominant user interface before graphical user interface (GUI) was invented in the 1980s. Nowadays, most system administrators, researchers and hackers still use the text console as the main I/O interface. Compared with GUI, text UI is relatively much easier to learn as a beginner. Python allows interaction between a program and a user via text IO, by proving a special type of objects: **strings**. A Python string can be treated as a sequence of text characters.

La entrada y salida de texto (E/S) solía ser la interfaz de usuario dominante antes de que se inventara la **interfaz gráfica de usuario (GUI) en la década de 1980.** Hoy en día, la mayoría de los administradores de sistemas, investigadores y piratas informáticos todavía utilizan la consola de texto como interfaz principal de E/S. En comparación con la GUI, la IU de texto es relativamente mucho más fácil de aprender como principiante. Python permite la interacción entre un programa y un usuario a través de texto IO, **al probar un tipo especial de objetos: cadenas**. Una cadena de Python se puede tratar como una secuencia de caracteres de texto.

String literals can be written in three different formats, the first being letters wrapped between a pair of single quotes:

Los literales de cadena se pueden escribir en tres formatos diferentes, siendo el primero letras entre comillas simples:

> > > s = ’ abc ’

> > > s

’ abc ’

> > > type ( s )

< type ’ str ’ >

In the example above, there are three text characters in the string s, namely ‘a’,

‘b’ and ‘c’. The type of the string s is str, which is the Python representation of the

string type.

The second way to write a string literal is to replace single quotes with double

quotes; this is convenient if the string itself contains single quotation marks.

> > > s = " he said : ’ hello ’"

> > > s

" he said : ’ hello ’"

> > > type ( s )

< type ’ str ’

In case both single and double quotes are in a string itself, a third form of string

literals can be used. It is specified by putting three double quotes on the left hand

side and three on the right hand side of the string. This form of string literals also

allows a string to span over multiple lines:

> > > s = " " " abc

... def

... ghi

... " " "

> > > s

’ abc \ ndef \ nghi \ n ’

> > > type ( s )

< type ’ str ’ >

**Line breaks are represented by a special character in the string, which can be written with the escaped form ‘\n’**. By default, Python displays the value of a string object in its escaped format, which is demonstrated by the example above. Escaped characters can also be used in a string literal directly.

> > > s = " abc \ ndef \ nghi "

> > > s

’ abc \ ndef \ nghi ’

In the example above, the two-character sequence ‘\n’, is used in the literal

explicitly to represent a new-line character. Note that explicit (i.e. not escaped) line

breaks are not allowed in string literals with single or double quotes. Another escape

character that has an escape form is the tab character, which can be written as ‘\t’.

> > > s = ’ abc def ghi ’

> > > s

’ abc \ tdef \ tghi ’

In the example above, the string s contains two explicit tab characters, which are

shown in escaped forms by Python. Note that the quotation marks around strings

serve only as indicators of a string literal—they are not a part of the string itself.

Hence ‘’ or “” specifies an empty string, a string that does not contain any character.

String operators. Similar to integers and floating point numbers, string objects also support a set of operators. A commonly used string operator is the concatenation operator +:

> > > s1 = ’ abc ’

> > > s2 = ’ def ’

> > > s1 + s2

’ a b c d e f ’

Figure 3.1 illustrates the changes that happen to the memory when the lines of

code above are executed. When the string literals ‘abc’ and ‘de f ’ are evaluated, two

corresponding string objects are constructed in the memory. They are associated with

their respective identifiers via the assignment statements. When the concatenation

operator is applied, a new string object is constructed, taking the concatenated value

of the two operands.

One thing to note is that the same operator + behaves differently when applied to

strings as compared to numbers. This fact is an example of polymorphism, which

will be discussed in Chap. 10.

The ∗ operator can also be applied to strings. It takes a string operand and an

integer operand, resulting in a string by repeating the string operand a number of

times, as specified by the integer operand.

> > > s = " abc "

> > > t = s \*3

> > > t

’ a b c a b c a b c ’

One important operator of strings is the getitem operator([]), which takes a string operand and an integer operand, resulting in the character in the string at the index specified by the integer. The getitem operator does not apply to numbers, such as integers and floating point numbers, but applies to other sequential objects, which will be introduced later in Chaps. 5 and 6. It takes the form of a pair of squared brackets, written immediately after the string operand and enclosing the integer operand, which specifies for the character index.

Un operador importante de cadenas es el **operador getitem ([])**, que toma un operando de cadena y un operando de entero, lo que da como resultado el carácter de la cadena en el índice especificado por el entero. El operador getitem no se aplica a números, como enteros y números de coma flotante, pero se aplica a otros objetos secuenciales, que se presentarán más adelante en los capítulos. 5 y 6. Toma la forma de un par de corchetes, escritos inmediatamente después del operando de cadena y encerrando el operando entero, que especifica el índice del carácter.

> > > s = ’ abc ’

> > > s [0]

# the f i r s t c h a r a c t e r

’ a ’

> > > s [1]

# the s e c o n d c h a r a c t e r

’ b ’

> > > s [2]

’ c ’

> > > x =0

> > > s [ x ]

# i n d e x can be any i n t e g e r e x p r e s s i o n

’ a ’

Note that string indices start from 0 rather than 1. **Hence s[i] stands for the i + 1th character in s when i ≥ 0**. This is also true for other sequential types in Python, and sequential types in many other programming languages.

Negative numbers can also be used to specify indices in Python. **Starting from −1, negative indices specify character indices from the right of the string**

> > > s = ’ abc ’

> > > i = -1

> > > s [ i ]

’ c ’

> > > i -=1

> > > i

-2

> > > s [ i ]

’ b ’

> > > i -=1

> > > i

-3

> > > s [ i ]

’ a ’

String indices must be within the valid range—trying to get a character that does not exist in the string will result in an error.

> > > s = ’ abc ’

> > > s [3]

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

I n d e x E r r o r : s t r i n g i n d e x out of r a n g e

> > > s [ -4]

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

I n d e x E r r o r : s t r i n g i n d e x out of r a n g e

A sub string can be extracted from a string by using the getslice operation ([]), which similar to the getitem operation by using the [] operator. It is expressed by replacing the character index in a getitem expression with a slice, which consists of a start and an end index, separated by a colon (:)

Una subcadena se puede extraer de una cadena usando la **operación getsslice ([])**, que es similar a la operación getitem usando el operador []. Se expresa reemplazando el índice de carácter en una expresión getitem con un segmento, que consta de un índice de inicio y uno de fin, separados por dos puntos (:)

> > > s = ’ abc ’

> > > s [ 1 : 2 ]

# i n d i c e s 1

’ b ’

> > > s [ 0 : 2 ]

# i n d i c e s 0 ,1

’ ab ’

> > > s [ 2 : 3 ]

# i n d i c e s 2

’ c ’

The way in which getslice is performed is illustrated by Fig. 3.2, where the locations at which a slice is taken is worth noting. Both the start index and the end index specify positions before characters. Similar to getitem, character indices start from 0. For example, ‘s[0 : 2]’ takes a slice out of s, by starting from before the first character (index = 0), and ending before the third character (index = 2), resulting in consisting of the first and second characters. Unlike getitem, getslice will not result in an index error; if the ending index is out of range, the slice will end with the last character in the string.

La forma en que se realiza getslice se ilustra en la figura 3.2, donde vale la pena señalar las ubicaciones en las que se toma un segmento. Tanto el índice inicial como el índice final especifican posiciones antes de los caracteres. De manera similar a getitem, los índices de caracteres comienzan desde 0. Por ejemplo, 's[0 : 2]' toma un segmento de s, comenzando desde antes del primer carácter (índice = 0) y terminando antes del tercer carácter (índice = 2), lo que resulta en el primer y segundo carácter. A diferencia de getitem, getslice no generará un error de índice; si el índice final está fuera de rango, el segmento terminará con el último carácter de la cadena.

> > > s = ’ abc ’

> > > s [ 2 : 5 ]

’ c ’

> > > s [ -2: -1]

’ b

By making the end index smaller than or equal to the start index, an empty string results from getslice.

> > > s = ’ abc ’

> > > s [ 2 : 2 ]

’ ’

> > > s [ 2 : 1 ]

’ ’

One or both indices in a getslice operation can be unspecified. The default value for an unspecified start index is 0, while the default value for an unspecified end index is the end of string. For example,

**Uno o ambos índices en una operación getsslice pueden no estar especificados. El valor predeterminado para un índice inicial no especificado es 0, mientras que el valor predeterminado para un índice final no especificado es el final de la cadena. Por ejemplo,**

> > > s = ’ abc ’

> > > s [1:]

# d e f a u l t end

’ bc ’

> > > s [:2]

# d e f a u l t s t a r t

’ ab ’

> > > s [:]

# both d e f a u l t

’ abc ’

To enhance the flexibility of slicing, a slice can be further specified by adding a third parameter, step, which is joined to the start and end indices by using an additional colon (:), and indicates an interval at which characters are taken from the string. For example,

Para mejorar la flexibilidad del corte, se puede especificar aún más un segmento agregando un tercer parámetro, paso, que se une a los índices de inicio y fin mediante dos puntos adicionales (:), e indica un intervalo en el que se toman los caracteres de la cadena. Por ejemplo,

> > > s = ’ a b c d e f ’

> > > s [ 0 : 6 : 2 ]

# 0 , 2 , 4

’ ace ’

> > > s [ 0 : 6 : 3 ]

’ ad ’

> > > s [ 1 : 5 : 2 ]

’ bd ’

When the step is 2, every second character is taken from the string; when the step is 3, every third character is taken. The start and end indices specify the same locations with or without the step parameter. The default value of the step parameter is 1, which results in a continuous sub string. When the step is larger than 1, a discontinuous sub string is extracted.

The step parameter can also be negative, in which case the slice is taken from right to left (i.e. right-to-left slicing), and hence the start index must be larger than the end index. Different from the left-to-right slicing, when slicing from right to left, indices indicate locations after corresponding characters. Similar to the left-to-right slicing, the absolute value of the step parameter specifies the interval at which characters are taken. For example,

**El parámetro de paso también puede ser negativo,** en cuyo caso el corte se toma de derecha a izquierda (es decir, corte de derecha a izquierda) y, por lo tanto, el índice inicial debe ser mayor que el índice final. A diferencia del corte de izquierda a derecha, cuando se corta de derecha a izquierda, los índices indican ubicaciones después de los caracteres correspondientes. Similar al corte de izquierda a derecha, el valor absoluto del parámetro de paso especifica el intervalo en el que se toman los caracteres. Por ejemplo,

> > > s = ’ a b c d e f ’

> > > s [5:2: -1]

# 5 , 4 , 3

’ fed ’

> > > s [:2: -1]

’ fed ’

> > > s [:: -1]

’ f e d c b a ’

> > > s [:: -2]

’ fdb ’

An important function that is associated with strings is **len**, which takes a single string argument and returns the number of characters in the string.

> > > s = ’ abc ’

> > > len ( s )

3

> > > s = ’ a b c d e f ’

> > > len ( s )

6

> > > s = ’ ’

> > > len ( s )

0

As shown in the last example above, the length of an empty string is 0.

Conversion between strings and other types. Similar to the int and float functions for type conversion into integers and floating point numbers, the str function can be used to convert floating point numbers and integers into strings.

> > > i =123

> > > f = 1 . 2 3

> > > s1 = str ( i )

> > > s1

’ 123 ’

> > > type ( s1 )

< type ’ str ’ >

> > > s2 = str ( f )

> > > s2

’ 1.23 ’

> > > type ( s2 )

< type ’ str ’ >

In the opposite direction, the int and float functions can turn strings that represent

integers and floating point numbers to integer and floating point objects, respectively.

> > > s = ’123 ’

> > > int ( s )

123

> > > s = ’1.23 ’

> > > f l o a t ( s )

3.1 Text Input and Output Using Strings 43

1.23

> > > int ( s )

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

V a l u e E r r o r : i n v a l i d l i t e r a l for int () with base 10: ’1.23 ’

> > > s = ’ abc ’

> > > f l o a t ( s )

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

V a l u e E r r o r : c o u l d not c o n v e r t s t r i n g to float : abc

The conversion from strings to integers and floating point numbers is strict: when

the string does not correspond to the target type, an error occurs. Two special strings

that can be converted to a floating point numbers are ‘inf’ and ‘-inf’, which represent

infinity and negative infinity, respectively. These two special floating point numbers

do not have literals, and must be converted from the strings ‘inf’ and ‘-inf’.

> > > a = f l o a t ( ’ inf ’ )

> > > type ( a )

< type ’ float ’ >

> > > a

inf

> > > b = f l o a t ( ’ - inf ’ )

> > > type ( b )

< type ’ float ’ >

> > > b

- inf

When a floating point number is converted into a string, Python automatically

rounds it up to a certain number of digits after the decimal point, so that the output

is more readable.

> > > 1 . 0 / 7

0 . 1 4 2 8 5 7 1 4 2 8 5 7

> > > s3 = str ( 1 . 0 / 7 )

> > > s3

’ 0 . 1 4 2 8 5 7 1 4 2 8 5 7 ’

> > > type ( s3 )

< type ’ str ’ >

If a specific number of digits after the decimal point is required in the output

string, the round function introduced in the previous chapter can be used to round up

the floating point number before it is converted into a string. Alternatively, a string

formatting expression can be used. String formatting expressions are a powerful

tool for specifying the format of numbers in a string. As a first example, consider the

formatting of integers:

> > > s = ’% d ’ %123

# n o r m a l f o r m

> > > s

> > > ’ 123 ’

> > > s = ’ %5 d ’ %123

# p a d d i n g to the left

> > > s

’ 123 ’

> > > s = ’ % -5 d ’ %123

# p a d d i n g to the f i r s t

> > > s

’ 123 ’

> > > s = ’ %1 d ’ %123

# no p a d d i n g

> > > s

’ 123 ’

A string formatting expression consists of two parts, separated by a % symbol (e.g. ′%5d′%123). The part on the left is a pattern string that contains a pattern %xd (e.g. ′%5d′), where % indicates the start of a pattern, and the letter d indicates that the formatted pattern is an integer. On the right hand side of the % symbol is theargument to fill the pattern, and in this case it is an integer to be formatted in thestring (e.g. 123). The x in the pattern is optional; it indicates the length of the string: if it is positive, spaces are padded on the left when the integer contains less digits than x, and if it is negative, spaces are padded on the right. If the integer contains more digits than the size specified by x, no space will be padded but the integer will not be truncated either.

**Una expresión de formato de cadena consta de dos partes, separadas por un símbolo % (por ejemplo, ′%5d′%123).** La parte de la izquierda es una cadena de patrón que contiene un patrón %xd (por ejemplo, ′%5d′), donde % indica el inicio de un patrón y la letra d indica que el patrón formateado es un número entero. En el lado derecho del símbolo % está el argumento para llenar el patrón, y en este caso es un número entero para ser formateado en la cadena (por ejemplo, 123). La x en el patrón es opcional; indica la longitud de la cadena: si es positivo, los espacios se rellenan a la izquierda cuando el entero contiene menos dígitos que x, y si es negativo, los espacios se rellenan a la derecha. Si el número entero contiene más dígitos que el tamaño especificado por x, no se rellenará ningún espacio, pero tampoco se truncará el número entero.

In addition to patterns, the pattern string can consist of other characters. Characters that are not a part of a pattern will remain unchanged when patterns are replaced with arguments during string formatting.

Además de patrones, la cadena de patrón puede constar de otros caracteres. Los caracteres que no forman parte de un patrón permanecerán sin cambios cuando los patrones se reemplacen con argumentos durante el formateo de cadenas.

> > > ’% dabc ’ %123

# p a t t e r n = % d

’ 123 abc ’

> > > ’ %5 dabc ’ %123

# p a t t e r n = %5 d

’ 123 abc ’

> > > ’ abc % -5 dd ’ %123

# p a t t e r n = % -5 d

’ a b c 1 2 3 d ’

To format a floating point number, the pattern in a pattern string is %x.y f , where x specifies the total size of the string, in the same way as integer formatting, y specifies the number of digits after the decimal point, and f marks a floating point pattern. x can be omitted, in which case no padding will be added.

Para dar formato a un número de punto flotante, el patrón en una cadena de patrones es %x.y f , donde x especifica el tamaño total de la cadena, de la misma manera que el formato de enteros, y especifica el número de dígitos después del punto decimal y f marca un patrón de punto flotante. x se puede omitir, en cuyo caso no se agregará relleno.

> > > ’ % f ’ % 1 . 2 3

# no p a d d i n g

’ 1.23 ’

> > > ’ %5.2 f ’ %1.23

# p a d d i n g to the l e f t ; two d i g i t s

a f t e r the d e c i m a l p o i n t

’ 1.23 ’

> > > ’ abc %5.2 f ’ %1.23

’ abc 1.23 ’

> > > ’ abc %5.2 f ’ % ( 1 . 0 / 7 )

’ abc 0.14 ’

> > > ’ abc %.2 f ’ % ( 1 . 0 / 7 )

# no p a d d i n g

’ abc0 .14 ’

More than one patterns can be defined in the pattern string, in which case a comma separated list of arguments must be given on the right within a pair of brackets. The patterns will be filled by the arguments in their input order.

Se puede definir más de un patrón en la cadena de patrón, en cuyo caso se debe dar una lista de argumentos separados por comas a la derecha entre un par de corchetes. Los patrones serán llenados por los argumentos en su orden de entrada.

> > > ’ a s t r i n g with one i n t e g e r % d f o l l o w e d by one

f l o a t i n g p o i n t n u m b e r %.2 f ’ % (123 , 1 . 2 3 )

’ a s t r i n g w i t h one i n t e g e r 123 f o l l o w e d by one

f l o a t i n g p o i n t n u m b e r 1.23 ’

If the number of patterns does not match the number of arguments, an error will be given:

> > > ’% d % d % f ’ % (1 , 2)

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

T y p e E r r o r : not e n o u g h a r g u m e n t s for f o r m a t s t r i n g

String representation in memory. As mentioned in the previous chapter, all types of information, including the alphabet, are ultimately represented by abstract information, or binary numbers, in memory. Python provides a built-in function, ord, which takes a single character as the only argument, and returns the integer that represents (i.e. encodes) the character:

Representación de cadenas en memoria. Como se mencionó en el capítulo anterior, todos los tipos de información, incluido el alfabeto, se representan en última instancia mediante información abstracta, o números binarios, en la memoria. Python proporciona una función integrada, **ord**, que toma un solo carácter como único argumento y devuelve el número entero que representa (es decir, codifica) el carácter:

> > > ord ( ’a ’)

97

> > > ord ( ’b ’)

98

> > > ord ( ’z ’)

122

> > > ord ( ’A ’)

65

> > > ord ( ’B ’)

66

> > > ord ( ’Z ’)

90

As can be seen from the example above, the representations of ’A’—’Z’ are 65–90,

and the representations of ’a’—’z’ are 97–122.

The reverse function to ord is chr, which takes an integer between 0 and 255 as the only argument, and returns the character interpretation of the integer.

La función inversa a ord es chr, que toma un número entero entre 0 y 255 como único argumento y devuelve la interpretación de caracteres del número entero.

> > > chr (97)

’ a ’

> > > chr (98)

’ b ’

> > > chr ( 1 0 6 )

’ j ’

3.1.1 Text IO

The print statement is used for text output; it shows the value of an object as a string on the text console. In most cases, the text console is the text terminal (i.e. Terminal in Linux and Mac OS, or Command Line in Windows). In the case of IDLE, the text console is IDLE itself. The syntax of the print statement is print x, where x is an expression. When print x is executed, the value of x is calculated first, and then converted into a string, before being displayed.

La declaración de impresión se utiliza para la salida de texto; muestra el valor de un objeto como una cadena en la consola de texto. En la mayoría de los casos, la consola de texto es el terminal de texto (es decir, Terminal en Linux y Mac OS, o Command Line en Windows). En el caso de IDLE, la consola de texto es IDLE en sí misma. La sintaxis de la declaración de impresión es imprimir x, donde x es una expresión. Cuando se ejecuta print x, el valor de x se calcula primero y luego se convierte en una cadena, antes de mostrarse.

> > > p r i n t 123

123

> > > p r i n t 3.6

3.6

> > > p r i n t ( 1 . 0 / 7 )

0 . 1 4 2 8 5 7 1 4 2 8 5 7

> > > p r i n t ’ abc ’

abc

> > > p r i n t 3+5 -2\*6

-4

> > > p r i n t ’ %.2 f ’ % 3 . 1 4 1 5

3.14

In the last two examples above, the values of the expressions 3 + 5 − 2 ∗ 6 and

′%.2 f ′%3.1415 are evaluated first, before being displayed on the console, respec-

tively. Here the string type is used as a bridge between the value of an expression

and the output console—an object is converted into a string before being displayed

on the console.

> > > 1 . 0 / 7

0 . 1 4 2 8 5 7 1 4 2 8 5 7 1 4 2 8 5

> > > str ( 1 . 0 / 7 )

’ 0 . 1 4 2 8 5 7 1 4 2 8 5 7 ’

> > > p r i n t 1 . 0 / 7

0 . 1 4 2 8 5 7 1 4 2 8 5 7

In the example above, the first two lines of code show Python’s internal repre-

sentation of the internal representation of the floating point number and its string

conversion, respectively. The third line of code is a print statement that writes the

value of the floating point number on the console. As can be seen from the example,

the floating point number displayed by the print statement is the same as the string

conversion, but different from the internal representation of the floating point number

itself. This is because the number has gone through a string conversion before being

printed.

The string conversion by the print statement is implicit—the str function is not

called explicitly in the statement. Later in Chap. 10 when custom types are introduced,

a custom string conversion process is discussed, and the implicit string conversion

by the print statement is reflected more directly.

The print statement displays a string on the console. This may appear to be a

redundant functionality, since IDLE will display the value of an expression anyway,

as the first two lines of code in the previous example demonstrate. However, as will

be shown in the next section, the print statement is necessary and important when

Python code is executed as a stand-alone program and independent of IDLE. In that

case, the value of an expression is not displayed when the expression is a single

line of code. In addition, although for integers and floating point numbers, Python’s

internal representation is similar to the string conversion, for many other types, the

two representations can be very different (see more details in Chap. 11), and hence

what IDLE displays can be rather different from what the print statement shows.

One final note is that when a string is printed, the escaped characters in the string

are displayed in their original form.

> > s = ’ abc \ ndef \ nghi \ tj ’

> > > s

# i n t e r n a l r e p r e s e n t a t i o n

’ abc \ ndef \ nghi \ tj ’

> > > p r i n t s

# p r i n t v e r s i o n

abc

def

ghi j

Because backslach(\) are used for escaped characters, their literal form in a string

is represented by ’\\’.

> > > s = ’ b a c k s l a c h is \\ ’

> > > s

’ b a c k s l a c h is \\ ’

> > > p r i n t s

b a c k s l a c h is \

Input from the text console can be achieved by using the raw\_input function, which asks the user to enter a string and returns the content of the string as a Python object. A command line prompt can be specified as an input argument to this function.

> > > s = r a w \_ i n p u t ( " E n t e r a s t r i n g : " )

E n t e r a s t r i n g : abc

> > > s

’ abc ’

In this example, when the first line of code is executed, the input argument “Enter

a string:”is displayed, with the program being temporarily stopped to wait for user

input. The user enters the letters ‘a’, ‘b’ and ‘c’, and then presses the Enter key to

complete the input. The function call then returns the string “abc” as its value.

Note that different from the case of the print statement, raw\_input(“Enter a

string:”) is a function call. It forms an expression on the right hand side of an

assignment statement in the example above, and is evaluated to a string object. The

specialness of this function call is its evaluation process: IDLE stops to ask the user

for text input, and evaluates the return value of the function call according to the

input, rather than according to the input string argument (e.g. “Enter a string:”),

which is used for console prompt only. This behavior is rather different from the

round, len and int function calls, which are all numerical and do not affect the exe-

cution of IDLE. However, it remains true that all function calls are expressions that

evaluate to a Python object.

Being an expression itself, raw\_input calls could be used as a part of a more

complex expression. For example,

> > > i = int ( r a w \_ i n p u t ( " E n t e r a n u m b e r : " ) ) \*2

E n t e r a n u m b e r : 56

> > > i

112

When the assignment statement above is executed, the right hand side of = is

first evaluated. It consists of (1) the evaluation of the raw\_input function call, (2)

the evaluation of the int function call, with the result of (1) being used as the input

argument, and (3) the evaluation of the \* operator, with the result of (2) being used

as one operand and the integer object 2 as the other. In step (1), IDLE displays the

prompt message and stops to ask the user for an input. The user enters the characters

‘5’, ‘6’, and presses Enter. The function therefore returns the string object ‘56’. In

step (2), the string is converted into the integer object 56. In step (3), the object is

multiplied by 2, resulting in the final value of the whole expression, 112. This object

is in turn bound to the identifier i in the assignment.

raw\_input is the basic mechanism in Python for accepting user input from a

text console. The return value of this function is always a string, and therefore type

conversion is necessary when the desired object is an integer, a floating point number

or other types. An alternative function for text console input is input, which performs

type conversion implicitly. The input function is implemented by adding an automatic

type conversion step after a raw\_input step. In effect, the function receives a literal

from the user, and converts it to a Python object as the return value.

> > > x = i n p u t ( " x = " )

x =123

> > > type ( x )

< type ’ int ’ >

> > > x

123

> > > x = i n p u t ( " x = " )

x = 1 2 . 3

> > > type ( x )

< type ’ float ’ >

> > > x

12.3

> > > x = i n p u t ( " x = " )

x = ’ abc ’

> > > type ( x )

< type ’ str ’ >

> > > x

’ abc ’

> > > x = i n p u t ( " x = " )

x =1+2 j

> > > type ( x )

< type ’ c o m p l e x ’ >

> > > x

(1+2 j )

> > > x = i n p u t ( " x = " )

x =1 L

> > > type ( x )

< type ’ long ’ >

> > > x

1 L

> > > x = i n p u t ( " x = " )

x = abc

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

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

N a m e E r r o r : name ’ abc ’ is not d e f i n e d

In the example above, the input function is called with the “x=” prompt being

displayed. An integer literal, a floating point number literal, a string literal, a complex

number literal, an explicit long number literal and an arbitrary string are entered,

respectively. While the literals of the corresponding types are converted to individual

objects, the arbitrary string, abc, causes an error, because it does not represent any

literal. In fact, it is treated as an identifier, and hence Python tries to look for the

objects that is associated with the name abc in the binding table, leading to a name

error. The correct way to enter a string literal, as shown by the ‘abc’ example, is to

include the quotations.

Because of the extra layer of functionality, the input function offers more flexibility

compared with the raw\_input function. However, it can also cause unexpected errors

if the user is not familiar with Python literals. From the programmer’s perspective,

the possibility for the return value of a function to be an arbitrary type can also lead

to difficulty in processing it. As a result, the raw\_input function with explicit type

conversion can be preferred when a certain type of object is expected from users.

3.2 The First Python Program

Text IO allows a stand-alone Python program to interact directly with the user. For a

simple Python program that does not rely on IDLE, consider the problem of interest

calculation. Suppose that a user has an initial sum of money in a savings account,

and the interest rate is fixed for a number of years. The problem is to find out the

total sum of savings after a certain number of years. A program can be written as

follows.

[ s a v i n g s . py ]

p r i n t ‘[ S a v i n g s C a l c u l a t o r ] ’

rate = float ( r a w \_ i n p u t (" P l e a s e e n t e r the i n t e r e s t rate

:") )

y e a r s = int ( r a w \_ i n p u t (" P l e a s e e n t e r the n u m b e r of

y e a r s :") )

init = float ( r a w \_ i n p u t (" P l e a s e e n t e r the i n i t i a l sum

of m o n e y : $ ") )

final = init \* (1.0+ rate ) \*\* years

p r i n t " A f t e r % d years , the s a v i n g s will be $ %.2 f " % (

years , f i n a l )

The program above, under the name “savings.py”, consists of 6 lines of code. It

can be typed in using any text editor, and saved into a specific folder in the file system.

For the convenience of illustration, suppose that it is saved under the Desktop folder.

The IDLE editor can also be used to edit the program. To start editing a new

program, go to the menu item [File→New Window], and a new text editor window

will be popped up. Enter the code above into the editor window, and go to the menu

item [File→Save...]. A dialog box will be opp up for entering the destination folder

and file name. Choose Desktop as the destination folder and savings.py as the file

name. A new Python code file savings.py will be created in the Desktop folder. It

also appears on the Desktop of the graphic user interface of the computer.

In order to open and edit an existing file using IDLE, go to the menu item

[File→Open...]. A dialog box will pop up for the selection of a specific file. In

the case of this example, select the file savings.py from the Desktop folder. A new

text editor window will pop up, which contains the current content of the file. Editing

can be performed in the editor window.

In order to execute the current program in the editor, go to the menu item

[Run→Run Module], or press the function key F5. The current window will be

switched to the interactive IDLE window, and the following result can be shown

according to the execution of the program.

> > > = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = R E S T A R T

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

[ S a v i n g s C a l c u l a t o r ]

P l e a s e e n t e r the i n t e r e s t rate : 0.036

P l e a s e e n t e r the n u m b e r of y e a r s : 5

P l e a s e e n t e r the i n i t i a l sum of m o n e y : $ 1 0 0 0 0

A f t e r 5\ , years , the s a v i n g s w i l l be $ 1 1 9 3 4 .35

> > >

The program displays the title ‘[Savings Calculator]’, and then asks the user to

input the interest rate, the number of years after which the total savings should be

calculated, and the initial sum of money. In this example, 0.036, 5 and 10000 are

entered as the three values, respectively. The program then displays the result in the

text console.

The Python program can also be executed directly using Python, under a text

console. On a Linux or Mac OS, Terminal is the default text console.

Zhangs - MacBook - Pro : a n y w h e r e $ cd

Zhangs - MacBook - Pro :~ y u e \_ z h a n g $ cd D e s k t o p /

Zhangs - MacBook - Pro : D e s k t o p y u e \_ z h a n g $ p y t h o n s a v i n g s .

py

[ S a v i n g s C a l c u l a t o r ]

P l e a s e e n t e r the i n t e r e s t rate : 0.036

P l e a s e e n t e r the n u m b e r of y e a r s : 5

P l e a s e e n t e r the i n i t i a l sum of m o n e y : $ 1 0 0 0 0

A f t e r 5\ , years , the s a v i n g s w i l l be $ 1 1 9 3 4 .35

Zhangs - MacBook - Pro : D e s k t o p y u e \_ z h a n g $

On a Windows OS, the Command Line tool is the default text console.

C :\ anywhere > cd % u s e r p r o f i l e %

C :\ U s e r s \ yuezhang > cd D e s k t o p

C :\ U s e r s \ y u e z h a n g \ Desktop > p y t h o n s a v i n g . py

[ S a v i n g s C a l c u l a t o r ]

P l e a s e e n t e r the i n t e r e s t rate : 0.036

P l e a s e e n t e r the n u m b e r of y e a r s : 5

P l e a s e e n t e r the i n i t i a l sum of m o n e y : $ 1 0 0 0 0

A f t e r 5\ , years , the s a v i n g s w i l l be $ 1 1 9 3 4 .35

C :\ U s e r s \ y u e z h a n g \ Desktop >

In both examples above, the cd command introduced in Chap. 1 is used to change

the current directory from an arbitrary location to the user’s home directory, and

then to the Desktop folder under this directory. Files in the folder are shown on the

desktop of the operating system. Direct execution of a Python program is done by

the command:

p y t h o n < p r o g r a m file path >

Now back to the program itself. It consists of 6 statements. The first and last

statements are two print statements, while the rest of the statements are assignment

statements. With respect to functionality, the first four statements inquire the user for

relevant inputs. The fifth statement calculates the answer based on user input, and

the last statements shows the result to the user.

The Python program is executed line by line, from top to bottom. It can be treated as

a list of statements, executed sequentially in a batch. The order is important, because

the execution of one statement can depend on the result of another. For example,

in the code above, final is calculated based on the user input of init. In subsequent

chapters, more complicated dynamic execution orders will be introduced. But by

default, Python always executes a program sequentially.

3.2.1 The Structure of Python Programs

The structure of a Python program can be summarized as Fig. 3.3, where a program

consists of a set of statements. Each statement can contain one or more mathemat-

ical expressions, and each expression consists of literals, identifiers and operators.

Python runs a program by executing its statements, each of which fulfills a specific

functionality.

As mentioned earlier, all statements are executed in a fixed and mechanical way.

Take the assignment statement introduced in the previous chapter for example. The

right hand side of = must be an expression, and the left hand side must be an identifier

(or a tuple, which will be introduced in Chap. 5). The statement is executed by

evaluating the expression first, and then binding the resulting object with associating

the left-hand-side identifier. Understanding the execution of each type of statement is

crucial to the understanding and design of Python programs. A list of the statements

that have been introduced up to this chapter is shown in Table 3.3

Note that in the previous chapter, an expression is sometimes used as a command

by itself. For example, by typing ‘3+5’ in IDLE, one obtains the result ‘8’. However,

as stated earlier, the usage above is possible only because IDLE displays the value

of an expression by default. In a standalone Python program, this is not possible.

Each expression is evaluated into a single object during program execution. An

expression is evaluated by applying operators to operands, which can be literals or

identifiers. Operators are applied according to a fixed precedence, and brackets can

be used to7 explicitly specify the order of evaluation. Table 3.2 give a summary of all

the operators that have been introduced, which include arithmetic operators (+, -, \*

etc.), the dot (.) operator, the indexing operator ([]) and the function call operator (()).

Their precedence is as follows. The dot, indexing and function call operators have

higher precedence than the arithmetic operators, while \*\* has higher precedence

than \*, /, // and % in arithmetic operators. Operators with the same precedence are

executed from left to right.

Both identifiers and literals represent Python objects in the underlying imple-

mentation. Each Python object is associated with a type. Two categories of types

have been introduced in this book, including numbers (int, float, long, complex) and

strings. While numbers are essential for mathematical computation, strings are useful

for text input and output functionalities. A list of the types that have been introduced

and their corresponding literals is shown in Table 3.1. More types will be introduced

in subsequent chapters.

Comments. In addition to statements, a Python program can also consist of com-

ments. Comments are not executed when a Python program is executed, but they

are useful for making Python programs more readable. It is a good habit to write

comments when programming, since proper comments can not only help other pro-

grammers understand the design of a program, but also remind a programmer of her

own thinking when designing the program. Comments are particularly important in

large software projects.

Python comments start with a # symbol. In a line of Python code, all the texts

after the first # symbol are treated as comments, and ignored during execution.

> > > g = 9 . 8 1

# the g r a v i t y c o n s t a n t

> > > y = g

# +1

> > > p r i n t y

# \* 2 # \* 3

9.81

In the example above, the first line contains some comments that explain the pur-

pose of the identifier g. The comments are useful for the understanding of subsequent

uses of the identifier. The second line in the example above illustrates another use of

comments, which is to temporarily delete a part of Python code. In this example, the

value of y is 9.81, rather than 10.81, since the text ‘+1’ is placed after a # symbol. In

this case, the programmer might have started with the statement y = g + 1, but then

decided to try y = x instead. She, however, does not want to completely remove the

‘+1’ part, since there is a chance that she wants to add it back after some testing.

Therefore, she chooses to keep ‘+1’ in the line but as a comment. In programming

terminology, she commented out the part of code. The last line above contains two

# symbols. It prints 9.81 on the console, but not 19.62 or 29.43, because all the text

after the first # symbol are treated as comments.

Comments can also be written as a single line, with the # symbol at the beginning

of the line. This form of comment is called in-line comment. In contrast, comments

after a statement is called an end-of-line comment. In-line comments are typically

used to explain the design of a set of multiple statements under the comment line.

Combined Statements. Multiple Python statements can be written in a single

line, separated by semi-colons (;)

> > > a =1; b =2; p r i n t a

# t h r e e s t a t e m e n t s c o m b i n e d

1

> > > b

2

Such combinations offer compactness in code, but can make programs more dif-

ficult to understand.

3.3 The Underlying Mechanism of Module Execution

Among the three types of statements in Table 3.3, the underlying mechanism of the

assignment statement has been introduced in much detail. The print statement has

been introduced in this chapter; it converts the value of an expression to string

and displays the string on the text console. The import statement, however, has been

introduced only briefly. This section gives more details on the underlying mechanism

of modules, which are useful for a better understanding of Python programs.

3.3.1 Module Objects

First, it has been introduced in the previous chapter that an imported module can

be accessed by an identifier that bares the same name of the module (i.e. math).

In addition, constants and functions defined in a module can be accessed by using

the module name and the dot (.) operator (i.e. math.e). The former functionality is

achieved by registering the name of the module in the binding table. To achieve

the latter functionality, an imported module is associated with a binding table of its

own, which contains the names of constants and functions defined in the module. For

example, the memory structure after the following three lines of code are executed

is shown in Fig. 3.4.

> > > x = 1

> > > y = ’ abc ’

> > > i m p o r t math

There are three identifiers in the main binding table: x, y and math. While x and

y are put into the binding table by the corresponding assignment statements, math is

inserted by the import statement. x and y are associated with the objects 1 and ‘abc’,

respectively. math, in contrast, is associated with a binding table, which contains

the constants and functions defined in the math module. In the figure, the name sqrt

is associated with a function object, the structure of which will be introduced in

Chap. 6.

The dot (.) operator syntactically connects a module name identifier to a second

identifier (e.g. math.e). The <module>.<identifier> expression is evaluated to the

value of identifier, looked up in the binding table of module. For example, when

the expression math.e is evaluated, Python first looks for the name math in the main

binding table. If it is found, Python follows the corresponding link to find the binding

table of math, in which is searches for the identifier e. In each of the two look-ups,

if the corresponding identifier is not found, an error will occur.

> > > i m p o r t math

> > > math . e

2 . 7 1 8 2 8 1 8 2 8 4 5 9 0 4 5

> > > math . e1

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

A t t r i b u t e E r r o r : ’ module ’ o b j e c t has no a t t r i b u t e ’ e1 ’

> > > m a t h 1 . e

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

N a m e E r r o r : name ’ math1 ’ is not d e f i n e d

In the example above, the second command (math.e) successfully returns the

value of math.e, since math has been imported to the current binding table and e is

a part of the math binding table. However, the third command (math.e1) raises an

error because the name e1 is not in the math binding table, and the fourth command

(math1.e) raises an error because the name math1 is not in the current binding table.

In Python, the dot (.) operator indicates a binding table, specified by the left operand.

An identifier is looked up in the specified binding table if it is the right operand of

a dot operator (e.g. math.e), but in the current binding table otherwise (e.g. e). The

main binding table of IDLE or a Python program is also called the global binding

table.

3.3.2 Library Modules

The Python modules that have been introduced in this book are used as libraries,

which are assemblies of code that provide specific functionalities for reuse. Most

Python Library modules are nothing but normal Python programs, typically stored

in a specific library path in the file system. The random module, for example, is a

Python program that can be found on a Linux or Mac OS at:

/ usr / lib / p y t h o n 2 .7/ r a n d o m . py

and on a Windows system at:

C :\ P y t h o n 2 7 \ Lib \ r a n d o m . py

Most libraries provided by the Python distribution are located in the folders above.

In addition to the Python distribution itself, libraries can also be downloaded from

a third party. For example, the SciPy toolkit contains libraries for many scientific

computation functionalities.1 After installation, such libraries are typically located

on a Linux or Max OS at:

/ L i b r a r y / P y t h o n / 2 . 7 / site - p a c k a g e s /

and on a Windows platform at:

C :\ P y t h o n 2 7 \ Lib \ site - p a c k a g e s \

Python automatically searches for the locations above when a library module is

imported, where the folder names depend on the Python version. Note that the math

module is an exception: it cannot be found under the library folder locations above.

This is because math is a very commonly used module, and is built in the main Python

program itself. Other commonly used modules, including cmath, are also built-in.

Making Use of Libraries. Before writing one’s own code for a specific function-

ality, it is wise to check whether there is an existing library module that can be used

directly. Python provides many powerful functionalities through modules distributed

along with the Python program itself, and the descriptions of all the modules can

be found in the Python documentation. To look for third-party modules with given

functionalities, online search engines are a useful resource. For example, searching

for ‘Python scientific computation’ can lead to the website of SciPy.

3.3.3 The Mechanism of Module Importation

The import statement creates an identifier that has the same name as the module.

For example, by using ‘import random’, ‘random.py’ is imported into the memory,

and associated with the name ‘random’ in the global binding table. To avoid name

clashes and for the convenience of reference, a module can also be given a different

name when imported into the binding table. This can be achieved by using a variant

of the import statement, with the syntax ‘import x as y’, where x is the name of the

module in the file system and y is the desired name in the binding table.

> > > i m p o r t math as m

> > > m . e

2 . 7 1 8 2 8 1 8 2 8 4 5 9 0 4 5

> > > m . s q r t ( 2 5 . 0 )

> > > 5.0

> > > math . e

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

N a m e E r r o r : name ’ math ’ is not d e f i n e d

In the example above, the math module is loaded into the binding table and

associated with the identifier m. As a result, by using “m.”, the corresponding binding

table of the math module can be accessed. However, the identifier “math” does not

exist in the global binding table this time.

Since both the assignment statement and the import statement changes the binding

table, they can override the value of an identifier. For example,

> > > x =1

> > > i m p o r t math as x

> > > x +1

# not a n u m b e r

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

T y p e E r r o r : u n s u p p o r t e d o p e r a n d type ( s ) for +: ’ module ’

and ’ int ’

> > > x . pi

# but a m o d u l e

3 . 1 4 1 5 9 2 6 5 3 5 8 9 7 9 3

In the example above, x is initially bound to a number, but then rebound to

a module. The original binding is overridden. An assignment statement can also

override an import statement. For example,

> > > i m p o r t math

> > > math =1

> > > math +1

# n u m b e r

2

> > > math . e

# e r r o r

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

A t t r i b u t e E r r o r : ’ int ’ o b j e c t has no a t t r i b u t e ’e ’

As shown earlier, modules such as random.py are Python programs themselves.

In fact, any Python program can be loaded as a module. To verify this, the program

saving.py can be used as an example. Suppose that another source file, textimport.py,

is created under the Desktop folder. It contains the following two lines of code

[ t e s t i m p o r t . py ]

i m p o r t s a v i n g s

x =1

p r i n t s a v i n g s . rate , " is e n t e r e d as the i n t e r e s t rate "

Executing the program above yields the following result.

Zhangs - MacBook - Pro :~ y u e \_ z h a n g $ cd D e s k t o p /

Zhangs - MacBook - Pro : D e s k t o p y u e \_ z h a n g $ p y t h o n

t e s t i m p o r t . py

[ S a v i n g s C a l c u l a t o r ]

P l e a s e e n t e r the i n t e r e s t rate : 0.036

P l e a s e e n t e r the n u m b e r of y e a r s : 5

P l e a s e e n t e r the i n i t i a l sum of m o n e y : $ 1 0 0 0 0

A f t e r 5 years , the s a v i n g s w i l l be $ 1 1 9 3 4 .35

0 . 0 3 6 is e n t e r e d as the i n t e r e s t rate

Zhangs - MacBook - Pro : D e s k t o p y u e \_ z h a n g $

One immediate thing to notice is that the program asks the user for the interest

rate, number of years and initial deposit again, just as if the savings.py program were

executed alone. Then it displays the message “0.036 is entered as the interest rate”.

Here, savings.rate is used to access the rate value defined in savings.py, in the same

way as the use of math.e to access the value of e defined in math.

The underlying mechanism of module importation is reflected in this example.

First, the importing module hands control over to the imported module, which is

executed, with its own binding table being the global binding table. Second, the

importing module resumes control, with the global binding table switched back to

the binding table of the importing module. A new entry added to the binding table of

the importing module, associating the name of the imported module with the binding

table of the imported module. This process can be illustrated in Fig. 3.5. The example

also shows a second use of the print statement, where a comma-separated list of

expressions is put after the keyword. In this case, each expression will be printed,

with a space separating the outputs

3.3.4 Duplicated Imports

If the same module is imported twice, the content of the module will not be exe-

cuted by the second import statement. For example, the following change to the

testimport.py code will not lead to any change in the runtime behavior.

[ t e s t i m p o r t . py ]

x =1

i m p o r t s a v i n g s

i m p o r t s a v i n g s

p r i n t s a v i n g s . rate , " is e n t e r e d as the i n t e r e s t rate "

The first two lines of the program remain the same as before, while the third line

is added for a second import of the same module. When this program is executed,

the user will be asked for input only once, with the same value of savings.rate being

shown when the fourth line of code is executed. Executing testimport.py from IDLE

can yield the following output.

> > > = = = = = = = = = = = = = = = = = = = = = = = = = = = = = = R E S T A R T

= = = = = = = = = = = = = = = = = = = = = = = = = = = = = =

[ S a v i n g s C a l c u l a t o r ]

P l e a s e e n t e r the i n t e r e s t rate : 0.036

P l e a s e e n t e r the n u m b e r of y e a r s : 5

P l e a s e e n t e r the i n i t i a l sum of m o n e y : $ 1 0 0 0 0

A f t e r 5 years , the s a v i n g s w i l l be $ 1 1 9 3 4 .35

0 . 0 3 6 is e n t e r e d as the i n t e r e s t rate

> > >

It can be seen in the example above that, no operation is taken when the second

import statement is executed. Note that for the example to run as expected, savings.py

must be in a folder that is accessible by IDLE. For example, if the Desktop folder has

been added to the PYTHON\_PATH environment variable as illustrated in Chap. 1, all

the files in the folder are accessible. Alternatively, if IDLE is started at the Desktop

folder, all the files in the folder are also accessible. The following example shows

how this can be done on a Linux or Max OS.

Zhangs - MacBook - Pro : a n y w h e r e $ cd

Zhangs - MacBook - Pro :~ y u e \_ z h a n g $ cd D e s k t o p /

Zhangs - MacBook - Pro : D e s k t o p y u e \_ z h a n g $ idle

The following example shows the case on a Windows OS.

c :\ anywhere > cd % u s e r p r o f i l e %

c :\ Users \ yue\_zhan g > cd D e s k t o p /

c :\ U s e r s \ y u e \_ z h a n g \ Desktop > idle

Two imports of the same module behave the same as in the previous example even

when they are executed from different modules—execution of the second import

statement does not result in the module being executed again. For example, suppose

that the following two files are in the Desktop folder.

[ t e s t i m p o r t 1 . py ]

i m p o r t s a v i n g s

x =1

[ t e s t i m p o r t 2 . py ]

i m p o r t s a v i n g s

i m p o r t t e s t i m p o r t 1

y =1

When testimport2.py is executed as the main program, the dynamic execution

sequence is:

i m p o r t s a v i n g s

[ e x e c u t e s a v i n g s . py ]

...

[ s a v i n g s . py f i n i s h e d ]

i m p o r t t e s t i m p o r t

[ e x e c u t e t e s t i m p o r t . py ]

i m p o r t s a v i n g s

[ e x e c u t e s a v i n g s . py ]

...

[ s a v i n g s . py f i n i s h e d ]

x =1

[ t e s t i m p o r t . py f i n i s h e d ]

y =1

savings.py is executed only once, when the first line of testimport2.py is executed.

When the second line of testimport2.py is executed, testimport1.py is executed. But

when the first line of testimport1.py is executed, savings.py is not executed a second

time. The same module object is simply associated with a new entry in the binding

table of testimport1. After the execution of y = 1, the memory structure is shown

in Fig. 3.6. Note that the same module object is associated with the name savings in

two different binding tables.

Reloading a module. In case the imported module is modified between the exe-

cution of two import statements, the second import will not reflect the modification.

Python has a special function, reload, which reloads the context of an imported

module by executing it once more. For example, suppose that savings.py is changed

between an import statement and a reload function call as follows:

a v i n g s . py i n i t i a l l y ]

p r i n t ‘[ S a v i n g s C a l c u l a t o r ] ’

rate = float ( r a w \_ i n p u t ( " P l e a s e enter the i n t e r e s t rate : " ) )

years = int ( r a w \_ i n p u t ( " P l e a s e e n t e r the n u m b e r of y e a r s : " ) )

init = float ( r a w \_ i n p u t ( " P l e a s e e n t e r the i n i t a l sum of m o n e y : $ " ) )

final = init \* (1.0+ rate ) \*\* years

print " After % d years , the s a v i n g s will be $ %.2 f " % ( years , final )

[ s a v i n g s . py a f t e r i m p o r t but b e f o r e r e l o a d ]

print ‘ The new version ’

init =0

Execution of the import and reload operations yields the following result:

> > > i m p o r t s a v i n g s

[ S a v i n g s C a l c u l a t o r ]

P l e a s e e n t e r the i n t e r e s t rate : 0.036

P l e a s e e n t e r the n u m b e r of y e a r s : 5

P l e a s e e n t e r the i n i t i a l sum of m o n e y : $ 1 0 0 0 0

A f t e r 5 years , the s a v i n g s w i l l be $ 1 1 9 3 4 .35

0 . 0 3 6 is e n t e r e d as the i n t e r e s t rate

> > > i m p o r t s a v i n g s

# n o t h i n g h a p p e n s

> > > r e l o a d ( s a v i n g s )

The new v e r s i o n

> > > p r i n t s a v i n g s . init

0

The reload function changes the context of the binding table of savings, where

the value of init becomes 0. As a result, 0 is printed by the last command in IDLE.

3.3.5 Importing Specific Identifiers

Sometimes only a small number of functions or variables needs to be imported from

a module. In such cases, the use of a module name in reference to the imported

functions or variables may be unnecessary. For example, suppose that only pi and

sin are needed by a program. In this case, importing the two identifiers directly into

the global binding table can be more convenient than importing the math module,

because in this way pi and sin can be referred to directly, instead of using math.pi

and math.sin.

Python provides a variation of the import statement for such imports. The syntax

is:

from < m o d u l e name > i m p o r t < i d e n t i f i e r s >

where <identifiers> is a comma-separated list of specific identifiers, or a wildcard

\* that represents all the identifiers in the module.

In the following example, the variable pi and the function sin are imported from

the math module directly.

> > > from math i m p o r t pi , sin

> > > sin ( pi )

1 . 2 2 4 6 4 6 7 9 9 1 4 7 3 5 3 2 e -16

> > > math . pi

T r a c e b a c k ( m o s t r e c e n t call last ) :

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

N a m e E r r o r : name ’ math ’ is not d e f i n e d

In the example above, the module name math cannot be found in the global

binding table, but the identifiers pi and sin are present. The mechanism for the from

... import ... statement is illustrated in Fig. 3.7. It consists of two steps: (1) execute

the module to be imported; (2) add the specified identifiers to the global binding

tables. The process is the same as the import ... statement in the first step, but differs

in the second step. As shown in Fig. 3.7, the resulting global binding table contains

the identifiers pi and sin, but not the identifier math. The binding table of the math

module is not referred to by any identifier in this case, and will be removed by the

garbage collector, together with the objects in the module other than pi and sin.

The following example shows the importing of all identifiers from the math

module.

> > > from math i m p o r t \*

> > > sin ( pi /2)

1.0

> > > log ( e )

1.0

> > > f a c t o r i a l (10)

3 6 2 8 8 0 0

> > > pow (2 , 8)

2 5 6 . 0

dir and the \_\_builtins\_\_ binding table

Python allows the listing of all entries in the global binding table by using the dir

function.

> > > dir ()

[ ’ \_ \_ b u i l t i n s \_ \_ ’ , ’ \_ \_ d o c \_ \_ ’ , ’ \_ \_ n a m e \_ \_ ’ , ’ \_ \_ p a c k a g e \_ \_ ’ ]

> > > x =1

> > > y =2

> > > dir ()

[ ’ \_ \_ b u i l t i n s \_ \_ ’ , ’ \_ \_ d o c \_ \_ ’ , ’ \_ \_ n a m e \_ \_ ’ , ’ \_ \_ p a c k a g e \_ \_ ’ ,

’x ’ , ’y ’]

In the example above, the first dir function call gives four entries, \_\_builtins\_\_,

\_\_doc\_\_, \_\_name\_\_ and \_\_package\_\_, which are the default entries when IDLE

(or a Python program) starts. Details about these entries will be introduced later in

Chap. 11. When the two assignment statements x = 1 and y = 2 have been executed,

two more entries, x and y are added to the global binding table, as shown by the

second call to the dir function.

The dir function can also take an input argument, which is an object that has a

binding table, and lists out all the entries in the binding table. As introduced earlier,

module objects are associated with binding tables. As a result, one can list out all the

identifiers defined in a module.

> > > i m p o r t math

> > > dir ( math )

[ ’ \_ \_ d o c \_ \_ ’ , ’ \_ \_ n a m e \_ \_ ’ , ’ \_ \_ p a c k a g e \_ \_ ’ , ’ acos ’ , ’ acosh ’

, ’ asin ’ , ’ asinh ’ , ’ atan ’ , ’ atan2 ’ , ’ atanh ’ , ’ ceil ’

, ’ c o p y s i g n ’ , ’ cos ’ , ’ cosh ’ , ’ d e g r e e s ’ , ’ e ’ , ’ erf ’ ,

’ erfc ’ , ’ exp ’ , ’ expm1 ’ , ’ fabs ’ , ’ f a c t o r i a l ’ , ’

floor ’ , ’ fmod ’ , ’ frexp ’ , ’ fsum ’ , ’ gamma ’ , ’ hypot ’ ,

’ i s i n f ’ , ’ i s n a n ’ , ’ l d e x p ’ , ’ l g a m m a ’ , ’ log ’ , ’ l o g 1 0 ’

, ’ log1p ’ , ’ modf ’ , ’ pi ’ , ’ pow ’ , ’ r a d i a n s ’ , ’ sin ’ , ’

sinh ’ , ’ sqrt ’ , ’ tan ’ , ’ tanh ’ , ’ trunc ’]

In the list above, the first three entries are special and exist for all modules; more

about them will be discussed in Chap. 11.

Note that the input argument to the dir function call must be available in the

current global binding table. Otherwise, an error will be given.

> > > dir ( r a n d o m )

# r a n d o m has not b e e n i m p o r t e d

T r a c e b a c k ( m o s t r e c e n t call last ) :

File " < p y s h e l l #6 > " , line 1 , in < module >

dir ( r a n d o m )

N a m e E r r o r : name ’ random ’ is not d e f i n e d

In addition to modules, other objects can also be associated with binding tables,

and they will be introduced in Chap. 10.

It can be noticed that the global binding table does not contain built-in functions

such as id, int, len and dir. The reason that they can be referred to without causing an

error is that Python organizes all built-in functions in a special module, \_\_builtins\_\_,

which is implicit in the system. When an identifier cannot be found in the global

binding table, the \_\_builtins\_\_ binding table will be searched. In fact, the \_\_builtins\_\_

in the global binding table by default, as shown earlier.

> > > dir ( \_ \_ b u i l t i n s \_ \_ )

[ ’ A r i t h m e t i c E r r o r ’ , ’ A s s e r t i o n E r r o r ’ , ’ A t t r i b u t e E r r o r ’

, ’ B a s e E x c e p t i o n ’ , ’ B u f f e r E r r o r ’ , ’ B y t e s W a r n i n

There are many built-in functions, such as round and len; many more will be

introduced in this book. By convention, Python names implicit or special objects by

enclosing their name in double underscores (\_\_).

Finally, note that the \_\_builtins\_\_ module is searched only when a reference to

the global binding table is not found. Identifiers in a specified binding table will not

be looked for in the \_\_builtins\_\_ binding table.

> > > i m p o r t math

> > > math . dir ()

T r a c e b a c k ( m o s t r e c e n t call last ) :

File " < p y s h e l l #9 > " , line 1 , in < module >

math . dir ()

A t t r i b u t e E r r o r : ’ module ’ o b j e c t has no a t t r i b u t e ’ dir ’

In the example above, because of the use of math., Python searches for the identi-

fier dir only in the math module. When it is not found, Python raises an error without

searching the \_\_builtins\_\_ module for the name dir.

Exercises

1. Does the following expression result in the string object ‘abc’? Why?

str ( abc )

2. Write a Python program that

(a) asks the user for a floating point number x0 , and prints out the value of

f (x) = x3 − 3x + 1 at x0 .

(b) asks the user for her name, and then prints “Hello, <name>! ”, when

<name> is the name of the user.

(c) asks the user for an integer, and then displays the number of digits in the

integer, and the logarithm of the integer.

(d) asks the user for an integer, and then displays the number of digits in the

integer to the power of 3.

(e) asks the user for a string s, and displays the following string: ‘<s>\*\*\*<s>

\*\*\*<s>’, where <s> is the string given by the user.

(f) asks the user for a string, and displays a string that repeats the input string

n times, where n is the length of the input string.

3. Write a program that asks the user for her name, and a short message below 50

characters, displaying the following output

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

\* \*

\* Hello , < name > \*

\* \*

\* Your m e s s a g e is : \*

\* \*

\* < The c o n t e n t of the message > \*

\* < m a y b e c o n t i n u e d message > \*

\* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \* \*

In the output above, each line contains 40 characters. <name> is the name of the

user. The string ‘Hello, <name>’ must be centered in the third line, regardless of

the length of <name>. The content of the message starts from the third character

of the seventh line. If it is more than 36 characters, the text should be wrapped

into the eighth line. Otherwise the eighth line is empty.

4. Write a program that asks for user input and performs

(a) Celsius to Fahrenheit conversion.

(b) Fahrenheit to Celsius conversion.

(c) Meter to foot conversion.

(d) Foot to meter conversion.

(e) Acre to square meter conversion.

(f) Square meter to acre conversion.

(g) Pound to kilogram conversion.

(h) Kilogram to pound conversion.

5. Make the answers to question 4 in the previous chapter full programs, which allow

arbitrary user input, giving the corresponding results.

6. Draw the binding tables after the following code is executed. How many times

are the math module executed?

[ lib . py ]

i m p o r t math

x = math . pi /2

[ main . py ]

i m p o r t lib

i m p o r t math

y = math . cos ( lib . x )

p r i n t y