CHAPTER II: DATA TYPES, VARIABLES AND OTHER BASIC ELEMENTS

Topics covered:

Comments, Docstrings, Data types- Numeric Data type, Compound Data Type, Boolean Data type, Dictionary, Sets, Mapping, Basic Elements of Python, Variables

Comments

The previous line-reading program is one of the longest we have seen to date—so long, in fact, that we have added comments as well as a docstring.

The docstring is primarily for people who want to use the program; it describes what the program does but not how.

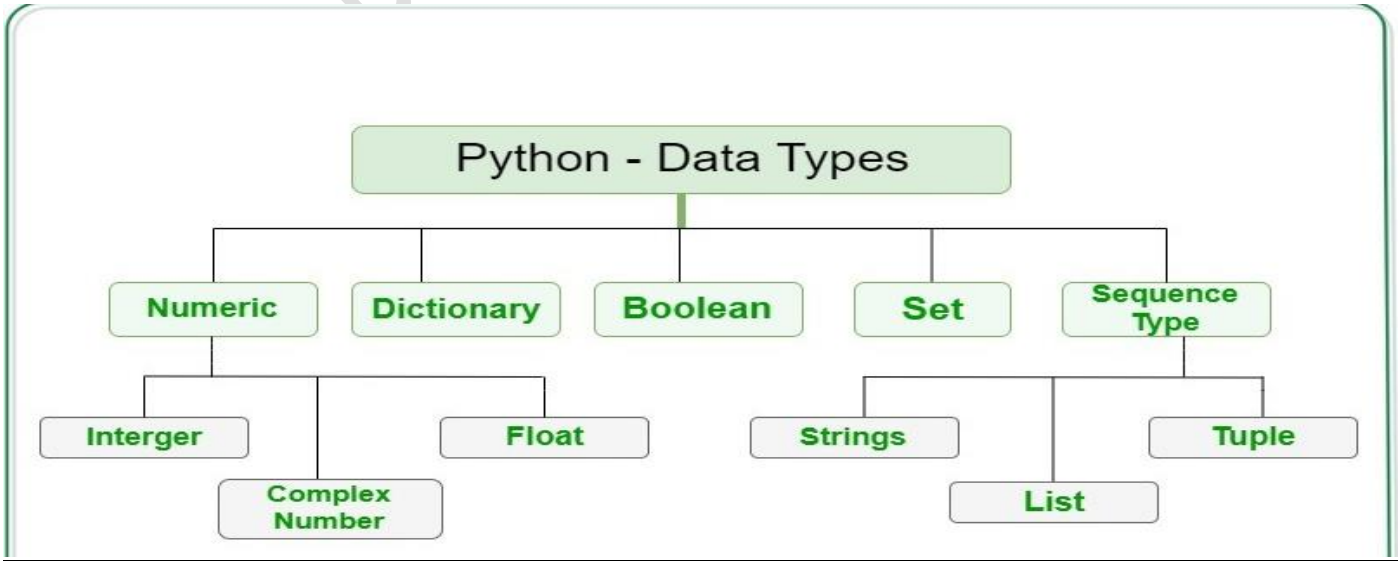
Comments, on the other hand, are written for the benefit of future developers.

- Each comment starts with the # character and runs to the end of the line. We can put whatever we want in comments, because Python ignores them completely.
- Here are a few rules for good commenting:
    - Assume your readers know as much Python as you do (for example, don't explain what strings are or what an assignment statement does).
    - Don't comment the obvious—the following comment is *not* useful:
      count = count + 1 # add one to count
    - Many programmers leave comments beginning with "TODO" or "FIXME" in code to remind themselves of things that need to be written or tidied up.
    - If you needed to think hard when you wrote a piece of software, you should write a comment so that the next person doesn't have to do the same thinking all over again. In particular, if you develop a program or function by writing a simple point-form description in English, then making the points more and more specific until they turn into code, you should keep the original points as comments.
    - Similarly, if a bug was difficult to find or if the fix is complicated, you should write a comment to explain it. If you don't, the next programmer to work on that part of the program might think that the code is needlessly complicated and undo your hard work.
    - On the other hand, if you need lots of comments to explain what a piece of code does, you should clean up the code.
    - For example, if you have to keep reminding readers what each of the fifteen lists in a function are for, you should break the function into smaller pieces, each of which works only with a few of those lists.

    And here's one more rule:

    • An out-of-date comment is worse than no comment at all, so if you change a piece of software, read the comments carefully and fix any that are no longer accurate.

Data types:



Identifiers and Keywords

When we create a data item we can either assign it to a variable, or insert it
Object into a collection. (As we noted in the preceding chapter, when we assign in Python, what really happens is that we bind an object reference to refer to the object in memory that holds the data.)
The names we give to our object references are called *identifiers* or just plain *names*.

A valid Python identifier is a nonempty sequence of characters of any length that consists of a "start character" and zero or more "continuation characters".

Such an identifier must obey a couple of rules and ought to follow certain conventions.
The first rule concerns the start and continuation characters. The start character can be anything that Unicode considers to be a letter, including the ASCII letters ("a", "b", …, "z", "A", "B", …, "Z"), the underscore ("_"), as well as the letters from most non-English languages. Each continuation character can be any character that is permitted as a start character, or pretty well any non-whitespace character, including any character that Unicode considers to be a digit, such as ("0", "1", …, "9"), or the Catalan character "·". Identifiers are case- sensitive, so for example, TAXRATE, Taxrate, TaxRate, taxRate, and taxrate are five different identifiers.
The precise set of characters that are permitted for the start and continuation
are described in the documentation (Python language reference, Lexical analysis,
Identifiers and keywords section), and in PEP 3131 (Supporting Non-ASCII Identifiers).

The second rule is that no identifier can have the same name as one of Python's keywords, so we cannot use any of the names shown in Table

Table 2.1 *Python's Keywords*

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

The first convention is: **Don't use the names of any of Python's predefined identifiers for your own identifiers.**

So, avoid using NotImplemented and Ellipsis, and the name of any of Python's built-in data types (such as int, float, list, str, and tuple), and any of Python's built-in functions or exceptions. How can we tell whether an identifier falls into one of these categories? Python has a
built-in function called dir() that returns a list of an object's attributes. If it is called with no arguments it returns the list of Python's built-in attributes.
For example:

```
>>> dir() # Python 3.1's list has an extra item, '__package__'
['__builtins__', '__doc__', '__name__']
```

The __builtins__ attribute is, in effect, a module that holds all of Python's built-in attributes. We can use it as an argument to the dir() function:

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
...
'sum', 'super', 'tuple', 'type', 'vars', 'zip']
```

NOTE: ★A "PEP" is a Python Enhancement Proposal. If someone wants to change or extend Python, providing they get enough support from the Python community, they submit a PEP with the details of their proposal so that it can be formally considered, and in some cases such as with PEP 3131, accepted and implemented. All the PEPs are accessible from www.python.org/dev/peps/.

There are about 130 names in the list, so we have omitted most of them. Those that begin with a capital letter are the names of Python's built-in exceptions; the rest are function and data type names.

The second convention concerns the use of underscores (_). Names that begin and end with two underscores (such as __lt__) should not be used. Python defines various special methods and variables that use such names (and in the case of special methods, we can reimplement them, that is, make our own versions of them), but we should not introduce new names of this kind ourselves.
We will cover such names in Chapter 6.Names that begin with one or two leading underscores (and that don't end with two underscores) are treated specially in some contexts.
A single underscore on its own can be used as an identifier, and inside an interactive interpreter or Python Shell, _ holds the result of the last expression that was evaluated. In a normal running program no _ exists, unless we use it explicitly in our code. Some programmers like to use _ in for …in loops when they don't care about the items being looped over. For example:

```
for _ in (0, 1, 2, 3, 4, 5):
print("Hello")
```

Be aware, however, that those who write programs that are internationalized often use _ as the name of their translation function. They do this so import that instead of writing
import
gettext.gettext("Translate me"), they can write _("Translate me"). (For this code to work we must have first imported the gettext module so that we can access the module's gettext() function.)
Let's look at some valid identifiers in a snippet of code written by a Spanishspeaking programmer. The code assumes we have done import math and that the variables radio and vieja_área have been created earlier in the program:

```
π = math.pi
ε = 0.0000001
nueva_área = π * radio * radio
if abs(nueva_área - vieja_área) < ε:
print("las áreas han convergido")
```

We've used the math module, set epsilon (ε) to be a very small floating-point number, and used the abs() function to get the absolute value of the difference between the areas—we cover all of these later in this chapter. What we are concerned with here is that we are free to use accented characters and Greek letters for identifiers. We could just as easily create identifiers using Arabic,
Chinese, Hebrew, Japanese, and Russian characters, or indeed characters from any other language supported by the Unicode character set.

The easiest way to check whether something is a valid identifier is to try to assign to it in an interactive Python interpreter or in IDLE's Python Shell window. Here are some examples:

```
>>> stretch-factor = 1
SyntaxError: can't assign to operator (...)
>>> 2miles = 2
SyntaxError: invalid syntax (...)
>>> str = 3 # Legal but BAD
>>> l'impôt31 = 4
SyntaxError: EOL while scanning single-quoted string (...)
>>> l_impôt31 = 5
>>>
```

When an invalid identifier is used it causes a SyntaxError Dealing with syntax errors exception to be raised.
In each case the part of the error message that appears in parentheses varies, so we have replaced it with an ellipsis. The first assignment fails because "-" is not a Unicode letter, digit, or underscore. The second one fails because the start character is not a Unicode letter or underscore; only continuation characters can be digits. No exception is raised if we create an identifier that
is valid—even if the identifier is the name of a built-in data type, exception, or function—so the third assignment works, although it is ill-advised. The fourth fails because a quote is not a Unicode letter, digit, or underscore. The fifth is fine.

Integral Types
Python provides two built-in integral types, int and bool.
★ Both integers and Booleans are immutable, but thanks to Python's augmented assignment operators this is rarely noticeable. When used in Boolean expressions, 0 and False are False, and any other integer

and True are True. When used in numerical expressions True evaluates to 1 and False to 0. This means that we can write some rather odd things—for example, we can increment an integer, i, using the expression i += True. Naturally, the correct way to do this is i += 1.

Integers

The size of an integer is limited only by the machine's memory, so integers hundreds of digits long can easily be created and worked with—although they will be slower to use than integers that can be represented natively by the machine's processor.

Table 2.2 *Numeric Operators and Functions*

Syntax	Description
`x + y`	Adds number `x` and number `y`
`x - y`	Subtracts `y` from `x`
`x * y`	Multiplies `x` by `y`
`x / y`	Divides `x` by `y`; always produces a `float` (or a `complex` if `x` or `y` is `complex`)
`x // y`	Divides `x` by `y`; truncates any fractional part so always produces an `int` result; see also the `round()` function
`x % y`	Produces the modulus (remainder) of dividing `x` by `y`
`x ** y`	Raises `x` to the power of `y`; see also the `pow()` functions
`-x`	Negates `x`; changes `x`'s sign if nonzero, does nothing if zero
`+x`	Does nothing; is sometimes used to clarify code
`abs(x)`	Returns the absolute value of `x`
`divmod(x, y)`	Returns the quotient and remainder of dividing `x` by `y` as a tuple of two `ints`
`pow(x, y)`	Raises `x` to the power of `y`; the same as the ** operator
`pow(x, y, z)`	A faster alternative to `(x ** y) % z`
`round(x, n)`	Returns `x` rounded to *n* integral digits if *n* is a negative `int` or returns `x` rounded to *n* decimal places if *n* is a positive `int`; the returned value has the same type as `x`; see the text

Table 2.3 *Integer Conversion Functions*

Syntax	Description
`bin(i)`	Returns the binary representation of `int i` **as a string, e.g.,** `bin(1980) == '0b11110111100'`
`hex(i)`	Returns the hexadecimal representation of `i` as a string, e.g., `hex(1980) == '0x7bc'`
`int(x)`	Converts object `x` to an integer; raises `ValueError` on failure—or `TypeError` if `x`'s data type does not support integer conversion. If `x` is a floating-point number it is truncated.
`int(s, base)`	Converts `str s` to an integer; raises `ValueError` on failure. If the optional *base* argument is given it should be an integer between 2 and 36 inclusive.
`oct(i)`	Returns the octal representation of `i` as a string, e.g., `oct(1980) == '0o3674'`

Integer literals are written using base 10 (decimal) by default, but other number bases can be used when this is convenient:

```
>>> 14600926 # decimal
14600926
>>> 0b110111101100101011011110 # binary
14600926
```

```
>>> 0o67545336 # octal
14600926
>>> 0xDECADE # hexadecimal
14600926
```

Binary numbers are written with a leading 0b, octal numbers with a leading 0o, and hexadecimal numbers with a leading 0x. Uppercase letters can also be used.

All the usual mathematical functions and operators can be used with integers, as Table 2.2 shows. Some of the functionality is provided by built-in functions like abs()—for example, abs(i) returns the absolute value of integer i—and other functionality is provided by int operators—for example,i + j returns the sum of integers i and j.

We will mention just one of the functions from Table 2.2, since all the others are sufficiently explained in the table itself. While for floats, the round() function works in the expected way—for example, round(1.246, 2) produces 1.25—for ints, using a positive rounding value has no effect and results in the same number being returned, since there are no decimal digits to work on. But when a negative rounding value is used a subtle and useful behavior is achieved—for example, round(13579, -3) produces 14000, and round(34.8, -1) produces 30.0.

All the binary numeric operators (+, -, /, //, %, and **) have augmented assignment versions (+=, -=, /=, //=, %=, and **=) where x *op*= y is logically equivalent to

x = x *op* y in the normal case when reading x's value has no side effects. Objects can be created by assigning literals to variables, for example, x = 17, or by calling the relevant data type as a function, for example, x = int(17). Some objects (e.g., those of type decimal.Decimal) can be created only by using the data type since they have no literal representation. When an object is created using its data type there are three possible use cases.

The first use case is when a data type is called with no arguments. In this case an object with a default value is created—for example, x = int() creates an integer of value 0. All the built-in types can be called with no arguments.

The second use case is when the data type is called with a single argument. If an argument of the same type is given, a new object which is a shallow copy of

the original object is created. Copying collections

If an argument of a different type is given, a conversion is attempted. This use is shown for the int type in Table 2.3. If the argument is of a type that supports conversions to the given type and the conversion fails, a ValueError exception is raised; otherwise, the resultant object of the given type is returned. If the argument's data type does Type conversions

not support conversion to the given type a TypeError exception is raised. The built-in float and str types both provide integer conversions; it is also possible to provide integer and other conversions for our own custom data types as we will see in upcoming chapters.

The third use case is where two or more arguments are given—not all types support this, and for those that do the argument types and their meanings vary. For the int type two arguments are permitted where the first is a string that represents an integer and the second is the number base of the string representation. For example, int("A4", 16) creates an integer of value 164.

This use is shown in Table 2.3.

The bitwise operators are shown in Table 2.4. All the binary bitwise operators

(|, ^, &, <<, and >>) have augmented assignment versions (|=, ^=, &=, <<=, and

>>=) where i *op*= j is logically equivalent to i = i *op* j in the normal case when

reading i's value has no side effects.

From Python 3.1, the int.bit_length() method is available. This returns **3.1**

the number of bits required to represent the int it is called on. For example,

(2145).bit_length() returns 12. (The parentheses are required if a literal integer

is used, but not if we use an integer variable.)

If many true/false flags need to be held, one possibility is to use a single integer, and to test individual bits using the bitwise operators. The same thing can be achieved less compactly, but more conveniently, using a list of Booleans.

Table 2.4 *Integer Bitwise Operators*

Syntax	Description
`i | j`	Bitwise OR of `int i` and `int j`; negative numbers are assumed to be represented using 2's complement
`i ^ j`	Bitwise XOR (exclusive or) of `i` and `j`
`i & j`	Bitwise AND of `i` and `j`
`i << j`	Shifts `i` left by `j` bits; like `i * (2 ** j)` without overflow checking
`i >> j`	Shifts `i` right by `j` bits; like `i // (2 ** j)` without overflow checking
`~i`	Inverts `i`'s bits

Booleans:

There are two built-in Boolean objects: **True** and **False**. Like all other Python data types (whether built-in, library, or custom), the bool data type can be called as a function—with no arguments it returns False, with a bool argument it returns a copy of the argument, and with any other argument it attempts to convert the given object to a bool. All the built-in and standard library data types can be converted to produce a Boolean value, and it is easy to provide
Boolean conversions for custom data types. Here are a couple of Boolean assignments and a couple of Boolean expressions:

```
>>> t = True
>>> f = False
>>> t and f
False
>>> t and True
True
```

Logical As we noted earlier, operators
Python provides three logical operators: and, or, and not.
Both and and or use short-circuit logic and return the operand that determined the result, whereas not always returns either True or False.
Programmers who have been using older versions of Python sometimes use 1 and 0 instead of True and False; this almost always works fine, but new code should use the built-in Boolean objects when a Boolean value is required.

Floating-Point Types :

Python provides three kinds of floating-point values: the built-in float and complex types, and the decimal. Decimal type from the standard library. All three are immutable. Type float holds double-precision floating-point numbers whose range depends on the C (or C# or Java) compiler Python was built with; they have limited precision and cannot reliably be compared for equality.
Numbers of type float are written with a decimal point, or using exponential notation, for example, 0.0, 4., 5.7, -2.5, -2e9, 8.9e-4.
Computers natively represent floating-point numbers using base 2—this means that some decimals can be represented exactly (such as 0.5), but others only approximately (such as 0.1and 0.2).Furthermore, the representation uses a fixed number of bits, so there is a limit to the number of digits that can be held. Here is a salutary example typed into IDLE:

```
>>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.4000000000000004, -2.5, 0.00089999999999999995)
```

Guages have this problem with floating-point numbers.
Python 3.1 produces much more sensible-looking output: **3.1**

```
>>> 0.0, 5.4, -2.5, 8.9e-4
(0.0, 5.4, -2.5, 0.00089)
```

When Python 3.1 outputs a floating-point number, in most cases it uses David

Gay's algorithm. This outputs the fewest possible digits without losing any accuracy. Although this produces nicer output, it doesn't change the fact that computers (no matter what computer language is used) effectively store floating-point numbers as approximations.
If we need really high precision there are two approaches we can take. One approach is to use ints—for example, working in terms of pennies or tenths of a penny or similar—and scale the numbers when necessary. This requires us to be quite careful, especially when dividing or taking percentages. The other approach is to use Python's decimal. Decimal numbers from the decimal module.
These perform calculations that are accurate to the level of precision we specify (by default, to 28 decimal places) and can represent periodic numbers like 0.1 exactly; but processing is a lot slower than with floats. Because of their accuracy, decimal. Decimal numbers are suitable for financial calculations.
Mixed mode arithmetic is supported such that using an int and a float produces a float, and using a float and a complex produces a complex. Because decimal.
Decimals are of fixed precision they can be used only with other decimal.
Decimals and with ints, in the latter case producing a decimal. Decimal result.
If an operation is attempted using incompatible types, a TypeError exception is raised.

Floating-Point Numbers

All the numeric operators and functions in Table 2.2 can be used with floats, including the augmented assignment versions. The float data type can be called as a function—with no arguments it returns 0.0, with a float argument it returns a copy of the argument, and with any other argument it attempts to convert the given object to a float. When used for conversions a string argument can be given, either using simple decimal notation or using exponential notation. It is possible that NaN ("not a number") or "infinity" may be produced by a calculation involving floats—unfortunately the behavior is not consistent across implementations and may differ depending on the system's underlying math library.

Here is a simple function for comparing floats for equality to the limit of the machine's accuracy:

Syntax	Description
`math.acos(x)`	Returns the arc cosine of `x` in radians
`math.acosh(x)`	Returns the arc hyperbolic cosine of `x` in radians
`math.asin(x)`	Returns the arc sine of `x` in radians
`math.asinh(x)`	Returns the arc hyperbolic sine of `x` in radians
`math.atan(x)`	Returns the arc tangent of `x` in radians
`math.atan2(y, x)`	Returns the arc tangent of `y / x` in radians
`math.atanh(x)`	Returns the arc hyperbolic tangent of `x` in radians
`math.ceil(x)`	Returns $\lceil x \rceil$, i.e., the smallest integer greater than or equal to `x` as an `int`; e.g., `math.ceil(5.4) == 6`
`math.copysign(x,y)`	Returns `x` with `y`'s sign
`math.cos(x)`	Returns the cosine of `x` in radians
`math.cosh(x)`	Returns the hyperbolic cosine of `x` in radians
`math.degrees(r)`	Converts `float r` from radians to degrees
`math.e`	The constant e; approximately 2.718 281 828 459 045 1
`math.exp(x)`	Returns e^x, i.e., `math.e ** x`
`math.fabs(x)`	Returns $\lvert x \rvert$, i.e., the absolute value of `x` as a `float`
`math.factorial(x)`	Returns $x!$
`math.floor(x)`	Returns $\lfloor x \rfloor$, i.e., the largest integer less than or equal to `x` as an `int`; e.g., `math.floor(5.4) == 5`

attributes; sys.float_info.epsilon is effectively the smallest difference that the machine can distinguish between two floating-point numbers. On one of the

author's 32-bit machines it is just over 0.0000000000000002. (Epsilon is the traditional name for this number.) Python floats normally provide reliable accuracy for up to 17 significant digits.

If you type sys.float_info into IDLE, all its attributes will be displayed; these include the minimum and maximum floating-point numbers the machine can represent. And typing help(sys.float_info) will print some information about the sys.float_info object.

Floating-point numbers can be converted to integers using the int() function which returns the whole part and throws away the fractional part, or using round() which accounts for the fractional part, or using math.floor() or math.ceil() which convert down to or up to the nearest integer. The float.is_integer() method returns True if a floating-point number's fractional part is 0, and a float's fractional representation can be obtained using the float.as_integer_ratio() method. For example, given x = 2.75, the call x.as_integer_ratio() returns (11, 4). Integers can be converted to floatingpoint numbers using float().

Floating-point numbers can also be represented as strings in hexadecimal format using the float.hex() method. Such strings can be converted back to floating-point numbers using the float.fromhex() method. For example:

```
s = 14.25.hex() # str s == '0x1.c800000000000p+3'
f = float.fromhex(s) # float f == 14.25
t = f.hex() # str t == '0x1.c800000000000p+3'
```

The exponent is indicated using p ("power") rather than e since e is a valid hexadecimal digit.

In addition to the built-in floating-point functionality, the math module provides many more functions that operate on floats, as shown in Tables 2.5 and 2.6.

Here are some code snippets that show how to make use of the module's functionality:

```
>>> import math
>>> math.pi * (5 ** 2) # Python 3.1 outputs: 78.53981633974483
78.539816339744831
>>> math.hypot(5, 12)
13.0
>>> math.modf(13.732) # Python 3.1 outputs: (0.7319999999999993, 13.0)
(0.73199999999999932, 13.0)
```

The math.hypot() function calculates the distance from the origin to the point (*x*, *y*) and produces the same result as math.sqrt((x ** 2) + (y ** 2)).

The math module is very dependent on the underlying math library that Python was compiled against. This means that some error conditions and boundary cases may behave differently on different platforms.

Complex Numbers:

The complex data type is an immutable type that holds a pair of floats, one representing the real part and the other the imaginary part of a complex number. Literal complex numbers are written with the real and imaginary parts joined by a + or - sign, and with the imaginary part followed by a j. Here are some examples: 3.5+2j, 0.5j, 4+0j, -1-3.7j. Notice that if the real part is 0, we can omit it entirely.

The separate parts of a complex are available as attributes real and imag.

For example:

```
>>> z = -89.5+2.125j
>>> z.real, z.imag
(-89.5, 2.125)
```

Except for //, %, divmod(), and the three-argument pow(), all the numeric operators and functions in Table 2.2 (55 ➤) can be used with complex numbers, and so can the augmented assignment versions. In addition, complex numbers have a method, conjugate(), which changes the sign of the imaginary part.

For example:

```
>>> z.conjugate()
(-89.5-2.125j)
>>> 3-4j.conjugate()
(3+4j)
```

Notice that here we have called a method on a literal complex number. In general, Python allows us to call methods or access attributes on any literal, as long as the literal's data type provides the called method or the attribute—however, this does not apply to special methods, since these always have corresponding operators such as + that should be used instead. For example, 4j.real produces 0.0, 4j.imag produces 4.0, and 4j + 3+2j produces 3+6j.
The complex data type can be called as a function—with no arguments it returns 0j, with a complex argument it returns a copy of the argument, and with any other argument it attempts to convert the given object to a complex. When used for conversions complex() accepts either a single string argument, or one or two floats. If just one float is given, the imaginary part is taken to be 0j.
The functions in the math module do not work with complex numbers. This is a deliberate design decision that ensures that users of the math module get exceptions rather than silently getting complex numbers in some situations. Users of complex numbers can import the cmath module, which provides complex number versions of most of the trigonometric and logarithmic functions that are in the math module, plus some complex number-specific functions such as cmath.phase(), cmath.polar(), and cmath.rect(), and also the cmath.pi and cmath.e constants which hold the same float values as their math module counterparts.

Decimal Numbers

In many applications the numerical inaccuracies that can occur when using floats don't matter, and in any case are far outweighed by the speed of calculation that floats offer. But in some cases we prefer the opposite trade-off, and want complete accuracy, even at the cost of speed. The decimal module provides immutable Decimal numbers that are as accurate as we specify. Calculations involving Decimals are slower than those involving floats, but whether this is noticeable will depend on the application.
To create a Decimal we must import the decimal module.
For example:

```
>>> import decimal
>>> a = decimal.Decimal(9876)
>>> b = decimal.Decimal("54321.012345678987654321")
>>> a + b
Decimal('64197.012345678987654321')
```

Decimal numbers are created using the decimal.Decimal() function. This function can take an integer or a string argument—but not a float, since floats are held inexactly whereas decimals are represented exactly. If a string is used it can use simple decimal notation or exponential notation. In addition to providing accuracy, the exact representation of decimal.Decimals means that they can be reliably compared for equality.
From Python 3.1 it is possible to convert floats to decimals using the deci- mal.Decimal.from_float() function. This function takes a float as argument
and returns the decimal.Decimal that is closest to the number the float approximates.
All the numeric operators and functions listed in Table 2.2 (55 ➤), including the augmented assignment versions, can be used with decimal.Decimals, but with a couple of caveats. If the ** operator has a decimal.Decimal left-hand operand, its right-hand operand must be an integer. Similarly, if the pow() function's first argument is a decimal.Decimal, then its second and optional third arguments must be integers.
The math and cmath modules are not suitable for use with decimal.Decimals, but some of the functions provided by the math module are provided as decimal. Decimal methods. For example, to calculate *ex* where x is a float, we write math.exp(x), but where x is a decimal.Decimal, we write x.exp(). From the discussion in Piece #3 (20 ➤), we can see that x.exp() is, in effect, syntactic sugar for decimal.Decimal.exp(x).
The decimal.Decimal data type also provides ln() which calculates the natural

(base *e*) logarithm (just like math.log() with one argument), log10(), and sqrt(), along with many other methods specific to the decimal.Decimal data type.
Numbers of type decimal.Decimal work within the scope of a *context*; the context is a collection of settings that affect how decimal. Decimals behave. The context specifies the precision that should be used (the default is 28 decimal places), the rounding technique, and some other details.
In some situations the difference in accuracy between floats and decimal. Decimals becomes obvious:

```
>>> 23 / 1.05
21.904761904761905
>>> print(23 / 1.05)
21.9047619048
>>> print(decimal.Decimal(23) / decimal.Decimal("1.05"))
21.904761904761904761904761904761
>>> decimal.Decimal(23) / decimal.Decimal("1.05")
Decimal('21.904761904761904761904761904761')
```

Although the division using decimal. Decimals is more accurate than the one involving floats, in this case (on a 32-bit machine) the difference only shows up in the fifteenth decimal place. In many situations this is insignificant—for example, in this book, all the examples that need floating-point numbers use floats.
One other point to note is that the last two of the preceding examples reveal for the first time that printing an object involves some behind-the-scenes formatting.
When we call print() on the result of decimal.Decimal(23) / decimal. Decimal("1.05") the bare number is printed—this output is in *string form*.
If we simply enter the expression we get a decimal.Decimal output—this output is in *representational form*. All Python objects have two output forms. String form is designed to be human-readable. Representational form is designed to produce output that if fed to a Python interpreter would (when possible) reproduce the represented object.

Compound Data Types:

compound data types provide ways to organize and manage data values of any data type. There are no constants for compound data types; the actual data values contained in a variable of a compound data type will always be of one or more of the basic data types.
IDL provides the following compound data types:

Lists

Lists are ordered collections of values, where the individual list values can be of different data types. Lists can grow and shrink dynamically, and the data type of a given list element can be changed. For more information, refer to LIST

Hashes

Hashes are unordered collections of key-value pairs, where the individual keys and values can be of different data types. Hashes can grow and shrink dynamically, and the data type of a given hash element can be changed. For more information, refer to HASH.

Structures

Structures consist of one or more tag-value pairs, where the value associated with a given tag can be of any data type. Once a structure has been defined, no tags can be added or removed, and the data type of tag values cannot be changed. Structures are discussed in Structure References.

Pointers

Pointers are references to dynamically-allocated *heap variables* that can contain values of any data type. Heap variables are available at all IDL program levels; as a result, pointers can be used to easily make data available to multiple routines without copying.

Object References

Object references are references to special heap variables that contain IDL object structures.

Dictionary:

- A dict is an unordered collection of zero or more key–value pairs whose keys are object references that refer to hashable objects,and whose values are object references referring to objects of any

type. Dictionaries are mutable, so we can easily add or remove items, but since they are unordered they have no notion of index position and so cannot be sliced or strided.

- Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.
- Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.
- It is best to think of a dictionary as a set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: {}. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.
- The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with del. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Accessing Values in a dictionary :

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example.

```
#!/usr/bin/python3
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
print ("dict['Name']: ", dict['Name'])
print ("dict['Age']: ", dict['Age'])
```

When the above code is executed, it produces the following result-

```
dict['Name']: Zara
```

Sets

- A set is an unordered collection of zero or more object references that refer to hashable objects. Sets are mutable, so we can easily add or remove items, but since they are unordered they have no notion of index position and so cannot be sliced or strided. Figure 3.3 illustrates the set created by the following code snippet:

S = {7, "veil", 0, -29, ("x", 11), "sun", frozenset({8, 4, 7}), 913}

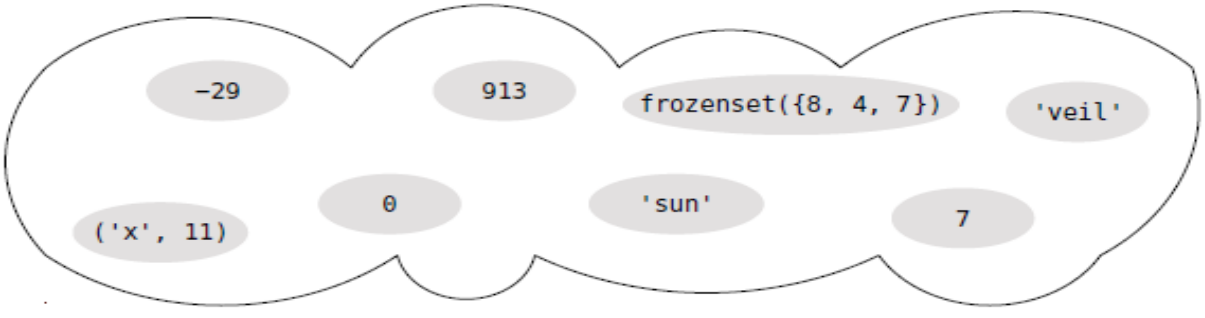


Figure 3.3 *A set is an unordered collection of unique items.*

- The set data type can be called as a function, set()—with no Shallow and deep copying arguments it returns an empty set, with a set argument it returns a shallow copy of the argument, and with any other argument it attempts to convert the given object to a set. It does not accept more than one argument. Nonempty sets can also be created without using the set() function, but the empty set must be created using set(), *not* using empty braces
- A set of one or more items can be created by using a comma-separated sequence of items inside braces. Another way of creating sets is to use a set comprehension—a topic we will cover later in this subsection.

Mapping:

- A *mapping* type is one that supports the membership operator (in) and the size function (len()), and is iterable. Mappings are collections of key–value items and provide methods for accessing items and their keys and values.
- When iterated, unordered mapping types provide their items in an arbitrary order. Python 3.0 provides two unordered mapping types, the built-in dict type and the standard library's collections. defaultdict type. A new, ordered mapping type, collections. OrderedDict,was introduced with Python 3.1; this is a dictionary that has the same methods and properties (i.e., the same API) as the built-in dict, but stores its items in *insertion* order.
- We will use the term *dictionary* to refer to any of these types when the difference doesn't matter.
- Hash- Only hashable objects may be used as dictionary keys, so immutable able objects data types such as float, frozenset, int, str, and tuple can be used as dictionary keys, but mutable types such as dict, list, and set cannot. On the other hand, each key's associated value can be an object reference referring to an object of any type, including numbers, strings, lists, sets, dictionaries, functions, and so on.
- Dictionary types can be compared using the standard equality comparison operators (== and !=), with the comparisons being applied item by item (and recursively for nested items such as tuples or dictionaries inside dictionaries).Comparisons using the other comparison operators (<, <=, >=, >) are not supported since they don't make sense for unordered collections such as dictionaries.

Variables :

What is a Variable in Python?

- A Python variable is a reserved memory location to store values. In other words, a variable in a python program gives data to the computer for processing.
- Every value in Python has a datatype. Different data types in Python are Numbers, List, Tuple, Strings, Dictionary, etc. Variables can be declared by any name or even alphabets like a, aa, abc, etc
- **Variable Naming Rules in Python**
- Variable name should start with letter(a-zA-Z) or underscore (_). Valid : age , _age , Age Invalid : 1age
- In variable name, no special characters allowed other than underscore (_). Valid : age_ , _age Invalid : age_*
- 3.Variables are case sensitive. age and Age are different, since variable names are case sensitive. 4.Variable name can have numbers but not at the beginning. Example: Age1
- 5.Variable name should not be a Python keyword.

Keywords are also called as reserved words. Example pass, break, continue.. etc are reserved for special meaning in Python. So, we should not declare keyword as a variable name.

How to Declare and use a Variable

Let see an example.
We will declare variable "a" and print it.

```
a=100
 print (a)
```

Re-declare a Variable

You can re-declare the variable even after you have declared it once.

```
a=100
print(a)
a='AECS Jaduguda'
print(a)
```

Delete a variable

You can also delete variable using the command del "variable name". The below table displays the list of available assignment operators in Python language.

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume).

Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

Global Variables

Variables that are created outside of a function (as in all of the examples above) are known as global variables.

Global variables can be used by everyone, both inside of functions and outside.

Example

Create a variable outside of a function, and use it inside the function

```
x = "awesome"

def myfunc():
  print("Python is " + x)

myfunc()
```

If you create a variable with the same name inside a function, this variable will be local, and can only be used inside the function. The global variable with the same name will remain as it was, global and with the original value.

Example

Create a variable inside a function, with the same name as the global variable

```
x = "awesome"

def myfunc():
  x = "fantastic"
  print("Python is " + x)

myfunc()

print("Python is " + x)
```

The global Keyword

Normally, when you create a variable inside a function, that variable is local, and can only be used inside that function.

To create a global variable inside a function, you can use the global keyword.

Example

If you use the global keyword, the variable belongs to the global scope:

```
def myfunc():
  global x
  x = "fantastic"

myfunc()

print("Python is " + x)
```
