Chapter 8

Modules and `import` Statements

The speed and accuracy with which computers perform computations are obviously important for solving problems or implementing tasks. However, quick and accurate computations, by themselves, cannot account for the myriad ways in which computers have revolutionized the way we live. Assume a machine is invented that can calculate the square root of an arbitrary number to any desired precision and can perform this calculation in a single attosecond (10^{-18} seconds). This is far beyond the ability of any computer currently in existence. But, assume this machine can only calculate square roots. As remarkable as this machine might be, it will almost certainly not revolutionize human existence.

Coupled with their speed and accuracy, it is flexibility that makes computers such remarkable devices. Given sufficient time and resources, computers are able to solve any problem that can be described by an algorithm. As discussed in Chap. 1, an algorithm must be translated into instructions the computer understands. In this book we write algorithms in the form of Python programs. So far, the programs we have written use built-in operators, built-in functions (or methods), as well as functions that we create.

At this point it is appropriate to ask: What else is built into Python? Perhaps an engineer wants to use Python and needs to perform calculations involving trigonometric functions such as sine, cosine, or tangent. Since inexpensive calculators can calculate these functions, you might expect a modern computer language to be able to calculate them as well. We'll return to this point in a moment. Let's first consider a programmer working in the information technology (IT) group of a company who wants to use Python to process Web documents. For this programmer the ability to use sophisticated string-matching tools that employ what are known as *regular expressions* might be vitally important. Are functions for using regular expressions built into Python? Note that an engineer may never need to use regular expressions while a typical IT worker may never need to work with trigonometric functions. When you consider all the different ways in which programmers want to use a computer, you quickly realize that it is a losing battle to try to provide all the built-in features needed to satisfy everybody.

Rather than trying to anticipate the needs of all programmers, at its core Python attempts to remain fairly simple. We can, in fact, obtain a list of all the "built-ins" in Python. If you issue the command `dir()` with no arguments, you obtain a `list` of the methods and attributes currently defined. One of the items in this `list` is `__builtins__`. If we issue the command

From the file: `import.tex`

`dir(__builtins__)` we obtain a `list` of everything built into Python. At the start of this `list` are all the exceptions (or errors) that can occur in Python (exceptions start with an uppercase letter). The exceptions are followed by seven items, which we won't discuss here, that start with an underscore. The remainder of this `list` provides all the built-in functions. Listing 8.1 demonstrates how this `list` can be obtained (for brevity, the exceptions and items starting with an underscore have been removed from the `list` that starts on line 8).

Listing 8.1 In the following, the integer variable `z` and the function `f()` are defined. When the `dir()` function is called with no arguments, we see these together with various items Python provides. Calling `dir()` with an argument of `__builtins__` provides a `list` of all the functions built into Python.

```
 1  >>> z = 111
 2  >>> def f(x):
 3  ...     return 2 * x
 4  ...
 5  >>> dir()
 6  ['__builtins__', '__doc__', '__name__', '__package__', 'f', 'z']
 7  >>> dir(__builtins__)
 8  [ <<START OF LIST DELETED>>
 9    'abs', 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes',
10    'callable', 'chr', 'classmethod', 'compile', 'complex', 'copyright',
11    'credits', 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval',
12    'exec', 'exit', 'filter', 'float', 'format', 'frozenset', 'getattr',
13    'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
14    'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
15    'locals', 'map', 'max', 'memoryview', 'min', 'next', 'object',
16    'oct', 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range',
17    'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
18    'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
19    'zip']
```

In lines 1 through 3 of Listing 8.1, the integer variable `z` and the function `f()` are defined. In line 5 the `dir()` function is called with no arguments. The resulting `list`, shown in line 6, contains these two items together with items that Python provides. The first item, `__builtins__`, is the one that is of current interest. Using this as the argument of `dir()` yields the `list` of functions built into Python (items in the `list` that are not functions have been deleted). The 72 built-in functions are given in lines 9 through 19.[1]

Several of the functions listed in Listing 8.1 have already been discussed, such as `dir()`, `divmod()`, `eval()`, and `float()`. There are some functions that we haven't considered yet but we can probably guess what they do. For example, we might guess that `exit()` causes Python to exit and that `copyright()` gives information about Python's copyright. We might even guess that `abs()` calculates the absolute value of its argument. All these guesses are, in fact, correct.

[1]In fact, not all of these are truly functions or methods. For example, `int()` is really a constructor for the class of integers, but for the user this distinction is really unimportant and we will continue to identify such objects as functions.

Of course, there are many other functions whose purpose we would have trouble guessing. We can use `help()` to provide information about these, but such information isn't really of interest now. What is important is that there are, when you consider it, a surprisingly small number of built-in functions. Note that there is nothing in this `list` that looks like a trigonometric function and, indeed, there are no trig functions built into Python. Nor are there any functions for working with regular expressions.

Why in the world would you want to provide a `copyright()` function but not provide a function for calculating the cosine of a number? The answer is that Python distributions include an extensive *standard library*. This library provides a math module that contains a large number of mathematical functions. The library also provides a module for working with regular expressions as well as much, much more.[2] A programmer can extend the capabilities of Python by *importing* modules or packages using an `import` statement.[3] There are several variations on the use of `import`. For example, a programmer can import an entire module or just the desired components of a module. We will consider the various forms in this chapter and describe a couple of the modules in the standard library. To provide more of a "real world" context for some of these constructs, we will introduce Python's implementation of complex numbers. We will also consider how we can import our own modules.

8.1 Importing Entire Modules

The `math` module provides a large number of mathematical functions. To gain access to these functions, we can write the keyword `import` followed by the name of the module, e.g., `math`. We typically write this statement at the start of a program or file of Python code. By issuing this statement we create a `module` object named `math`. The "functions" we want to use are really methods of this object. Thus, to access these methods we have to use the "dot-notation" where we provide the object name, followed by a dot (i.e., the access operator), followed by the method name. This is illustrated in Listing 8.2. The code is discussed following the listing.

Listing 8.2 Use of an `import` statement to gain access to the methods and attributes of the `math` module.

```
>>> import math
>>> type(math)
<class 'module'>
>>> dir(math)
['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh',
  'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
  'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1',
  'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma',
  'hypot', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
  'log10', 'log1p', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
```

[2]The complete documentation for the standard library can be found at **`docs.python.org/py3k/library/`**.

[3]We will not distinguish between modules and packages. Technically a module is a single Python `.py` file while a package is a directory containing multiple modules. To the programmer wishing to use the module or package, the distinction between the two is inconsequential.

```
11    'sqrt', 'tan', 'tanh', 'trunc']
12  >>> help(math.cos)      # Obtain help on math.cos.
13  Help on built-in function cos in module math:
14
15  cos(...)
16      cos(x)
17
18      Return the cosine of x (measured in radians).
19
20  >>> math.cos(0)         # Cosine of 0.
21  1.0
22  >>> math.pi             # math module provides pi = 3.1414...
23  3.141592653589793
24  >>> math.cos(math.pi)   # Cosine of pi.
25  -1.0
26  >>> cos(0)              # cos() is not defined.  Must use math.cos().
27  Traceback (most recent call last):
28    File "<stdin>", line 1, in <module>
29  NameError: name 'cos' is not defined
```

In line 1 the `math` module is imported. This creates an object called `math`. In line 2 the `type()` function is used to check `math`'s type and we see, in line 3, it is a `module`. In line 4 `dir()` is used to obtain a `list` of the methods and attributes of `math`. In the `list` that appears in lines 5 through 11 we see names that look like trig functions, e.g., `cos`, `sin`, and `tan`. There are other functions whose purpose we can probably guess from the name. For example, `sqrt` calculates the square root of its argument while `log10` calculates the logarithm, base 10, of its argument. However, rather than guessing what these functions do, we can use the `help()` function to learn about them. This is demonstrated in line 12 where the `help()` function is used to obtain help on `cos()`. We see, in line 18, that it calculates the cosine of its argument, where the argument is assumed to be in radians.

In line 20 the `cos()` function is used to calculate the cosine of `0` which is `1.0`. Not only does the `math` module provide functions (or methods), it also provides attributes, i.e., data or numbers. For example, `math.pi` is a finite approximation of π (i.e., $3.14159\cdots$, the ratio of the circumference to the diameter of a circle) while `math.e` is an approximation of Euler's constant (i.e., $e = 2.71828\cdots$). The cosine of π is -1 and this is confirmed in line 24 of the code. The statement in line 26 shows that the function `cos()` is not defined. For the way we have imported the `math` module, we must use the proper dot notation to gain access to its methods/functions, i.e., we must write `math.cos()`.

Assuming the `math` module has been imported, Listing 8.3 shows a portion of the output obtained from the command `help(math)`. After importing a module one can usually obtain extensive information about the module in this way.

Listing 8.3 Information about the `math` module can be obtained via `help(math)`. The following shows a portion of this information.

```
>>> help(math)
Help on module math:

NAME
    math

MODULE REFERENCE
    http://docs.python.org/3.2/library/math

    The following documentation is automatically generated from the
    Python source files.  It may be incomplete, incorrect or include
    features that are considered implementation detail and may vary
    between Python implementations.  When in doubt, consult the module
    reference at the location listed above.

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    acosh(...)
        acosh(x)

         Return the hyperbolic arc cosine (measured in radians) of x.
<<REMAINING OUTPUT DELETED>>
```

8.2 Introduction to Complex Numbers

Another data type that Python provides is the `complex` type for complex numbers. Complex numbers are extremely important in a wide range of problems in math, science, and engineering. Complex numbers and the complex mathematical framework that surrounds them provide beautifully elegant solutions to problems that are cumbersome or even impossible to solve using only real numbers. Complex numbers can be thought of as points in a plane, which is commonly referred to as the *complex plane*, as depicted in Fig. 8.1.

Instead of specifying the points as an ordered pair of numbers as you might with points in a Cartesian plane (i.e., an “(x, y) pair”), we describe points in a complex plane as consisting of a real part and an imaginary part. The use of the word “imaginary” is rather unfortunate since imaginary can imply something fictitious or made up. But imaginary numbers are just as real as “real” numbers—they provide ways for us to describe and quantify the world. The real part of a complex number represents the projection of the complex number onto the “real axis” (i.e., the

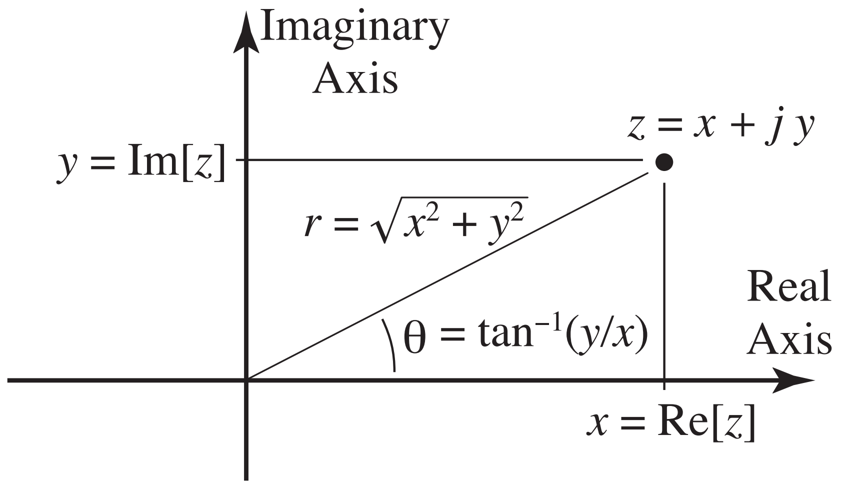

Figure 8.1: Representation of the complex number z in the *complex plane*. The number has a real part $\Re[z] = x$ (which gives the displacement along the horizontal, i.e., real, axis) and an imaginary part $\Im[z] = y$ (which gives the displacement along the vertical, i.e., imaginary, axis). The number can also be specified in polar form, $z = r\underline{/\theta}$ where the magnitude r is the displacement from the origin and the phase θ is the angle between the positive real axis and a ray from the origin to the point.

real number line). The *imaginary part* of a complex number is itself a real number! This number represents the projection of the complex number onto the "imaginary axis" which is orthogonal to the real axis. You can think of the real axis as corresponding to the x axis and the imaginary axis as corresponding to the y axis in a Cartesian plane. In fact, we often write a complex number z as $x + jy$ where x is the real part of z and y is the imaginary part of z. Again, y is a real number. However, j is defined as the "imaginary" number such that it is the square root of -1, i.e., $\sqrt{-1} = \pm j$ or, thought of another way, $(\pm j)^2 = -1$. In the complex plane, the number j is located along the imaginary axis a unit distance from the origin (a distance of 1 from the origin). When we write jy, we mean a displacement of y from the origin along the imaginary axis.

As with the usual representation of points in a plane, complex numbers can be also be expressed in polar form (ref. Fig. 8.1). We can write $z = r\underline{/\theta}$, where r is known as the magnitude (or modulus or absolute value) of z and is the displacement from the origin to the point z, and θ is known as the phase (or argument) and is the angle between the positive real axis and a ray from the origin to the point z. One can relate the Cartesian and polar forms of a complex number via $x = r\cos(\theta)$ and $y = r\sin(\theta)$. Though we won't explore it here, one of the remarkable and beautiful properties of complex numbers is described by Euler's formula which states $e^{j\theta} = \cos(\theta) + j\sin(\theta)$. Given this, we can write a complex number z either as $x + jy$ or $re^{j\theta}$.

To add two complex numbers, we simply add their real and imaginary parts (this is like vector addition for position vectors in a plane). So, in general, the sum of two complex numbers z_1 and z_2 is given by

$$z_1 + z_2 = (x_1 + jy_1) + (x_2 + jy_2) = (x_1 + x_2) + j(y_1 + y_2)$$

To multiply two complex numbers, we can follow the rules of multiplication that pertain to multiplying sums of real numbers. Thus, the product of z_1 and z_2 is given by:

$$z_1 \times z_2 = (x_1 + jy_1) \times (x_2 + jy_2) = x_1x_2 + j^2y_1y_2 + j(x_1y_2 + x_2y_1) = (x_1x_2 - y_1y_2) + j(x_1y_2 + x_2y_1)$$

A literal complex number is written in Python by writing `j` at the end of the numeric value representing the imaginary part. For example, in Python we write `1 + 1j` to obtain the complex number that is typically written in a math, engineering, or science course as $1 + j$. Note that if we write simply `j` or `j1`, these are treated as identifiers (i.e., treated as variables since these are valid identifiers). If these identifiers have not been previously defined, we obtain a syntactic error (an error in grammar). If they have been defined previously, we will obtain a semantic error (i.e., an error in meaning—Python will not produce an exception, but we will not obtain the complex number we wanted). In Python the complex number $5.6 - j7.3$ is written as `5.6 - 7.3j`. Writing `5.6 - j7.3` will result in an error since `j7.3` is neither a valid number nor a valid identifier.

The code in Listing 8.4 demonstrates the use of complex numbers. In line 1 the variable `z1` is assigned the complex value that is typically written as $1 + j$ in a math or engineering textbook. In line 15 the variable `z3` is assigned the value that is typically written as $5.6 - j7.3$. The discussion continues following the listing.

Listing 8.4 Demonstration of the use of complex numbers.

```
 1  >>> z1 = 1 + 1j       # Create complex number z1.
 2  >>> type(z1)         # Check z1's type.
 3  <class 'complex'>
 4  >>> z1               # Check z1.
 5  (1+1j)
 6  >>> j = 10           # Set j to 10.
 7  >>> z2 = 1 + j       # Create integer z2 -- not complex!
 8  >>> z2               # Check z2.
 9  11
10  >>> z3 = 5.6 - j7.3  # Attempt to create complex number z3.
11    File "<stdin>", line 1
12      z3 = 5.6 - j7.3
13                    ^
14  SyntaxError: invalid syntax
15  >>> z3 = 5.6 - 7.3j  # Correct way to create complex number z3.
16  >>> z1 + z3          # Sum of complex numbers.
17  (6.6-6.3j)
18  >>> z1 * z3          # Product of complex numbers.
19  (12.899999999999999-1.7000000000000002j)
```

In line 2 the `type()` function is used to show that `z1` is a `complex` object. In line 6 the variable `j` is assigned the integer value `10`. The statement in line 7 looks deceptively like the statement in line 1. However, while the statement in line 1 creates the complex number with a real and imaginary part of `1`, the statement in line 7 adds `1` to the current value of the variable `j`. Thus, `z2` is assigned the value `11` as shown in line 9.

The statement in line 10 is a failed attempt to assign the complex value $5.6 - j7.3$ to the variable `z3`. Again, the `j` used to indicate an imaginary number must be the trailing part of the number. Line 15 shows the correct way to make this assignment.

Line 16 produces the sum of `z1` and `z3` while line 18 yields their product. We can mix complex numbers with `floats` and integers in most arithmetic operations. (There are just a few exceptions to this. For example, we cannot use complex numbers with floor division or modulo operators.)

Listing 8.5 illustrates additional aspects of complex numbers. In line 1 the value $3 + j4$ is assigned to the identifier `z`. Then, in line 2, the `dir()` function is used to show the methods and attributes of this `complex` value. We are only concerned with the attributes `real` and `imag` that appear at the end of the listing. Note that they are attributes and not methods. Thus, to obtain the real and imaginary parts of `z` we write (without parentheses) `z.real` and `z.imag`, respectively. This is demonstrated in lines 13 through 16. Both the real and imaginary parts of a `complex` value are stored as `floats` as indicated in lines 17 and 18. The discussion continues following the listing.

Listing 8.5 The methods and attributes associated with a complex number.

```
1  >>> z = 3 + 4j
2  >>> dir(z)
3  ['__abs__', '__add__', '__bool__', '__class__', '__delattr__',
4   '__divmod__', '__doc__', '__eq__', '__float__', '__floordiv__',
5   '__format__', '__ge__', '__getattribute__', '__getnewargs__',
6   '__gt__', '__hash__', '__init__', '__int__', '__le__', '__lt__',
7   '__mod__', '__mul__', '__ne__', '__neg__', '__new__', '__pos__',
8   '__pow__', '__radd__', '__rdivmod__', '__reduce__',
9   '__reduce_ex__', '__repr__', '__rfloordiv__', '__rmod__',
10  '__rmul__', '__rpow__', '__rsub__', '__rtruediv__', '__setattr__',
11  '__sizeof__', '__str__', '__sub__', '__subclasshook__',
12  '__truediv__', 'conjugate', 'imag', 'real']
13 >>> z.real     # The real part of z.
14 3.0
15 >>> z.imag     # The real imaginary part of z.
16 4.0
17 >>> type(z.real)
18 <class 'float'>
19 >>> (z.real ** 2 + z.imag ** 2) ** 0.5   # Magnitude of z.
20 5.0
21 >>> abs(z)     # Simpler way to obtain magnitude.
22 5.0
```

As mentioned above, the magnitude of z is given by $r = \sqrt{x^2 + y^2}$. Line 19 shows one way to calculate this value. However, the built-in function `abs()` can also be used to obtain the magnitude of a complex number as demonstrated in lines 21 and 22.

8.3 Complex Numbers and the `cmath` Module

Given this background on complex numbers, we now can ask: How do we calculate the square root or cosine of a complex number? These are, in fact, well defined operations mathematically, but can Python do them?

Listing 8.6 illustrates what happens when we try to use complex numbers with the functions from the `math` module. In line 1 the `math` module is imported. In line 2 the complex number `z1` is created with a real part of zero and an imaginary part of `1.0`. (Note that if no real part is given, such as the case here, then it is zero. Also, both the real and imaginary parts of a complex number are stored as `floats`.) Line 3 shows that when we square `z1`, we obtain the complex number `(-1+j0)`. Another thing to note is that complex numbers are not converted back to `floats` when their imaginary part is zero nor are they converted to integers even when they correspond to whole numbers. The discussion continues following the listing.

Listing 8.6 Attempt to use complex numbers with functions from the `math` module.

```
 1 >>> import math
 2 >>> z1 = 1j      # z1 has zero real part and imaginary part of 1.
 3 >>> z1 * z1      # z1 squared is negative 1.
 4 (-1+0j)
 5 >>> # Cannot use math.sqrt() on a complex number.
 6 >>> math.sqrt(z1)
 7 Traceback (most recent call last):
 8   File "<stdin>", line 1, in <module>
 9 TypeError: can't convert complex to float
10 >>> # Cannot use math.sqrt() on a complex number.
11 >>> math.sqrt(-1)
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14 ValueError: math domain error
15 >>> # Cannot use math.cos() on a complex number.
16 >>> math.cos(5.6 - 7.3j)
17 Traceback (most recent call last):
18   File "<stdin>", line 1, in <module>
19 TypeError: can't convert complex to float
```

In lines 6 through 9 we see that we cannot use `math.sqrt()` with a complex number. In fact, lines 11 through 14 show that we cannot even use `math.sqrt()` with a negative real number. Lines 16 through 19 show that `math.cos()` cannot be used with complex numbers.

So, does that mean that one cannot calculate things such as $\sqrt{j}$ or $\cos(5.6 - j7.3)$ in Python? Not at all. Rather this serves as a reminder that Python tries to compartmentalize the functionality it provides. There are many programmers who may need to work with real numbers and real functions and yet will never need to work with complex numbers. In the interest of speed and program size, it is best if these programmers do not have to work with functions that are designed to deal with the "complexity" of complex numbers.

If a programmer needs to have complex functions at his or her disposal, they should import the `cmath` module. This is demonstrated in Listing 8.7.

Listing 8.7 Demonstration of the `cmath` module which provides support for complex numbers.

```
1  >>> import cmath
2  >>> dir(cmath)
3  ['__doc__', '__file__', '__name__', '__package__', 'acos', 'acosh',
4    'asin', 'asinh', 'atan', 'atanh', 'cos', 'cosh', 'e', 'exp',
5    'isfinite', 'isinf', 'isnan', 'log', 'log10', 'phase', 'pi',
6    'polar', 'rect', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
7  >>> z1 = 1j
8  >>> cmath.sqrt(z1)
9  (0.7071067811865476+0.7071067811865475j)
10 >>> cmath.cos(5.6 - 7.6j)
11 (774.8664714775274-630.6970442971782j)
12 >>> z2 = 3 + 4j
13 >>> cmath.polar(z2)
14 (5.0, 0.9272952180016122)
```

The `cmath` module is imported in line 1. The `dir()` function is used to show the methods and attributes in this module. If you compare lines 3 through 6 to lines 5 through 11 in Listing 8.2, you notice that the names of the functions in the `cmath` module largely duplicate those in the `math` module. There are just a few functions with new names in the `cmath` module.[4] But, despite the common names, the functions in these two modules are different, e.g., `cmath.sqrt()` is different from `math.sqrt()`. This is evident from the fact that we obtain the square root of `z1` in line 8 of Listing 8.7 but a similar statement produced an error in line 6 of Listing 8.6. Line 10 shows that we can calculate the cosine of a complex number if we use `cmath.cos()`.

In line 12 the complex value $3 + j4$ is assigned to the identifier `z2`. Then, in line 13, this value is given as the argument of the `polar()` function which returns, as a tuple, the magnitude and phase of the complex number. We see that the first element of this tuple agrees with the magnitude that was calculated in lines 19 and 21 of Listing 8.5. The second value of the tuple is the phase and is given in radians. Note that 0.927295 radians corresponds to approximately 53.13 degrees, i.e., $\tan^{-1}(4/3)$ expressed in degrees.

If one needs to import multiple modules, one either writes multiple `import` statements or writes a single `import` statement with the modules separated by commas. Thus,

```
import math, cmath
```

is equivalent to this

```
import math
import cmath
```

[4]The new functions are `phase()`, `polar()`, and `rect()`.

Again, despite the fact that there are methods within these modules that share a common name, these methods are distinct because the only way to access them is via the dot notation with the module objects (and the module objects do have distinct names).

There is one variation of the `import` statement that is appropriate to mention here. We are not required to use the module's name as the identifier we use within our code. We can import a module **as** some other identifier. For example, the statement

```
import math as realmath
```

will import the `math` module as `realmath` so that we must write `realmath.sqrt()` to access the square root function within this module. We can import more than one module with a single statement and use alternate identifiers for each. As an example, the code in Listing 8.8 imports the `math` module as `realmath` and the `cmath` module as `complexmath`.

Listing 8.8 Example of importing multiple modules with a single statement and using alternate identifiers for the modules.

```
>>> import math as realmath, cmath as complexmath
>>> realmath.sqrt(2)      # Real function, integer argument.
1.4142135623730951
>>> complexmath.sqrt(2)   # Complex function, integer argument
(1.4142135623730951+0j)
```

8.4 Importing Selected Parts of a Module

Often there is no reason to import all the methods and attributes contained in a module. Perhaps an engineer needs to work with the cosine function and the number π but does not need anything more than these from the `math` module. There are alternate forms of the `import` statement by which one can specify the individual objects to be imported. Listing 8.9 provides templates for these alternatives. Each statement starts with the keyword `from`. This is followed by the module name and then the keyword `import`. The remainder of the statement specifies what is to be imported and, if an `as` qualifier is present, what the identifier should be for the specified object (`as` is also a keyword). If more than one object is to be imported, the objects are separated by commas. When importing multiple objects, the `as` qualifier may be added or omitted as appropriate.

Listing 8.9 Using the `from-import` construct to select individual components of a module.

```
# Import a single object.
from <module> import <object>
# Import a single object and assign to an alternative name.
from <module> import <object> as <ident>
# Import multiple objects.
from <module> import <object1>, <object2>, ..., <objectN>
# Import multiple objects and assign to alternative names.
# The as qualifier may be omitted as appropriate.
from <module> import <object1> as <ident1>, ..., <objectN> as <identN>
```

To demonstrate importation of only selected objects from a module, consider the code shown in Listing 8.10 where the `sqrt()` function and the number `pi` are imported from the `math` module. The statement in line 1 accomplishes the desired importing. Since the `as` qualifier was not used, the objects are assigned to the identifiers they have within the module. The statements in lines 2 and 4 show that these objects were successfully imported. Line 6 shows that the `help()` function can still be used to obtain help on a function that has been imported. However, since the entire `math` module itself has not been imported, we cannot obtain help on the entire module. Thus, the statement in line 13 produces an error.

Listing 8.10 Demonstration of an `import` statement that employs `from` to select individual objects for importation.

```
1  >>> from math import sqrt, pi
2  >>> sqrt(2)
3  1.4142135623730951
4  >>> print(pi)
5  3.141592653589793
6  >>> help(sqrt)
7  Help on built-in function sqrt in module math:
8
9  sqrt(...)
10     sqrt(x)
11
12     Return the square root of x.
13 >>> help(math)
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in <module>
16 NameError: name 'math' is not defined
```

Now assume that we want to import the square root functions from both the `math` module and the `cmath` module. An attempt to do this is shown in Listing 8.11. In line 1 the `sqrt()` function is imported from the `cmath` module. In line 2 the `sqrt()` function is imported from the `math` module. Since these functions have the same name, the second `import` statement in line 2 associates the identifier `sqrt` with the `sqrt()` function from the `math` module—the `sqrt()` function from the `cmath` module is lost to us. Thus, in line 3, we produce an error when we attempt to take the square root of negative one (i.e., mathematically the result should be the complex number j, but the `sqrt()` function from the `math` module is not capable of producing this result).

Listing 8.11 Attempt to import the `sqrt()` function from both the `math` and `cmath` modules.

```
>>> from cmath import sqrt
>>> from math import sqrt
>>> sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

Listing 8.12 demonstrates how, using the `as` qualifier, we can import both `sqrt()` functions and keep them separate. In line 1 we import `sqrt()` from the `math` module and assign it to the identifier `rsqrt`. Also in line 1 we import the number `pi`. Since we do not provide an `as` qualifier, this is imported simply as `pi`. In line 2 the `sqrt()` function from the `cmath` module is imported and assigned to the identifier `csqrt`. Additionally, `pi` is imported and assigned to `cpi`. The statements in lines 3 and 5 show that these two different square-root functions are now available for use. The statement in line 7 is used to show that the value of `pi` in both `math` and `cmath` modules is indeed identical.

Listing 8.12 Demonstration of `import` statements that employ `from` to select individual objects for importation.

```
1 >>> from math import sqrt as rsqrt, pi
2 >>> from cmath import sqrt as csqrt, pi as cpi
3 >>> rsqrt(2)
4 1.4142135623730951
5 >>> csqrt(2+2j)
6 (1.5537739740300374+0.6435942529055826j)
7 >>> print(pi, cpi, pi - cpi)
8 3.141592653589793 3.141592653589793 0.0
```

8.5 Importing an Entire Module Using ⋆

Assume a programmer needs to work with most, or even all, of the objects contained in a module. In this case it can be tiresome to have to type `module.object` whenever accessing one of the objects. As you saw in the previous section, one can use the `from` variant of an `import` statement to import objects so that they are available directly in the local scope, i.e., we don't need to use the `module`-dot notation (in fact, we *cannot* use it). So, one possibility is to use a `from-import` statement where one lists all the objects in a module. However, this is rather cumbersome. Fortunately there is a better way.

Python allows you to import everything from a module if, in a `from-import` statement, you simply write an asterisk for the object you want to import. The asterisk serves as a "wildcard," meaning everything. The code in Listing 8.13 demonstrates how this is done.

Listing 8.13 Demonstration of directly importing all the contents of a module into the local scope.

```
1 >>> from math import *  # Import everything from the math module.
2 >>> pi
3 3.141592653589793
4 >>> sqrt(2)
5 1.4142135623730951
6 >>> # Cosine of pi / 2 is zero.  But, because of finite precision, the
7 >>> # result isn't exactly zero (but it's close!).
8 >>> cos(pi / 2)
9 6.123233995736766e-17
```

In line 1 the entire contents of the `math` module are imported. Line 2 shows `pi` is available. Line 4 shows the `sqrt()` function is available. Finally, line 8 shows the `cos` function is available. As something of an aside and related to the statement in line 8, cosine of $\pi/2$ is identically zero. But, because `pi` is a finite approximation to π, the result shown in line 9 differs slightly from zero.

There are some things to note when using the asterisk form of importing. Although it is certainly convenient, it has some drawbacks. Because this imports everything from a module, there is a chance that identifiers you use in your code will conflict with the identifiers used within the module. This can cause problems. Also, by importing in this way you will not have access to `help()` on the module as a whole (although help will be available for the individual functions that have been imported).

8.6 Importing Your Own Module

A module is simply a `.py` file. Thus, we can easily create our own modules that contain a collection of functions or other objects. Let us consider an example. Assume the code shown in Listing 8.14 has been placed in the file `my_mod.py`. The file starts with a multi-line string that gives a brief description of the module. In line 6 the variable `scale` is assigned the value `2`. Since this assignment is made outside any function, `scale` is globally defined within this module—any function can use or change this variable. Also, as we will see, this variable is "visible" wherever this module is imported. The module also contains two function definitions. The first function, `scaler()`, returns `scale` times its argument while the second function, `reverser()` returns the reverse of its argument (thus, of course, there are restrictions on what arguments can be passed to these functions). Note that the body of each function contains a docstring (docstrings were discussed in Sec. 4.3).

Listing 8.14 Code used to demonstrate importing a module of our own.

```
1 """
2 This module contains two functions and one globally defined
3 integer.
4 """
5
6 scale = 2
7
```

```python
def scaler(arg):
    """Return the scaled value of the argument."""
    return scale * arg

def reverser(xlist):
    """Return the reverse of the argument."""
    # Reverse using a slice with negative increment and default start
    # and stop.
    return xlist[ : : -1]
```

Now let's consider, as shown in Listing 8.15, what happens when we import this module and access the objects it contains. In line 1 we import the module. Importantly, note that the file itself is `my_mod.py` but *we do not include the* `.py` *in the* `import` *statement*. Lines 2 and 3 show us that `my_mod.scale` is set to `2`. In line 4 the `my_mod.scaler()` function is called with an argument of `42.0`. Since this function returns the product of its argument and `my_mod.scale`, we obtain `84.0` as shown in line 5. `my_mod.scaler()` can be called with a string argument as shown in line 6. Lines 8 and 9 demonstrate that `my_mod.reverser()` works properly. The discussion continues following the listing.

Listing 8.15 Importation and use of the module shown in Listing 8.14.

```
1  >>> import my_mod
2  >>> my_mod.scale
3  2
4  >>> my_mod.scaler(42.0)
5  84.0
6  >>> my_mod.scaler("liar")
7  'liarliar'
8  >>> my_mod.reverser("!pleH")
9  'Help!'
10 >>> my_mod.scale = 3     # Change value of my_mod.scale.
11 >>> my_mod.scaler("La")  # See that change affects my_mod.scaler().
12 'LaLaLa'
13 >>> help(my_mod)         # See what help is available for my_mod.
14 Help on module my_mod:
15 
16 NAME
17     my_mod
18 
19 DESCRIPTION
20     This modules contains two functions and one globally defined
21     integer.
22 
23 FUNCTIONS
24     reverser(xlist)
25         Return the reverse of the argument.
```

```
26
27      scaler(arg)
28          Return the scaled value of the argument.
29
30  DATA
31      scale = 3
32
33  FILE
34      /Users/schneidj/Documents/my_mod.py
```

In line 10 the value of `my_mod.scale` is changed to `3`. The call of `my_mod.scaler()` in line 11 shows that the output is affected by this change, i.e., the argument is now repeated three times instead of two. You should be aware, however, that this change did *not* change the value stored in the file `my_mod.py`—it remains `2` so that the next time the module is imported, `my_mod.scale` is 2.[5]

The remaining lines of Listing 8.15 show the output obtained when we ask `help()` to tell us about `my_mod`. The docstrings that were provided in the bodies of the functions as well as the string at the start of the file are used to describe the module. We also see, in line 31, that `scale` is part of the "`DATA`" in the module. Rather interestingly, we are told the current value of `scale` rather than the value it had when the module was first imported.

When working with our own modules, there are some additional things we must note. First, in the interest of speed, Python will import a module only once. So, for example, if you import a module into the interactive environment, notice a bug in the module, use an editor to correct the bug, and then issue another `import` statement in the same interactive session, Python does *not* re-import the module. There is a way to force Python to reload a module, but this can cause some subtle problems and thus we will not describe how to do this.[6] In situations like this, it is best simply to close the Python interpreter and start again. If you are using IDLE, you can simply select "Restart Shell" from the Shell menu or, if you are running your code from a file, IDLE will automatically restart the interpreter for you when you select "Run Module."

The second thing to be aware of is that Python needs to find your module in order to import it. Perhaps the simplest way to ensure a module is found is to put the module in one of the following places (which are appropriate for Python 3.2.x). On a Windows machine the module can be placed in

```
C:\Python32\Lib\site-packages
```

On a Macintosh, the module can be placed in

```
/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/site-packages
```

On a Debian/Ubuntu Linux machine, the module can be placed in

```
/usr/lib/python3.2/dist-packages
```

[5]However, after a module has been imported once during a session, it takes some additional commands in order to be able to re-import it.

[6]But, for the truly curious, you can do this by first importing the `imp` module and then using the `reload()` method of this module. So, for example, to "reload" the `my_mod` module, you would issue these commands: `import imp; imp.reload(my_mod)`.

Alternatively, one can modify where Python searches for modules. The directories where Python searches for modules are collectively known as the *path*. One of the places Python always looks is what is known as the “current working directory” (which is often abbreviated cwd). To see what Python considers the current working directory, you can issue these commands

```
import os     # Import the "operating system" module os.
os.getcwd()   # Display the current working directory.
```

If the current working directory doesn’t match where the module is located, one can use the `os.chdir()` to change the current working directory. The argument to `os.chdir()` should be a string appropriate for the operating system. For example, on a Macintosh, the current working directory can be set to the `Documents` folder for user `guido` by issuing the following statement

```
os.chdir("/Users/guido/Documents")
```

On a Windows box, where the desired folder is `My Documents`, the command might be

```
os.chdir("C:/Users/guido/My Documents")
```

Note the use of single forward slashes in this command! Typically one uses backslashes in Windows, but backslashes are used to indicate escape sequences in strings in Python. So, if you prefer to use backslashes, you have to use two of them, i.e.,

```
os.chdir("C:\\Users\\guido\\My Documents")
```

Rather than changing the working directory, one can modify the path over which Python searches for modules. Thus, on a Macintosh, to ensure Python searches in the `Documents` directory for the user `guido`, the appropriate commands are

```
import sys
sys.path.append("/Users/guido/Documents")
```

On a Windows machine where we want to search in the `My Documents` folder, the commands are

```
import sys
sys.path.append("C:/Users/guido/My Documents")
```

Again, backslashes can be used instead of forward slashes, but they have to be repeated twice.

8.7 Chapter Summary

Modules can by imported to enhance the capabilities of Python. To accomplish this, one uses an **`import`** statement.

The contents of an entire module can be imported using any of the following statements:

1. `import <module>`
2. `import <module> as <id>`
3. `from <module> import *`

In the second statement `<id>` serves as an alias for `<module>`. For the first and second statements, dot-notation is required to access an imported object, e.g., `<module>.<object>`. For the third statement, imported objects are directly visible, e.g., an imported function can be called simply using `<function>()`.

Multiple modules can be specified using the first statement with module names separated by commas. Multiple modules and their corresponding identifiers can be specified using the second statement with pairs of modules and identifiers separated by commas, e.g.,

```
import <mod1> as <id1>,\
       <mod2> as <id2>
```

Multiple modules cannot be specified using the third statement. Instead, multiple (separate) statements must be used.

A portion of a module can be imported using either of the following:

1. `from <module> import <object>`
2. `from <module> import <object> as <id>`

Multiple objects can be specified using the first statement with objects separated by commas. Multiple objects and their corresponding identifiers can be specified using the second statement with the pairs separated by commas, e.g.,

```
from <module> import \
     <obj1> as <id1>,\
     <obj2> as <id2>
```

A module must be in Python's path to be imported.

8.8 Review Questions

1. Which statement would import all of the objects of a module called `mymodule` so that they could be used directly (without dot notation).

   (a) `import mymodule`

   (b) `import mymodule.*`

   (c) `import * from mymodule`

   (d) `from mymodule import all`

   (e) `from mymodule import *`

2. What statement allows the `math` module to be used in a program?

   (a) `input math`

   (b) `import * from math`

   (c) `import math`

   (d) Either (b) or (c).

ANSWERS: 1) e; 2) c.