Python Basics

Python: Python is an interpreter, high-level and general-purpose programming language.

1 Statements and Syntax

Some rules and certain symbols are used with regard to statements in Python:

- Hash mark (#) indicates Python comments
- NEWLINE (\n) is the standard line separator (one statement per line)
- Backslash (\ ) continues a line
- Semicolon (;) joins two statements on a line
- Colon (:) separates a header line from its suite
- Statements (code blocks) grouped as suites
- Suites delimited via indentation
- Python files organized as modules

1.1 Comments (#)

Python comment statements begin with the pound sign or hash symbol (#). A comment can begin anywhere on a line. All characters following the # to the end of the line are ignored by the interpreter.

1.2 Continuation (\ )

Python statements are, in general, delimited by NEWLINEs, meaning one statement per line. Single statements can be broken up into multiple lines by use of the backslash. The backslash symbol (\ ) can be placed before a NEWLINE to continue the current statement onto the next line.

```
# check conditions
if (weather_is_hot == 1) and \
(shark_warnings == 0):
send_goto_beach_mesg_to_pager()
```

1.3 Multiple Statement Groups as Suites (:)

Groups of individual statements making up a single code block are called “suites” in Python. Compound or complex statements, such as if, while, def, and class, are those that require a header line and a suite. Header lines begin the statement (with the keyword) and terminate with a colon (:) and are followed by one or more lines that make up the suite.

3.1.4 Suites Delimited via Indentation

Python employs indentation as a means of delimiting blocks of code. Code at inner levels is indented via spaces or tabs. Indentation requires exact indentation; in other words, all the lines of code in a suite must be indented at the exact same level. Indented lines starting at different positions or column numbers are not allowed; each line would be considered part of another suite and would more than likely result in syntax errors.

1.5 Multiple Statements on a Single Line (;)

The semicolon (;) allows multiple statements on a single line given that neither statement starts a new code block. Here is a sample snip using the semicolon:

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

1.6 Modules

Each Python script is considered a module. Modules have a physical presence as disk files.

2 Variable Assignments

Assignment Operator

The equal sign (=) is the main Python assignment operator.

```
anInt = -12
aString = 'cart'
aFloat = -3.1415 * (5.0 ** 2)
anotherString = 'shop' + 'ping'
aList = [3.14e10, '2nd elmt of a list', 8.82-4.371j]
```

Augmented Assignment

Beginning in Python 2.0, the equal sign can be combined with an arithmetic operation and the resulting value reassigned to the existing variable. Known as augmented assignment, statements such as . . .

x = x + 1

. . . can now be written as . . .

x += 1

Multiple Assignment

The process of assigning a single object to multiple variables is known as multiple assignment.

```
>>> x = y = z = 1
>>> x
1
>>> y
1
>>> z
1
```

"Multuple" Assignment

The process of assigning a multiple objects to multiple variables is known as multuple assignment.

```
>>> x, y, z = 1, 2, 'a string'
>>> x
1
>>> y
2
>>> z
'a string'
```

3 Identifiers

Identifiers are the set of valid strings that are allowed as names in a computer language. We segregate out those that are keywords, names that form a construct of the language. Such identifiers are reserved words that may not be used for any other purpose.

Python also has an additional set of identifiers known as built-ins, and although they are not reserved words.

3.1 Valid Python Identifiers

The rules for Python identifier strings are like most other high-level programming languages that come from the C world:

- First character must be a letter or underscore (_)
- Any additional characters can be alphanumeric or underscore
- Case-sensitive

3.2 Keywords

Table 3.1 Python Keywords[a]

and	as[b]	assert[c]	break
class	continue	def	del
elif	else	except	exec
finally	for	from	global
if	import	in	is
lambda	not	or	pass
print	raise	return	try
while	with[b]	yield[d]	None[e]

3.4 Basic Style Guidelines

3.4.1 Module Structure and Layout

Modules are simply physical ways of logically organizing all your Python code. Within each file, you should set up a consistent and easy-to-read structure. One such layout is the following:

```
# (1) startup line (Unix)
# (2) module documentation
# (3) module imports
# (4) variable declarations
# (5) class declarations
# (6) function declarations
# (7) "main" body
```

(1) Startup line

Generally used only in Unix environments, the startup line allows for script execution by name only .

(2) Module documentation

Summary of a module's functionality and significant global variables; accessible externally as module.__doc__.

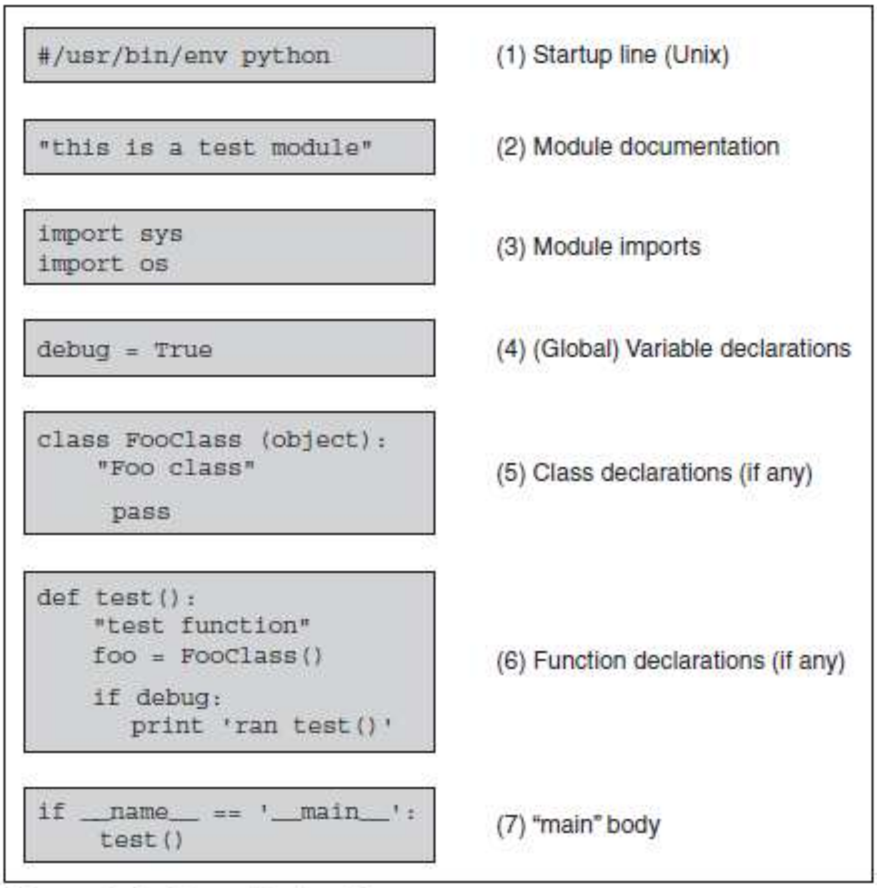


Figure 3–1 Typical Python file structure

(3) Module imports

Import all the modules necessary for all the code in current module; modules are imported once (when this module is loaded); imports within functions are not invoked until those functions are called.

(4) Variable declarations

Declare here (global) variables that are used by multiple functions in this module. We favor the use of local variables over globals, for good programming style mostly, and to a lesser extent, for improved performance and less memory usage.

(5) Class declarations

Any classes should be declared here. A class is defined when this module is imported and the class statement executed.

Documentation variable is class.__doc__.

(6) Function declarations

Functions that are declared here are accessible externally as module.function(); function is defined when this module is imported and the def statement executed. Documentation variable is function.__doc__.

(7) “main” body

All code at this level is executed, whether this module is imported or started as a script; generally does not include much functional code, but rather gives direction depending on mode of execution.

3.5 Memory Management

Details about variables and memory management are including:

- Variables not declared ahead of time
- Variable types not declared
- No memory management on programmers’ part
- Variable names can be “recycled”
- del statement allows for explicit “deallocation”

3.5.1 Variable Declarations (or Lack Thereof)

In most compiled languages, variables must be declared before they are used. In Python, there are no explicit variable declarations. Variables are “declared” on first assignment.

Like most languages, however, variables cannot be accessed until they are (created and) assigned:

```
>>> a
Traceback (innermost last):
File "<stdin>", line 1, in ?
NameError: a
```

Once a variable has been assigned, you can access it by using its name:

```
>>> x = 4
>>> y = 'this is a string'
>>> x
4
>>> y
'this is a string'
```

3.5.2 Dynamic Typing

In Python, the type and memory space for an object are determined and allocated at runtime. Although code is byte-compiled, Python is still an interpreted language. On creation—that is, on assignment—the interpreter creates an object whose type is dictated by the syntax that is used for the operand on the right-hand side of an assignment. After the object is created, a reference to that object is assigned to the variable on the left-hand side of the assignment.

3.5.3 Memory Allocation

Python simplifies application writing because the complexities of memory management have been pushed down to the interpreter.

3.5.4 Reference Counting

To keep track of objects in memory, Python uses the simple technique of reference counting. This means that internally, Python keeps track of all objects in use and how many interested parties there are for any particular object. An internal tracking variable, called a reference counter, keeps track of how many references are being made to each object, called a refcount.

When an object is created, a reference is made to that object, and when it is no longer needed, i.e., when an object's refcount goes down to zero, it isngarbage-collected.

Incrementing the Reference Count

The refcount for an object is initially set to 1 when an object is created and (its reference) assigned.

New references to objects, also called aliases, occur when additional variables are assigned to the same object, passed as arguments to invoke other bodies of code such as functions, methods, or class instantiation, or assigned as members of a sequence or mapping.

Decrementing the Reference Count

When references to an object "go away," the refcount is decreased. The most obvious case is when a reference goes out of scope. This occurs most often when the function in which a reference is made completes. The local (automatic) variable is gone, and an object's reference counter is decremented.

del Statement

The del statement removes a single reference to an object. Its syntax is:

del obj1[, obj2[,... objN]]

3.5.5 Garbage Collection

The garbage collector is a separate piece of code that looks for objects with reference counts of zero. It is also responsible to check for objects with a reference count greater than zero that need to be deallocated.

Certain situations lead to cycles.

1. Python Objects

Python uses the object model abstraction for data storage. Any construct that contains any type of value is an object.

All Python objects have the following three characteristics: an identity, a type, and a value.

IDENTITY: Unique identifier that differentiates an object from all others. Any object's identifier can be obtained using the id() built-in function (BIF).

TYPE: An object's type indicates what kind of values an object can hold, what operations can be applied to such objects, and what behavioral rules these objects are subject to. We can use the type() BIF to reveal the type of a Python object.

VALUE: Data item that is represented by an object.

1.1. Object Attributes

Certain Python objects have attributes, data values or executable code such as methods, associated with them. Attributes are accessed in the dotted attribute notation, which includes the name of the associated object.

2. Standard Types

- Numbers
    - Integer
    - Boolean
    - Floating point real number
    - Complex number
- String
- List
- Tuple
- Dictionary

3. Other Built-in Types

- Type
- Null object (None)
- File
- Set/Frozenset
- Function/Method
- Module
- Class

3.1 Type Objects and the type Type Object

The amount of information necessary to describe a type cannot fit into a single string; therefore types cannot simply be strings, nor should this information be stored with the data, so we are defined types as objects. We can find out the type of an object by calling type() with that object:

```
>>> type(42)
<type 'int'>
```

Here <type 'int'> is a int type object.

```
>>> type(type(42))
<type 'type'>
```

The type of all type objects is type. The type type object is also the mother of all types and is the default metaclass for all standard Python classes.

3.2. None, Python's Null Object

Python has a special type known as the Null object or NoneType. It has only one value, None. The type of None is NoneType. It does not have any operators or BIFs. None has no (useful) attributes and always evaluates to having a Boolean False value.

The following are defined as having false values in Python:

- None
- False (Boolean)
- Any numeric zero:
- 0 (integer)
- 0.0 (float)
- 0L (long integer)
- 0.0+0.0j (complex)
- "" (empty string)
- [] (empty list)
- () (empty tuple)
- {} (empty dictionary)

4. Internal Types

- Code
- Frame
- Traceback
- Slice
- Ellipsis
- Xrange

4.1. Code Objects

Code objects are executable pieces of Python source that are byte-compiled, usually as return values from calling the compile () BIF. Such objects are appropriate for execution by either **exec** or by the eval() BIF. Code objects themselves do not contain any information regarding their execution environment, but they are at the heart of every user-defined function, all of which *do* contain some execution context.

4.2. Frame Objects

These are objects representing execution stack frames in Python. Frame objects contain all the information the Python interpreter needs to know during a runtime execution environment. Some of its attributes include a link to the previous stack frame, the code object (see above) that is being executed, dictionaries for the local and global namespaces, and the current instruction. Each function call results in a new frame object, and for each frame object, a C stack frame is created as well.

4.3. Traceback Objects

When you make an error in Python, an exception is raised. If exceptions are not caught or "handled," the interpreter exits with some diagnostic information similar to the output shown below:

Traceback (innermost last):

File "<stdin>", line N?, in ???

ErrorName: error reason

The traceback object is just a data item that holds the stack trace information for an exception and is created when an exception occurs. If a handler is provided for an exception, this handler is given access to the traceback object.

4.4. Slice Objects

Slice objects are created using the Python extended slice syntax. This extended syntax allows for different types of indexing. These various types of indexing include *stride indexing*, multi-dimensional indexing, and indexing using the Ellipsis type. The syntax for multi-dimensional indexing is *sequence* [*start1: end1*, *start2: end2]*, or using the ellipsis, sequence [..., start1: end1]. Slice objects can also be generated by the slice () BIF.

Stride indexing for sequence types allows for a third slice element that allows for "step"-like access with a syntax of *sequence[starting_index : ending_index : stride]*.

An example of stride indexing:

```
>>> foostr = 'abcde'
>>> foostr[::-1]
'edcba'
>>> foostr[::-2]
'eca'
```

4.5. Ellipsis Objects

Ellipsis objects are used in extended slice notations. These objects are used to represent the actual ellipses in the slice syntax (...). Like the Null object None, ellipsis objects also have a single name, Ellipsis, and have a Boolean True value at all times.

4.6. XRange Objects

XRange objects are created by the BIF xrange(), a sibling of the range() BIF, and used when memory is limited and when range() generates an unusually large data set.

5. Standard Type Operators

5.1. Object Value Comparison

Comparison operators are used to determine equality of two data values between members of the same type. These comparison operators are supported for all built-in types. Comparisons yield Boolean True or False values, based on the validity of the comparison expression.

Standard Type Value Comparison Operators

Operator	Function
expr1 < expr2	expr1 is less than expr2
expr1 > expr2	expr1 is greater than expr2
expr1 <= expr2	expr1 is less than or equal to expr2

expr1 >= expr2	expr1 is greater than or equal to expr2
expr1 == expr2	expr1 is equal to expr2
expr1 != expr2	expr1 is not equal to expr2 (C-style)
expr1 <> expr2	expr1 is not equal to expr2 (ABC/Pascal-style)

```
>>> 2 == 2
True
>>> 2.46 <= 8.33
True
>>> 5+4j >= 2-3j
True
>>> 'abc' == 'xyz'
False
>>> 'abc' > 'xyz'
False
>>> 'abc' < 'xyz'
True
>>> [3, 'abc'] == ['abc', 3]
False
>>> [3, 'abc'] == [3, 'abc']
True
```

Multiple comparisons can be made on the same line, evaluated in left-to-right order:

```
>>> 3 < 4 < 7 # same as ( 3 < 4 ) and ( 4 < 7 )
True
>>> 4 > 3 == 3 # same as ( 4 > 3 ) and ( 3 == 3 )
True
>>> 4 < 3 < 5 != 2 < 7
False
```

5.2. Object Identity Comparison

In addition to value comparisons, Python also supports the notion of directly comparing objects themselves. Objects can be assigned to other variables (by reference). Because each variable points to the same (shared) data object, any change effected through one variable will change the object and hence be reflected through all references to the same object.

Python provides is and is not operators to test if a pair of variables do indeed refer to the same object. Performing a check such as a is b is an equivalent expression to id(a) == id(b)

The object identity comparison operators all share the same precedence level.

Standard Type Object Identity Comparison Operators

Operator	Function
obj1 is obj2	obj1 is the same object as obj2
obj1 is not obj2	obj1 is not the same object as obj2

```
>>> a = [ 5, 'hat', -9.3]
>>> b = a
>>> a is b
True
```

```
>>> a is not b
False
```

5.3. Boolean

Expressions may be linked together or negated using the Boolean logical operators and, or, and not, all of which are Python keywords. These Boolean operations are in highest-to-lowest order of precedence in Table.

Table 4.3. Standard Type Boolean Operators

Operator	Function
not expr	Logical NOT of expr (negation)
expr1 and expr2	Logical AND of expr1 and expr2 (conjunction)
expr1 or expr2	Logical OR of expr1 and expr2 (disjunction)

```
>>> x, y = 3.1415926536, -1024
>>> (x < 5.0) or (y > 2.718281828)
True
>>> (x < 5.0) and (y > 2.718281828)
False
>>> not (x is y)
True
```

6. Standard Type Built-in Functions

Python provides some BIFs that can be applied to all the basic object types: cmp(),repr(),str(),type(), and the single reverse or back quotes(``) operator, which is functionally equivalent to repr().

Standard Type Built-in Functions

Function	Operation
cmp(obj1, obj2)	Compares obj1 and obj2, returns integer i where: i < 0 if obj1 < obj2 i > 0 if obj1 > obj2 i == 0 if obj1 == obj2
repr(obj) or `obj`	Returns evaluatable string representation of obj
str(obj)	Returns printable string representation of obj
type(obj)	Determines type of obj and return type object

6.1. type()

The syntax for type() is:

type(object)   type() takes an object and returns its type. The return value is a type object.

```
>>> type(4) # int type
<type 'int'>
>>> type('Hello World!') # string type
<type 'string'>
>>>
>>> type(type(4)) # type type
<type 'type'>
```

6.2. cmp()

The cmp() BIF comares two objects, say, obj1 and obj2, and returns a negative number (integer) if obj1 is less than obj2, a positive number if obj1 is greater than obj2, and zero if obj1 is equal to obj2. some samples of using the cmp() BIF with numbers and strings.

```
>>> a, b = -4, 12
>>> cmp(a,b)
-1
>>> cmp(b,a)
1
>>> b = -4
>>> cmp(a,b)
0
```

6.3. str() and repr() (and `` Operator)

The str() string and repr() representation BIFs or reverse quote operator (``) used either re-create an object through evaluation or obtain a human-readable view of the contents of objects, data values, object types, etc. To use these operations, a Python object is provided as an argument and some type of string representation of that object is returned. In the examples, some random Python types and convert them to their
string representations.

```
>>> str(4.53-2j)
'(4.53-2j)'
>>>
>>> str(1)
'1'
>>> str(2e10)
'20000000000.0'
>>>
>>> str([0, 5, 9, 9])
'[0, 5, 9, 9]'
>>>
>>> repr([0, 5, 9, 9])
'[0, 5, 9, 9]'
>>>
>>> `[0, 5, 9, 9]`
'[0, 5, 9, 9]'
```

Although all three are similar in nature and functionality, only repr() and `` do exactly the same thing, and using them will deliver the "official" string representation of an object that can be evaluated as a valid Python expression (using the eval() BIF). In contrast, str() has the job of delivering a "printable" string representation of an object, which may not necessarily be acceptable by eval(), but will look nice in a print statement. There is a limitation that while most return values from repr() can be evaluated, not all can:

```
>>> eval(`type(type))`)
File "<stdin>", line 1
eval(`type(type))`)
^
```

SyntaxError: invalid syntax
The repr( ) is Python-friendly while str() produces human-friendly output.

6.4. type() and isinstance()

Python provides a BIF type () returns the type for any Python object. Some examples of what type () returns when we give it various objects.

```
>>> type('')
<type 'str'>
>>> s = 'xyz'
>>> type(s)
<type 'str'>
>>> type(100)
<type 'int'>
>>> type(0+0j)
<type 'complex'>
>>> type(0.0)
<type 'float'>
>>> type([])
<type 'list'>
>>> type(())
<type 'tuple'>
>>> type({})
<type 'dict'>
>>> type(type)
<type 'type'>
>>> class Foo: pass # new-style class
...
>>> foo = Foo()
>>> class Bar(object): pass # new-style class
...
>>> bar = Bar()
>>>
>>> type(Foo)
<type 'classobj'>
>>> type(foo)
<type 'instance'>
>>> type(Bar)
<type 'type'>
>>> type(bar)
<class '__main__.Bar'>
```

In addition to type(), there is another useful BIF called is isinstance(). It is used to determine the type of an object.

Example

The function displayNumType() takes a numeric argument and uses the type() built-in to indicate its type (or "not a number," if that is the case).

```
def displayNumType(num):
    print num, 'is',
    if isinstance(num, (int, long, float, complex)):
        print 'a number of type:', type(num).__name__
    else:
    print 'not a number at all!!'

displayNumType(-69)
displayNumType(9999999999999999999999L)
displayNumType(98.6)
displayNumType(-5.2+1.9j)
displayNumType('xxx')
```

Running typechk.py, we get the following output:

```
-69 is a number of type: int
9999999999999999999999 is a number of type: long
98.6 is a number of type: float
(-5.2+1.9j) is a number of type: complex
xxx is not a number at all!!
```

8. Categorizing the Standard Types

Python's "basic built-in data object primitive types" are

- "Basic," indicating that these are the standard or core types that Python provides
- "Built-in," due to the fact that these types come by default in Python
- "Data," because they are used for general data storage
- "Object," because objects are the default abstraction for data and functionality
- "Primitive," because these types provide the lowest-level granularity of data storage
- "Types," because that's what they are: data types!

There are three different models help us to categorize the standard types.

8.1. Storage Model

The first way we can categorize the types is by how many objects can be stored in an object of this type. Python's types can hold either single or multiple values. A type which holds a single literal object is called atomic or scalar storage, and those which can hold multiple objects we will refer to as container storage. (Container objects are also referred to as composite or compound objects)

Storage Model Category	**Python Types That Fit Category**
Scalar/atom	Numbers (all numeric types), strings (all are literals)
Container	Lists, tuples, dictionaries

8.2. Update Model

Another way of categorizing the standard types is based on, "Once an object is created, can be changed, or can their values be updated". Mutable objects are those whose values can be changed, and immutable objects are those whose values cannot be changed.

Update Model Category	Python Types That Fit Category
Mutable	Lists, dictionaries
Immutable	Numbers, strings, tuples

8.3. Access Model

Access Model means how the stored data accessed. There are three categories under the access model: direct, sequence, and mapping. The different access models and which types fall into each respective category.

Table 4.8. Types Categorized by the Access Model

Access Model Category	Types That Fit Category
Direct	Numbers
Sequence	Strings, lists, tuples
Mapping	Dictionaries

Direct types indicate single-element, non-container types. All numeric types fit into this category.

Sequence types are those whose elements are sequentially accessible via index values starting at 0. Accessed items can be either single elements or in groups, better known as slices. Types that fall into this category include strings, lists, and tuples.

Mapping types are similar to the indexing properties of sequences, except instead of indexing on a sequential numeric offset, elements (values) are unordered and accessed with a key, thus making mapping types a set of hashed key-value pairs.

Table 4.9. Categorizing the Standard Types

Data Type	Storage Model	Update Model	Access Model
Numbers	**Scalar**	**Immutable**	**Direct**
Strings	**Scalar**	**Immutable**	**Sequence**
Lists	**Container**	**Mutable**	**Sequence**
Tuples	**Container**	**Immutable**	**Sequence**
Dictionaries	**Container**	**Mutable**	**Mapping**

9. Unsupported Types

char or byte

Python does not have a char or byte type to hold either single character or 8-bit integers. Use strings of length one for characters and integers for 8-bit numbers.

pointer

Since Python manages memory, there is no need to access addresses. The address of an object can be accessed by using the id() BIF.

int versus short versus long

Python's plain integers are the universal "standard" integer type, avoiding the need for three different integer types.

float versus double

Python does not support a single-precision floating point type because its benefits are outweighed by the overhead required to support two types of floating point types. For more accuracy and willing to give up a wider range of numbers, Python has a decimal floating point number too, but you have to import the decimal module to use the Decimal type. Floats are always estimations. Decimals are exact and arbitrary precision. Decimals make sense concerning things like money where the values are exact. Floats make sense for things that are estimates anyway, such as weights, lengths, and other measurements**.**

5.1. Introduction to Numbers

Numbers provide literal or scalar storage and direct access. A number is also an immutable type, meaning that changing or updating its value results in a newly allocated object.

Python has several numeric types: "plain" integers, long integers, Boolean, double-precision floating point real numbers, decimal floating point numbers, and complex numbers.

How to Create and Assign Numbers (Number Objects)

Creating numbers is as simple as assigning a value to a variable:

anInt = 1

aLong = -9999999999999999L

aFloat = 3.1415926535897932384626433832795

aComplex = 1.23+4.56J

How to Update Numbers

You can "update" an existing number by (re)assigning a variable to another number. The new value can be related to its previous value or to a completely different number altogether. *Update* means, you are not really changing the value of the original variable. Because numbers are immutable, you are making a new number and reassigning the reference.

anInt += 1

aFloat = 2.718281828

How to Remove Numbers

Under normal circumstances, you do not really "remove" a number; you just stop using it! If you really want to delete a reference to a number object, use the **del** statement. You can no longer use the variable name, once removed, unless you assign it to a new object;

otherwise, it will cause a NameError exception to occur.

del anInt

del aLong, aFloat, aComplex

5.2. Integers

Python has several types of integers. There is the Boolean type with two possible values. There are the regular or plain integers

5.2.1. Boolean

The Boolean type has two possible values, True and False.

5.2.2. Standard (Regular or Plain) Integers

Python's "plain" integers are the universal numeric type. It can represent any range of integers.

Integers are normally represented in base 10 decimal formats, but they can also be specified in base 8 or base 16 representation. Octal values have a "0" prefix, and hexadecimal values have either "0x" or "0X" prefixes.

5.3. Double Precision Floating Point Numbers

Floats in Python are implemented as C doubles; double precision floating point real numbers, values that can be represented in straightforward decimal or scientific notations. These 8-byte (64-bit) values conform to the IEEE 754 definition (52M/11E/1S) where 52 bits are allocated to the mantissa, 11 bits to the exponent (this gives you about ± 10308.25 in range), and the final bit to the sign.

Floating point values are denoted by a decimal point (.) in the appropriate place and an optional "e" suffix representing scientific notation. We can use either lowercase (e) or uppercase (E). Positive (+) or negative (-) signs between the "e" and the exponent indicate the sign of the exponent. Here are some floating point values:

0.0 -5.555567119 96e3 * 1.0

4.3e25 9.384e-23 -2.172818

5.4. Complex Numbers

A complex number is any ordered pair of floating point real numbers (x, y) denoted by x + yj where x is the real part and y is the imaginary part of a complex number. Complex numbers are used a lot in everyday math, engineering, electronics, etc.

Here are some facts about Python's support of complex numbers:

- Imaginary numbers by themselves are not supported in Python
- Complex numbers are made up of real and imaginary parts
- Syntax for a complex number: *real+imagj*
- Both real and imaginary components are floating point values
- Imaginary part is suffixed with letter "J" lowercase (j) or uppercase (J)

The following are examples of complex numbers:

64.375+1j 4.23-8.5j 0.23-8.55j 1.23e-045+6.7e+089j

6.23+1.5j -1.23-875J 0+1j 9.80665-8.31441J -.0224+0j

5.4.1. Complex Number Built-in Attributes

Complex numbers are one example of objects with data attributes. The data attributes are the real and imaginary components of the complex number object they belong to. Complex numbers also have a method attribute that can be invoked, returning the complex conjugate of the object.

```
>>> aComplex = -8.333-1.47j
>>> aComplex
(-8.333-1.47j)
>>> aComplex.real
-8.333
>>> aComplex.imag
-1.47
>>> aComplex.conjugate()
(-8.333+1.47j)
```

Table 5.1 describes the attributes of complex numbers.

Table 5.1. Complex Number Attributes

Attribute	***Description***
num.real	Real component of complex number *num*
num.imag	Imaginary component of complex number *num*
num.conjugate()	Returns complex conjugate of *num*

5.6. Built-in and Factory Functions

5.6.1. Standard Type Functions

the cmp(), str(), and type() are built-in functions that apply for all standard types. For numbers, these functions will compare two numbers, convert numbers into strings, and tell you a number's type, respectively. Here are some examples of using these functions:

```
>>> cmp(-6, 2)
-1
>>> cmp(-4.333333, -2.718281828)
-1
>>> cmp(0xFF, 255)
0
>>> str(0xFF)
'255'
>>> str(55.3e2)
'5530.0'
>>> type(0xFF)
<type 'int'>
>>> type(2-1j)
<type 'complex'>
```

5.6.2. Numeric Type Functions

Python currently supports different sets of built-in functions for numeric types. Some convert from one numeric type to another while others are more operational, performing some type of calculation on their numeric arguments.

Conversion Factory Functions

The int(), float(), and complex() functions are used to convert from any numeric type to another. bool() was used to normalize Boolean values to their integer equivalents of one and zero for true and false values.

The following are some examples of using these functions:

```
>>> int(4.25555)
4
>>> float(4)
4.0
>>> complex(4)
(4+0j)
```

Table 5.5 summarizes the numeric type factory functions.

Table 5.5. Numeric Type Factory Functions[a]

Class (Factory Function)	***Operation***
bool(obj)	Returns the Boolean value of *obj*.
int*(obj, base=10)*	Returns integer representation of string or number *obj*;
float*(obj)*	Returns floating point representation of string or number *obj*;
complex*(str)* **or** *complex(real, imag=0.0)*	Returns complex number representation of str, or builds one given *real* (and perhaps *imag*inary) component(s)

Operational

Python has five operational built-in functions for numeric types: abs(), coerce(), divmod(), pow(), and round().

abs() returns the absolute value of the given argument. If the argument is a complex number, then math.sqrt(*num* .real2 + *num*.imag2) is returned. Here are some examples of using the abs() built-in function:

```
>>> abs(-1)
1
>>> abs(1.2-2.1j)
2.41867732449
```

The coerce() function is a numeric type conversion function, does not convert to a specific type and acts more like an operator. coerce() just returns a tuple containing the converted pair of numbers. Here are some examples:

```
>>> coerce(1, 2)
(1, 2)
>>>
>>> coerce(1.3, 134)
(1.3, 134.0)
>>>
>>> coerce(1j, 134)
(1j, (134+0j))
>>>
>>> coerce(1.23-41j, 134)
((1.23-41j), (134+0j))
```

The divmod() built-in function combines division and modulus operations into a single function call that returns the pair as a tuple. The values returned are the same as those given for the classic division and modulus operators for integer types. For floats, the quotient returned is math.
floor(*num1/num2*) and for complex numbers, the quotient is math.floor((*num1/num2*).real).

```
>>> divmod(10,3)
(3, 1)
>>> divmod(3,10)
(0, 3)
>>> divmod(10,2.5)
(4.0, 0.0)
>>> divmod(2.5,10)
(0.0, 2.5)
```

Both pow() and the double star (**) operator perform exponentiation; however, there are differences other than the fact that one is an operator and the other is a built-in function.

```
>>> pow(2,5)
32
>>> pow(5,2)
25
```

The round() built-in function has a syntax of round(*flt,ndig*=0). It normally rounds a floating point number to the nearest integral number and returns that result (still) as a float. When the optional *ndig* option is given, round() will round the argument to the specific number of decimal places.

```
>>> round(3)
3.0
>>> round(3.45)
```

```
3.0
>>> round(3.4999999)
3.0
>>> round(3.4999999, 1)
3.5
```

5.6.3. Integer-Only Functions

In addition to the built-in functions for all numeric types, Python supports a few that are specific only to integers (plain). These functions fall into two categories, base presentation with hex() and oct(), and ASCII conversion featuring chr() and ord().

Base Representation

Python integers automatically support octal and hexadecimal representations in addition to the decimal standard. Also, Python has two built-in functions that return string representations of an integer's octal or hexadecimal equivalent. These are the oct() and hex() built-in functions, respectively. They both take an integer (in any representation) object and return a string with the corresponding value. The following are some examples of their usage:

```
>>> hex(255)
'0xff'
>>> hex(65535*2)
'0x1fffe'
```

ASCII Conversion

chr() takes a single-byte integer value and returns a one-character string with the equivalent ASCII character. ord() does the opposite, taking a single ASCII character in the form of a string of length one and returns the corresponding ASCII value as an integer:

```
>>> ord('a')
97
>>> ord('A')
65
>>> ord('0')
48
>>> chr(97)
'a'
>>> chr(65L)
'A'
>>> chr(48)
'0'
```

5.8. Related Modules

There are a number of modules in the Python standard library that add on to the functionality of the operators and built-in functions for numeric types.

Table 5.8. Numeric Type Related Modules

Module	***Contents***
decimal	Decimal floating point class Decimal
array	Efficient arrays of numeric values (characters, ints, floats, etc.)
math/cmath	most functions available in math are implemented for complex numbers in the cmath module

operator	Numeric operators available as function calls, i.e., operator.sub*(m, n)* is equivalent to the difference *(m - n)* for numbers *m* and *n*
random	Various pseudo-random number generators (obsoletes rand and wHRandom)

Core Module: random

The most commonly used functions in the random module:

randint() Takes two integer values and returns a random integer between those values inclusive.

randrange() Takes the same input as range() and returns a random integer that falls within that range.

uniform() Does almost the same thing as randint(), but returns a float and is inclusive only of the smaller number(exclusive of the larger number)

random() Works just like uniform() except that the smaller number is fixed at 0.0, and the larger number is fixed at 1.0

choice() Given a sequence , randomly selects and returns a sequence item.

Python Operators

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a particular programming language. Python provides a variety of operators described as follows.

- Arithmetic operators
- Comparison operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Arithmetic operators

Arithmetic operators are used to perform arithmetic operations between two operands. It includes +(addition), - (subtraction), *(multiplication), /(divide), %(reminder), //(floor division), and exponent (**).

Consider the following table for a detailed explanation of arithmetic operators.

Operator	Description
+ (Addition)	It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30
- (Subtraction)	It is used to subtract the second operand from the first operand. If the first operand is less than the second operand, the value result negative. For example, if a = 20, b = 10 => a ? b = 10
/ (divide)	It returns the quotient after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a/b = 2
** * (Multiplication)**	It is used to multiply one operand with the other. For example, if a = 20, b = 10 => a * b = 200

% (reminder)	It returns the reminder after dividing the first operand by the second operand. For example, if a = 20, b = 10 => a%b = 0
**** (Exponent)**	It is an exponent operator represented as it calculates the first operand power to second operand.
// (Floor division)	It gives the floor value of the quotient produced by dividing the two operands.

Comparison operator

Comparison operators are used to comparing the value of the two operands and returns boolean true or false accordingly. The comparison operators are described in the following table.

Operator	Description
==	If the value of two operands is equal, then the condition becomes true.
!=	If the value of two operands is not equal then the condition becomes true.
<=	If the first operand is less than or equal to the second operand, then the condition becomes true.
>=	If the first operand is greater than or equal to the second operand, then the condition becomes true.
<>	If the value of two operands is not equal, then the condition becomes true.
>	If the first operand is greater than the second operand, then the condition becomes true.
<	If the first operand is less than the second operand, then the condition becomes true.

Python assignment operators

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

Operator	Description
=	It assigns the the value of the right expression to the left operand.
+=	It increases the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b will be equal to a = a+ b and therefore, a = 30.
-=	It decreases the value of the left operand by the value of the right operand and assign

	the modified value back to left operand. For example, if a = 20, b = 10 => a- = b will be equal to a = a- b and therefore, a = 10.
*=	It multiplies the value of the left operand by the value of the right operand and assign the modified value back to left operand. For example, if a = 10, b = 20 => a* = b will be equal to a = a* b and therefore, a = 200.
%=	It divides the value of the left operand by the value of the right operand and assign the reminder back to left operand. For example, if a = 20, b = 10 => a % = b will be equal to a = a % b and therefore, a = 0.
**=	a**=b will be equal to a=a**b, for example, if a = 4, b =2, a**=b will assign 4**2 = 16 to a.
//=	A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign 4//3 = 1 to a.

Bitwise operator

The bitwise operators perform bit by bit operation on the values of the two operands.

For example,

1. **if** a = 7;
2. b = 6;
3. then, binary (a) = 0111
4. binary (b) = 0011
5.
6. hence, a & b = 0011
7. a | b = 0111
8. a ^ b = 0100
9. ~ a = 1000

Operator	Description
& (binary and)	If both the bits at the same place in two operands are 1, then 1 is copied to the result. Otherwise, 0 is copied.
| (binary or)	The resulting bit will be 0 if both the bits are zero otherwise the resulting bit will be 1.
^ (binary xor)	The resulting bit will be 1 if both the bits are different otherwise the resulting bit will be 0.
~ (negation)	It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting bit will be 1 and vice versa.
<< (left	The left operand value is moved left by the number of bits present in the right operand.

shift)	
>> (right shift)	The left operand is moved right by the number of bits present in the right operand.

Logical Operators

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

Operator	Description
and	If both the expression are true, then the condition will be true. If a and b are the two expressions, a → true, b → true => a and b → true.
or	If one of the expressions is true, then the condition will be true. If a and b are the two expressions, a → true, b → false => a or b → true.
not	If an expression **a** is true then not (a) will be false and vice versa.

Membership Operators

Python membership operators are used to check the membership of value inside a data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

Operator	Description
in	It is evaluated to be true if the first operand is found in the second operand (list, tuple, or dictionary).
not in	It is evaluated to be true if the first operand is not found in the second operand (list, tuple, or dictionary).

Identity Operators

Operator	Description
is	It is evaluated to be true if the reference present at both sides point to the same object.
is not	It is evaluated to be true if the reference present at both side do not point to the same object.

Operator Precedence

The precedence of the operators is important to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in python is given below.

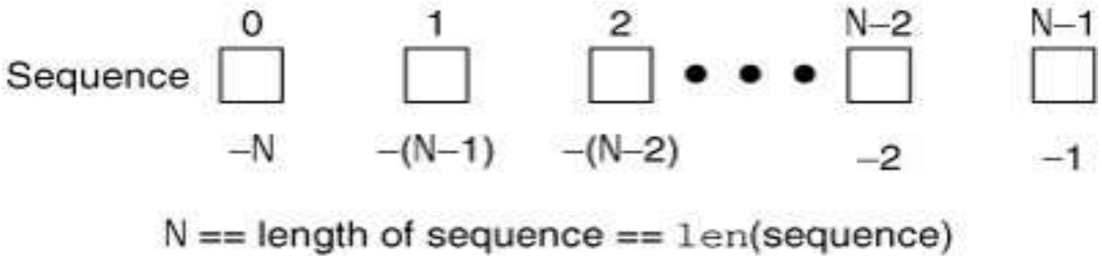
Operator	Description
**	The exponent operator is given priority over all the others used in the expression.
~ + -	The negation, unary plus and minus.
* / % //	The multiplication, divide, modules, reminder, and floor division.
+ -	Binary plus and minus
>> <<	Left shift and right shift
&	Binary and.
^ |	Binary xor and or
<= < > >=	Comparison operators (less then, less then equal to, greater then, greater then equal to).
<> == !=	Equality operators.
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

6. Sequences: Strings, Lists, and Tuples

6.1. Sequences

Sequence types all share the same access model: ordered set with sequentially indexed offsets to get to each element. Multiple elements may be selected by using the slice operators. The numbering scheme used starts from zero (0) and ends with one less than the length of the sequence.

Figure 6.1. How sequence elements are stored and accessed



6.1.1 Standard Type Operators

The standard type operators (Python Operators) generally work with all sequence types.

6.1.2 Sequence Type Operators

A list of all the operators applicable to all sequence types is given in Table 6.1.

Table 6.1 Sequence Type Operators

Sequence Operator	*Function*
seq[ind]	Element located at index ind of seq
seq[ind1:ind2]	Elements from ind1 up to but not including ind2 of seq
seq * expr	seq repeated expr times
seq1 + seq2	Concatenates sequences seq1 and seq2
obj **in** seq	Tests if obj is a member of sequence seq
obj **not in** seq	Tests if obj is not a member of sequence seq

Membership (in, not in)

Membership test operators are used to determine whether an element is in or is a member of a sequence. For strings, this test is whether a character is in a string, and for lists and tuples, it is whether an object is an element of those sequences. The in and not in operators are Boolean in nature; they return True if the membership is confirmed and False otherwise. The syntax for using the membership operators is as follows:

obj [not] in sequence

Concatenation (+)

This operation allows us to take one sequence and join it with another sequence of the same type. The syntax for using the concatenation operator is as follows:

sequence1 + sequence2

The resulting expression is a new sequence that contains the combined contents of sequences sequence1 and sequence2.

Repetition (*)

The repetition operator is useful when consecutive copies of sequence elements are desired. The syntax for using the repetition operator is as follows:

sequence * copies_int

Slices ([ ], [:], [: :])
Sequences are data structures that hold objects in an ordered manner. You can get access to individual elements with an index and pair of brackets, or a consecutive group of elements with the brackets and colons giving the indices of the elements you want starting from one index and going up to but not including the ending index.

Sequences are structured data types whose elements are placed sequentially in an ordered manner. This format allows for individual element access by index offset or by an index range of indices to select groups of sequential elements in a sequence. This type of access is called slicing, and the slicing operators allow us to perform such access.
The syntax for accessing an individual element is:

sequence[index]

sequence is the name of the sequence and index is the offset into the sequence where the desired element is located. Index values can be positive, ranging from 0 to the maximum index (which is length of the sequence less one). Using the len(), this gives an index with the range 0 <= index <= len(sequence)-1.
Alternatively, negative indexes can be used, ranging from –1 to the negative length of the sequence, -len(sequence), i.e., -len(sequence) <= index <= -1. The difference between the positive and negative indexes is that positive indexes start from the beginning of the sequences and negative indexes work backward from the end.
Attempting to retrieve a sequence element with an index outside of the length of the sequence results in an IndexError exception.

Accessing a group of elements is similar to accessing just a single item. Starting and ending indexes may be given, separated by a colon (:). The syntax for accessing a group of elements is:

sequence[starting_index:ending_index]

Using this syntax, we can obtain a “slice” of elements in sequence from the starting_index up to but not including the element at the ending_index index. Both starting_index and ending_index are optional, and if not provided, or if None is used as an index, the slice will go from the beginning of the sequence or until the end of the sequence, respectively.

Extended Slicing with Stride Indices
The final slice syntax for sequences, known as extended slicing, involves a third index known as a stride. You can think of a stride index like a “step” value as the third element of a call to the range()built-in function

```
>>> s = 'abcdefgh'
>>> s[::-1] # think of it as 'reverse'
'hgfedcba'
>>> s[::2] # think of it as skipping by 2
'aceg'
```

6.1.3 Built-in Functions (BIFs)

Table 6.2 Sequence Type *Conversion* Factory Functions

Function	*Operation*
`list(iter)`	Converts *iterable* to a list
`str(obj)`	Converts *obj* to string (a printable string representation)
`unicode(obj)`	Converts *obj* to a Unicode string (using default encoding)
`basestring()`	Abstract factory function serves only as parent class of `str` and `unicode`, so cannot be called/instantiated (see Section 6.2)
`tuple(iter)`	Converts *iterable* to a tuple

6.2. Strings

Strings are sequence of characters. We can create them simply by enclosing characters in quotes. Python treats single quotes the same as double quotes. Strings are a literal or scalar type, meaning they are treated by the interpreter as a singular value and are not containers that hold other Python objects. Strings are immutable, meaning that changing an element of a string requires creating a new string. Strings are made up of individual characters, and such elements of strings may be accessed sequentially via slicing.

How to Create and Assign Strings

Creating strings is as simple as using a scalar value or having the str() factory function make one and assigning it to a variable:

```
>>> aString = 'Hello World!' # using single quotes
>>> anotherString = "Python is cool!" # double quotes
>>> print aString # print, no quotes!
Hello World!
>>> anotherString # no print, quotes!
'Python is cool!'
>>> s = str(range(4)) # turn list to string
>>> s
'[0, 1, 2, 3]'
```

How to Access Values (Characters and Substrings) in Strings

Python does not support a character type; these are treated as strings of length one, thus also considered a substring. To access substrings, use the square brackets for slicing along with the index or indices to obtain your substring:

```
>>> aString = 'Hello World!'
>>> aString[0]
'H'
```

```
>>> aString[1:5]
'ello'
>>> aString[6:]
'World!'
```

How to Update Strings

You can "update" an existing string by (re)assigning a variable to another string. The new value can be related to its previous value or to a completely different string altogether.

```
>>> aString = aString[:6] + 'Python!'
>>> aString
'Hello Python!'
>>> aString = 'different string altogether'
>>> aString
'different string altogether'
```

Like numbers, strings are immutable, so you cannot change an existing string without creating a new one from scratch. That means that you cannot update individual characters or substrings in a string.

How to Remove Characters and Strings

Strings are immutable, so you cannot remove individual characters from an existing string. We want to remove one letter from "Hello World!"...the (lowercase) letter "l," for example:

```
>>> aString = 'Hello World!'
>>> aString = aString[:3] + aString[4:]
>>> aString
'Helo World!'
```

To clear or remove a string, you assign an empty string or use the **del** statement, respectively:

```
>>> del aString
```

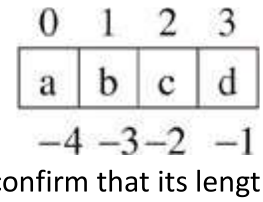
6.3. Strings and Operators

Slices ([ ] and [:])

We can access individual or a group of elements from a sequence.

- Counting forward
- Counting backward
- Default/missing indexes

For the following examples, we use the single string 'abcd'. Provided in the figure is a list of positive and negative indexes that indicate the position in which each character is located within the string itself.



Using the length operator, we can confirm that its length is 4:

```
>>> aString = 'abcd'
```

>>> len(aString)

4

When counting forward, indexes start at 0 to the left and end at one less than the length of the string (because we started from zero). The final index of our string is:

final_index = len(aString) - 1

= 4 - 1

= 3

We can access any substring within this range. The slice operator with a single argument will give us a single character, and the slice operator with a range, i.e., using a colon (:), will give us multiple consecutive characters. Again, for any ranges [*start*:*end*], we will get all characters starting at offset *start* up to, but not including, the character at *end*. In other words, for all characters *x* in the range

[*start*:*end*], *start* <= x < *end*.

>>> aString[0]

'a'

>>> aString[1:3]

'bc'

When counting backward, we start at index -1 and move toward the beginning of the string, ending at negative value of the length of the string. The final index (the first character) is located at:

final_index = -len(aString)

= -4

>>> aString[-1]

'd'

>>> aString[-3:-1]

'bc'

>>> aString[-4]

'a'

When either a starting or an ending index is missing, they default to the beginning or end of the string, respectively.

>>> aString[2:]

'cd'

>>> aString[1:]

'bcd'

Membership (in, not in)

The membership operators are used to test a (sub)string appears in a string or not. True is returned if that character appears in the string and False otherwise.

>>> 'bc' **in** 'abcd'

True

>>> 'n' **in** 'abcd'

False

>>> 'nm' **not in** 'abcd'
True
Built-in String functions
Python provides various in-built functions that are used for string handling. Many String fun

Method	**Description**
capitalize()	It capitalizes the first character of the String. This function is deprecated in python3
casefold()	It returns a version of s suitable for case-less comparisons.
center(width ,fillchar)	It returns a space padded string with the original string centred with equal number of left and right spaces.
count(string,begin,end)	It counts the number of occurrences of a substring in a String between begin and end index.
decode(encoding = 'UTF8', errors = 'strict')	Decodes the string using codec registered for encoding.
encode()	Encode S using the codec registered for encoding. Default encoding is 'utf-8'.
endswith(suffix ,begin=0,end=len(string))	It returns a Boolean value if the string terminates with given suffix between begin and end.
expandtabs(tabsize = 8)	It defines tabs in string to multiple spaces. The default space value is 8.
find(substring ,beginIndex, endIndex)	It returns the index value of the string where substring is found between begin index and end index.
format(value)	It returns a formatted version of S, using the passed value.
index(subsring, beginIndex, endIndex)	It throws an exception if string is not found. It works same as find() method.
isalnum()	It returns true if the characters in the string are alphanumeric i.e., alphabets or numbers and there is at least 1 character. Otherwise, it returns false.

isalpha()	It returns true if all the characters are alphabets and there is at least one character, otherwise False.
isdecimal()	It returns true if all the characters of the string are decimals.
isdigit()	It returns true if all the characters are digits and there is at least one character, otherwise False.
isidentifier()	It returns true if the string is the valid identifier.
islower()	It returns true if the characters of a string are in lower case, otherwise false.
isnumeric()	It returns true if the string contains only numeric characters.
isprintable()	It returns true if all the characters of s are printable or s is empty, false otherwise.
isupper()	It returns false if characters of a string are in Upper case, otherwise False.
isspace()	It returns true if the characters of a string are white-space, otherwise false.
istitle()	It returns true if the string is titled properly and false otherwise. A title string is the one in which the first character is upper-case whereas the other characters are lower-case.
isupper()	It returns true if all the characters of the string(if exists) is true otherwise it returns false.
join(seq)	It merges the strings representation of the given sequence.
len(string)	It returns the length of a string.
ljust(width[,fillchar])	It returns the space padded strings with the original string left justified to the given width.
lower()	It converts all the characters of a string to Lower case.

lstrip()	It removes all leading whitespaces of a string and can also be used to remove particular character from leading.
partition()	It searches for the separator sep in S, and returns the part before it, the separator itself, and the part after it. If the separator is not found, return S and two empty strings.
maketrans()	It returns a translation table to be used in translate function.
replace(old,new[,count])	It replaces the old sequence of characters with the new sequence. The max characters are replaced if max is given.
rfind(str,beg=0,end=len(str))	It is similar to find but it traverses the string in backward direction.
rindex(str,beg=0,end=len(str))	It is same as index but it traverses the string in backward direction.
rjust(width,[,fillchar])	Returns a space padded string having original string right justified to the number of characters specified.
rstrip()	It removes all trailing whitespace of a string and can also be used to remove particular character from trailing.
rsplit(sep=None, maxsplit = -1)	It is same as split() but it processes the string from the backward direction. It returns the list of words in the string. If Separator is not specified then the string splits according to the white-space.
split(str,num=string.count(str))	Splits the string according to the delimiter str. The string splits according to the space if the delimiter is not provided. It returns the list of substring concatenated with the delimiter.
splitlines(num=string.count('\n'))	It returns the list of strings at each line with newline removed.
startswith(str,beg=0,end=len(str))	It returns a Boolean value if the string starts with given str between begin and end.

strip([chars])	It is used to perform lstrip() and rstrip() on the string.
swapcase()	It inverts case of all characters in a string.
title()	It is used to convert the string into the title-case i.e., The string **meEruT** will be converted to Meerut.
translate(table,deletechars = '')	It translates the string according to the translation table passed in the function .
upper()	It converts all the characters of a string to Upper Case.

6.11 Lists

Lists provide sequential storage through an index offset and access to single or consecutive elements through slices. Strings consist only of characters and are immutable (cannot change individual elements), while lists are flexible container objects that hold an arbitrary number of Python objects. Creating lists is simple; adding to lists is easy.

Lists can be populated, empty, sorted, and reversed. Lists can be grown and shrunk. They can be taken apart and put together with other lists. Individual or multiple items can be inserted, updated, or removed at will.

How to Create and Assign Lists

Creating lists is as simple as assigning a value to a variable. Lists are delimited by surrounding square brackets ([ ]). You can also use the factory function.

```
>>> aList = [123, 'abc', 4.56, ['inner', 'list'], 7-9j]
>>> anotherList = [None, 'something to see here']
>>> print aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> print anotherList
[None, 'something to see here']
>>> aListThatStartedEmpty = []
>>> print aListThatStartedEmpty
[]
>>> list('foo')
['f', 'o', 'o']
```

How to Access Values in Lists

Slicing works similar to strings; use the square bracket slice operator ([ ]) along with the index or indices.

```
>>> aList[0]
123
>>> aList[1:4]
```

```
['abc', 4.56, ['inner', 'list']]
>>> aList[:3]
[123, 'abc', 4.56]
>>> aList[3][1]
'list'
```

How to Update Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method:

```
>>> aList
[123, 'abc', 4.56, ['inner', 'list'], (7-9j)]
>>> aList[2]
4.56
>>> aList[2] = 'float replacer'
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>>
>>> anotherList.append("hi, i'm new here")
>>> print anotherList
[None, 'something to see here', "hi, i'm new here"]
>>> aListThatStartedEmpty.append('not empty anymore')
>>> print aListThatStartedEmpty
['not empty anymore']
```

How to Remove List Elements and Lists

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know.

```
>>> aList
[123, 'abc', 'float replacer', ['inner', 'list'], (7-9j)]
>>> del aList[1]
>>> aList
[123, 'float replacer', ['inner', 'list'], (7-9j)]
>>> aList.remove(123)
>>> aList
['float replacer', ['inner', 'list'], (7-9j)]
```

You can also use the pop() method to remove and return a specific object from a list.

if we want to explicitly remove an entire list, they use the del statement:

```
del aList
```

6.12 Operators

6.12.1 Standard Type Operators

```
>>> list1 = ['abc', 123]
>>> list2 = ['xyz', 789]
>>> list3 = ['abc', 123]
>>> 1ist1 < list2
True
>>> list2 < list3
False
>>> list2 > list3 and list1 == list3
True
```

6.12.2 Sequence Type Operators

Slices ([ ] and [:])

Slicing with lists is very similar to strings, but rather than using individual characters or substrings, slices of lists pull out an object or a group of objects that are elements of the list operated on. Focusing specifically on lists, we make the following definitions:

```
>>> num_list = [43, -1.23, -2, 6.19e5]
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
>>> mixup_list = [4.0, [1, 'x'], 'beef', -1.9+6j]
```

Slicing operators obey the same rules regarding positive and negative indexes, starting and ending indexes, as well as missing indexes, which default to the beginning or to the end of a sequence.

```
>>> num_list[1]
-1.23
>>>
>>> num_list[1:]
[-1.23, -2, 619000.0]
>>>
>>> num_list[2:-1]
[-2]
>>>
>>> str_list[2]
'over'
>>> str_list[:2]
['jack', 'jumped']
>>>
>>> mixup_list
[4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>> mixup_list[1]
[1, 'x']
```

Membership (in, not in)

With lists (and tuples), we can check whether an object is a member of a list (or tuple).

```
>>> mixup_list
[4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>>
>>> 'beef' in mixup_list
True
>>>
>>> 'x' in mixup_list
False
```

Concatenation (+)

The concatenation operator allows us to join multiple lists together. you can concatenate only objects of the same type. You cannot concatenate two different types even if both are sequences.

```
>>> num_list = [43, -1.23, -2, 6.19e5]
>>> str_list = ['jack', 'jumped', 'over', 'candlestick']
>>> mixup_list = [4.0, [1, 'x'], 'beef', -1.9+6j]
>>>
>>> num_list + mixup_list
[43, -1.23, -2, 619000.0, 4.0, [1, 'x'], 'beef', (-1.9+6j)]
>>>
>>> str_list + num_list
['jack', 'jumped', 'over', 'candlestick', 43, -1.23, -2, 619000.0]
```

Repetition (*)

Use of the repetition operator may make more sense with strings, but as a sequence type, lists and tuples can also benefit from this operation, if needed:

```
>>> num_list * 2
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0]
>>>
>>> num_list * 3
[43, -1.23, -2, 619000.0, 43, -1.23, -2, 619000.0, 43,-1.23, -2, 619000.0]
```

6.13.1 Standard Type Functions

cmp()

```
>>> list1, list2 = [123, 'xyz'], [456, 'abc']
>>> cmp(list1, list2)
-1
>>>
>>> cmp(list2, list1)
```

1
>>> list3 = list2 + [789]
>>> list3
[456, 'abc', 789]
>>>
>>> cmp(list2, list3)
-1
Comparison algorithm highlights:
1. Compare elements of both lists.
2. If elements are of the same type, perform the compare and return the result.
3. If elements are different types, check to see if they are numbers.
    a. If numbers, perform numeric coercion if necessary and compare.
    b. If either element is a number, then the other element is “larger” (numbers are “smallest”).
    c. Otherwise, types are sorted alphabetically by name.
4. If we reach the end of one of the lists, the longer list is “larger.”
5. If we exhaust both lists and share the same data, the result is a tie, meaning that 0 is returned.

6.13.2 Sequence Type Functions
len()
len() returns the number of elements in the list (or tuple).
>>> len(num_list)
4

max() and min()
>>> max(str_list)
'over'
>>> max(num_list)
619000.0
>>> min(str_list)
'candlestick'
>>> min(num_list)
-2
sorted() and reversed()
>>> s = ['They', 'stamp', 'them', 'when', "they're", 'small']
>>> for t in reversed(s):
... print t,
...
small they're when them stamp They
>>> sorted(s)

['They', 'small', 'stamp', 'them', "they're", 'when']

Table 6.11 List Type Built-in Methods

List Method	*Operation*
list.append(*obj*)	Adds *obj* to the end of *list*
list.count(*obj*)	Returns count of how many times *obj* occurs in *list*
list.extend(*seq*)[a]	Appends contents of *seq* to *list*
list.index(*obj*, *i=0*, *j=len*(*list*))	Returns lowest index *k* where *list*[*k*] == *obj* and *i* <= *k* < *j*; otherwise ValueError raised
list.insert(*index*, *obj*)	Inserts *obj* into *list* at offset *index*
list.pop(*index*=-1)[a]	Removes and returns *obj* at given or last *index* from *list*
list.remove(*obj*)	Removes object *obj* from *list*
list.reverse()	Reverses objects of *list* in place
list.sort(*func*=None, *key*=None, *reverse*=False)[b]	Sorts list members with optional comparison *function*; *key* is a callback when extracting elements for sorting, and if *reverse* flag is True, then list is sorted in reverse order

6.16 Tuples

Tuples are another container type extremely similar in nature to lists. The only visible difference between tuples and lists is that tuples use parentheses and lists use square brackets. Functionally, there is a more significant difference, and that is the fact that tuples are immutable. Because of this, tuples can do something that lists cannot do . . . be a dictionary key. Tuples are also the default when dealing with a group of objects.

How to Create and Assign Tuples

Creating and assigning tuples are practically identical to creating and assigning lists, with the exception of tuples with only one element—these require a trailing comma (,) enclosed in the tuple delimiting parentheses (( )) to prevent them from being confused with the natural grouping operation of parentheses. Do not forget the factory function!

```
>>> aTuple = (123, 'abc', 4.56, ['inner', 'tuple'], 7-9j)
>>> anotherTuple = (None, 'something to see here')
>>> print aTuple
```

```
(123, 'abc', 4.56, ['inner', 'tuple'], (7-9j))
>>> print anotherTuple
(None, 'something to see here')
>>> singleItemTuple = (None,)
>>> print singleItemTuple
(None,)
>>> tuple('bar')
('b', 'a', 'r')
```

How to Access Values in Tuples

Slicing works similarly to lists. Use the square bracket slice operator ([ ]) along with the index or indices.

```
>>> aTuple[1:4]
('abc', 4.56, ['inner', 'tuple'])
>>> aTuple[:3]
(123, 'abc', 4.56)
>>> aTuple[3][1]
'tuple'
```

How to Update Tuples

Like numbers and strings, tuples are immutable, which means you cannot update them or change values of tuple elements.

```
>>> aTuple = aTuple[0], aTuple[1], aTuple[-1]
>>> aTuple
(123, 'abc', (7-9j))
>>> tup1 = (12, 34.56)
>>> tup2 = ('abc', 'xyz')
>>> tup3 = tup1 + tup2
>>> tup3
(12, 34.56, 'abc', 'xyz')
```

How to Remove Tuple Elements and Tuples

Removing individual tuple elements is not possible.

6.17.1 Standard and Sequence Type

Operators and Built-in Functions

Object and sequence operators and built-in functions act the exact same way toward tuples as they do with lists. You can still take slices of tuples, concatenate and make multiple copies of tuples, validate membership, and compare tuples.

Creation, Repetition, Concatenation

```
>>> t = (['xyz', 123], 23, -103.4)
>>> t
(['xyz', 123], 23, -103.4)
>>> t * 2
(['xyz', 123], 23, -103.4, ['xyz', 123], 23, -103.4)
>>> t = t + ('free', 'easy')
>>> t
(['xyz', 123], 23, -103.4, 'free', 'easy')
```

Membership, Slicing

```
>>> 23 in t
True
>>> 123 in t
False
>>> t[0][1]
123
>>> t[1:]
(23, -103.4, 'free', 'easy')
```

Built-in Functions

```
>>> str(t)
(['xyz', 123], 23, -103.4, 'free', 'easy')
>>> len(t)
5
>>> max(t)
'free'
>>> min(t)
-103.4
>>> cmp(t, (['xyz', 123], 23, -103.4, 'free', 'easy'))
0
>>> list(t)
[['xyz', 123], 23, -103.4, 'free', 'easy']
```

Operators

```
>>> (4, 2) < (3, 5)
False
>>> (2, 4) < (3, -1)
True
>>> (2, 4) == (3, -1)
False
>>> (2, 4) == (2, 4)
True
```

7.1 Mapping Type: Dictionaries

Dictionaries are the sole mapping type in Python. Mapping objects have a one to- much correspondence between hashable values (keys) and the objects they represent (values). A dictionary object itself is mutable and is yet another container type that can store any number of Python objects, including other container types.

Hash tables are a data structure that store each piece of data, called a value, based on an associated data item, called a key. Together, these are known as key-value pairs.The hash table algorithm takes your key, performs an operation on it, called a hash function, and based on the result of the calculation, chooses where in the data structure to store your value. Where any one particular value is stored depends on what its key is. Because of this randomness, there is no ordering of the values in the hash table. You have an unordered collection of data.

The only kind of ordering you can obtain is by taking either a dictionary's set of keys or values. The keys() or values() method returns lists, which are sortable. You can also call items() to get a list of keys and values as tuple pairs and sort that. Dictionaries themselves have no implicit ordering because they are hashes. Python dictionaries are implemented as resizeable hash tables. The syntax of a dictionary entry is key:value. Also, dictionary entries are enclosed in braces ({ }).

How to Create and Assign Dictionaries

Creating dictionaries simply involves assigning a dictionary to a variable, regardless of whether the dictionary has elements or not:

```
>>> dict1 = {}
>>> dict2 = {'name': 'earth', 'port': 80}
>>> dict1, dict2
({}, {'port': 80, 'name': 'earth'})
```

In Python versions 2.2 and newer, dictionaries may also be created using the factory function dict(). We discuss more examples later when we take a closer look at dict(), but here's a sneak peek for now:

```
>>> fdict = dict((['x', 1], ['y', 2]))
>>> fdict
{'y': 2, 'x': 1}
```

How to Access Values in Dictionaries

To traverse a dictionary (normally by key), you only need to cycle through its keys, like this:

```
>>> dict2 = {'name': 'earth', 'port': 80}
>>>
>>>> for key in dict2.keys():
... print 'key=%s, value=%s' % (key, dict2[key])
...
key=name, value=earth
key=port, value=80
```

```
>>> dict2 = {'name': 'earth', 'port': 80}
>>>
>>>> for key in dict2:
... print 'key=%s, value=%s' % (key, dict2[key])
...
key=name, value=earth
key=port, value=80
```

To access individual dictionary elements, you use the familiar square brackets along with the key to obtain its value:

```
>>> dict2['name']
'earth'
>>>
```

If we attempt to access a data item with a key that is not part of the dictionary,
we get an error:

```
>>> dict2['server']
Traceback (innermost last):
File "<stdin>", line 1, in ?
KeyError: server
```

The in and not in operators are Boolean, returning True if a dictionary has that key and False otherwise.

```
>>> 'server' in dict2
False
>>> 'name' in dict2
True
>>> dict2['name']
'earth'
```

How to Update Dictionaries

You can update a dictionary by adding a new entry or element (i.e., a keyvalue pair), modifying an existing entry, or deleting an existing entry.

```
>>> dict2['name'] = 'venus' # update existing entry
>>> dict2['port'] = 6969 # update existing entry
>>> dict2['arch'] = 'sunos5'# add new entry
>>>
```

How to Remove Dictionary Elements and Dictionaries

Removing an entire dictionary is not a typical operation. Generally, you either remove individual dictionary elements or clear the entire contents of a dictionary.

However, if you really want to “remove” an entire dictionary, use the del statement. Here are some deletion examples for dictionaries and dictionary elements:

```
del dict2['name'] # remove entry with key 'name'
dict2.clear() # remove all entries in dict1
```

del dict2 # delete entire dictionary
dict2.pop('name') # remove & return entry w/key

7.2 Mapping Type Operators

Dictionaries will work with all of the standard type operators but do not support operations such as concatenation and repetition.

7.2.1 Standard Type Operators

Here are some basic examples using some of those operators:

```
>>> dict4 = {'abc': 123}
>>> dict5 = {'abc': 456}
>>> dict6 = {'abc': 123, 98.6: 37}
>>> dict7 = {'xyz': 123}
>>> dict4 < dict5
True
>>> (dict4 < dict6) and (dict4 < dict7)
True
>>> (dict5 < dict6) and (dict5 < dict7)
True
>>> dict6 < dict7
False
```

7.2.2 Mapping Type Operators

Dictionary Key-Lookup Operator ([ ])

The only operator specific to dictionaries is the key-lookup operator, which works very similarly to the single element slice operator for sequence types. For sequence types, an index offset is the sole argument or subscript to access a single element of a sequence. For a dictionary, lookups are by key, so that is the argument rather than an index. The key-lookup operator is used for both assigning values to and retrieving values from a dictionary:

```
d[k] = v # set value in dictionary
d[k] # lookup value in dictionary
```

(Key) Membership (in, not in)

programmers can use the in and not in operators to check key membership instead of the has_key() method:

```
>>> 'name' in dict2
True
>>> 'phone' in dict2
False
```

7.3 Mapping Type Built-in and Factory Functions

7.3.1 Standard Type Functions [type(), str(), and cmp()]

The type() factory function, when applied to a dict, returns, as you might expect, the dict type, “<type 'dict'>”. The str() factory function will produce a printable string representation of a dictionary.

Comparisons of dictionaries are based on an algorithm that starts with sizes first, then keys, and finally values. However, using cmp() on dictionaries isn’t usually very useful.

Dictionary Comparison Algorithm

```
>>> dict1 = {}
>>> dict2 = {'host': 'earth', 'port': 80}
>>> cmp(dict1, dict2)
-1
>>> dict1['host'] = 'earth'
>>> cmp(dict1, dict2)
-1
```

In the first comparison, dict1 is deemed smaller because dict2 has more elements (2 items vs. 0 items). After adding one element to dict1, it is still smaller (2 vs. 1), even if the item added is also in dict2.

```
>>> dict1['port'] = 8080
>>> cmp(dict1, dict2)
1
>>> dict1['port'] = 80
>>> cmp(dict1, dict2)
0
```

After we add the second element to dict1, both dictionaries have the same size, so their keys are then compared. At this juncture, both sets of keys match, so comparison proceeds to checking their values. The values for the 'host' keys are the same, but when we get to the 'port' key, dict1 is deemed larger because its value is greater than that of dict2’s 'port' key (8080 vs. 80). When resetting dict2’s 'port' key to the same value as dict1’s 'port' key, then both dictionaries form equals: They have the same size, their keys match, and so do their values, hence the reason that 0 is returned by cmp().

```
>>> dict1['prot'] = 'tcp'
>>> cmp(dict1, dict2)
1
>>> dict2['prot'] = 'udp'
>>> cmp(dict1, dict2)
-1
```

As soon as an element is added to one of the dictionaries, it immediately becomes the “larger one,” as in this case with dict1. Adding another keyvalue pair to dict2 can tip the scales again, as both dictionaries’ sizes match and comparison progresses to checking keys and values.

```
>>> cdict = {'fruits':1}
```

```
>>> ddict = {'fruits':1}
>>> cmp(cdict, ddict)
0
>>> cdict['oranges'] = 0
>>> ddict['apples'] = 0
>>> cmp(cdict, ddict)
14
```

The algorithm pursues comparisons in the following order.

(1) Compare Dictionary Sizes

If the dictionary lengths are different, then for cmp(dict1, dict2), cmp() will return a positive number if dict1 is longer and a negative number if dict2 is longer. In other words, the dictionary with more keys is greater, i.e., len(dict1) > len(dict2) or dict1 > dict2

(2) Compare Dictionary Keys

If both dictionaries are the same size, then their keys are compared; the order in which the keys are checked is the same order as returned by the keys() method. At the point where keys from both do not match, they are directly compared and cmp() will return a positive number if the first differing key for dict1 is greater than the first differing key of dict2.

(3) Compare Dictionary Values

If both dictionary lengths are the same and the keys match exactly, the values for each key in both dictionaries are compared. Once the first key with nonmatching values is found, those values are compared directly. Then cmp() will return a positive number if, using the same key, the value in dict1 is greater than the value in dict2.

(4) Exact Match

If we have reached this point, i.e., the dictionaries have the same length, the same keys, and the same values for each key, then the dictionaries are an exact match and 0 is returned.

7.4 Mapping Type Built-in Methods

Table 7.2 Dictionary Type Methods

Method Name	*Operation*
`dict.clear`[a]`()`	Removes all elements of *dict*
`dict.copy()`	Returns a (shallow[b]) copy of *dict*
`dict.fromkeys`[c] `(seq, val=None)`	Creates and returns a new dictionary with the elements of *seq* as the keys and *val* as the initial value (defaults to None if not given) for all keys
`dict.get(key, default=None)`[a]	For key *key*, returns value or *default* if *key* not in *dict* (note that *default*'s default is None)
`dict.has_key(key)`[f]	Returns True if *key* is in *dict*, False otherwise; partially deprecated by the **in** and **not in** operators in 2.2 but still provides a functional interface
`dict.items()`	Returns an iterable[g] of the (key, value) tuple pairs of *dict*
`dict.iter*`[d]`()`	iteritems(), iterkeys(), itervalues() are all methods that behave the same as their non-iterator counterparts but return an iterator instead of a list
`dict.keys()`	Returns an iterable[g] of the keys of *dict*
`dict.pop`[c]`(key [, default])`	Similar to get() but removes and returns *dict*[*key*] if key present and raises KeyError if key not in *dict* and *default* not given
`dict.setdefault (key, default=None)`[e]	Similar to get(), but sets *dict*[*key*]=*default* if key is not already in dict
`dict.update(dict2)`[a]	Add the key-value pairs of *dict2* to *dict*
`dict.values()`	Returns an iterable[g] of the values of *dict*

Basic dictionary methods focus on their keys and values. These are keys(), which returns a list of the dictionary's keys, values(), which returns a list of the dictionary's values, and items(), which returns a list of (key, value) tuple pairs. These are useful when you wish to iterate through a dictionary's keys or values, albeit in no particular order.

```
>>> dict2.keys()
['port', 'name']
>>>
>>> dict2.values()
[80, 'earth']
>>>
```

```
>>> dict2.items()
[('port', 80), ('name', 'earth')]
>>>
>>> for eachKey in dict2.keys():
... print 'dict2 key', eachKey, 'has value', dict2[eachKey]
...
dict2 key port has value 80
dict2 key name has value earth
```

7.6 Set Types

In mathematics, a set is any collection of distinct items, and its members are often referred to as set elements. A set object is an unordered collection of hashable values.

Like other container types, sets support membership testing via in and not in operators, cardinality using the len()BIF, and iteration over the set membership using for loops. However, since sets are unordered, you do not index into or slice them, and there are no keys used to access a value.

There are two different types of sets available, mutable (set) and immutable (frozenset). you are allowed to add and remove elements from the mutable form but not the immutable. Note that mutable sets are not hashable and thus cannot be used as either a dictionary key or as an element of another set. The reverse is true for frozen sets, i.e., they have a hash value and can be used as a dictionary key or a member of a set.

Table 7.3 Set Operation and Relation Symbols

Mathematical Symbol	*Python Symbol*	*Description*
$\in$	`in`	Is a member of
$\notin$	`not in`	Is not a member of
	`==`	Is equal to
$\neq$	`!=`	Is not equal to
$\subset$	`<`	Is a (strict) subset of
$\subseteq$	`<=`	Is a subset of (includes improper subsets)
$\supset$	`>`	Is a (strict) superset of
$\supseteq$	`>=`	Is a superset of (includes improper supersets)
$\cap$	`&`	Intersection
$\cup$	`|`	Union
$-$ or $\setminus$	`-`	Difference or relative complement
Δ	`^`	Symmetric difference

How to Create and Assign Set Types

There is no special syntax for sets like there is for lists ([ ]) and dictionaries ({ }). Lists and dictionaries can also be created with their corresponding factory functions list() and dict(), and that is also the only way sets can be created, using their factory functions set() and frozenset():

```
>>> s = set('cheeseshop')
>>> s
set(['c', 'e', 'h', 'o', 'p', 's'])
>>> t = frozenset('bookshop')
>>> t
frozenset(['b', 'h', 'k', 'o', 'p', 's'])
>>> type(s)
<type 'set'>
>>> type(t)
<type 'frozenset'>
>>> len(s)
6
>>> len(s) == len(t)
True
>>> s == t
False
```

How to Access Values in Sets

You are either going to iterate through set members or check if an item is a member (or not) of a set:

```
>>> 'k' in s
False
>>> 'k' in t
True
>>> 'c' not in t
True
>>> for i in s:
... print i
...
c
e
h
o
p
s
```

How to Update Sets

You can add and remove members to and from a set using various built-in methods and operators:

```
>>> s.add('z')
>>> s
set(['c', 'e', 'h', 'o', 'p', 's', 'z'])
>>> s.update('pypi')
>>> s
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y', 'z'])
>>> s.remove('z')
>>> s
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y'])
>>> s -= set('pypi')
>>> s
set(['c', 'e', 'h', 'o', 's'])
```

How to Remove Set Members and Sets

Removing sets themselves, like any Python object, you can let them go out of scope or explicitly remove them from the current namespace with del. If the reference count goes to zero, then it is tagged for garbage collection.

```
>>> del s
>>>
```

7.7 Set Type Operators

7.7.1 Standard Type Operators (all set types)

Membership (in, not in)

As for sequences, Python's in and not in operators are used to determine whether an element is (or is not) a member of a set.

```
>>> s = set('cheeseshop')
>>> t = frozenset('bookshop')
>>> 'k' in s
False
>>> 'k' in t
True
>>> 'c' not in t
True
```

Set Equality/Inequality

Equality (or inequality) may be checked between the same or different set types. Two sets are equal if and only if every member of each set is a member of the other. You can also say that each set must be a(n improper) subset of the other, e.g., both expressions s <= t and s

>= t are true, or (s <= t and s >= t) is True. Equality (or inequality) is independent of set type or ordering of members when the sets were created—it is all based on the set membership.

```
>>> s == t
False
>>> s != t
True
>>> u = frozenset(s)
>>> s == u
True
>>> set('posh') == set('shop')
True
```

Subset Of/Superset Of

Sets use the Python comparison operators to check whether sets are subsets or supersets of other sets. The "less than" symbols (<, <=) are used for subsets while the "greater than" symbols (>, >=) are used for supersets.

Sets support both proper (<) and improper (<=) subsets as well as proper (>) and improper (>=) supersets. A set is "less than" another set if and only if the first set is a proper subset of the second set (is a subset but not equal), and a set is "greater than" another set if and only if the first set is a proper superset of the second set (is a superset but not equal).

```
>>> set('shop') < set('cheeseshop')
True
>>> set('bookshop') >= set('shop')
True
```

7.7.2 Set Type Operators (All Set Types)

Union (|)

The union operation is practically equivalent to the OR (or inclusive disjunction) of sets. The union of two sets is another set where each element is a member of at least one of the sets, i.e., a member of one set or the other. The union symbol has a method equivalent, union().

```
>>> s | t
set(['c', 'b', 'e', 'h', 'k', 'o', 'p', 's'])
```

Intersection (&)

The intersection of two sets is another set where each element must be a member of at both sets, i.e., a member of one set and the other. The intersection symbol has a method equivalent, intersection().

```
>>> s & t
set(['h', 's', 'o', 'p']
```

Difference/Relative Complement (–)

The difference, or relative complement, between two sets is another set where each element is in one set but not the other. The difference symbol has a method equivalent, difference().

```
>>> s - t
set(['c', 'e'])
```

Symmetric Difference (^)

Similar to the other Boolean set operations, symmetric difference is the XOR (or exclusive disjunction) of sets. The symmetric difference between two sets is another set where each element is a member of one set but not the other. The symmetric difference symbol has a method equivalent,
symmetric_difference().

```
>>> s ^ t
set(['k', 'b', 'e', 'c'])
```

7.7.3 Set Type Operators (Mutable Sets Only)

(Union) Update (|=)

The update operation adds (possibly multiple) members from another set to the existing set. The method equivalent is update().

```
>>> s = set('cheeseshop')
>>> u = frozenset(s)
>>> s |= set('pypi')
>>> s
set(['c', 'e', 'i', 'h', 'o', 'p', 's', 'y'])
```

Retention/Intersection Update (&=)

The retention (or intersection update) operation keeps only the existing set members that are also elements of the other set. The method equivalent is intersection_update().

```
>>> s = set(u)
>>> s &= set('shop')
>>> s
set(['h', 's', 'o', 'p'])
```

Difference Update (–=)

The difference update operation returns a set whose elements are members of the original set after removing elements that are (also) members of the other set. The method equivalent is difference_update().

```
>>> s = set(u)
>>> s -= set('shop')
>>> s
set(['c', 'e'])
```

Symmetric Difference Update (^=)

The symmetric difference update operation returns a set whose members are either elements of the original or other set but not both. The method equivalent is symmetric_difference_update().

```
>>> s = set(u)
>>> t = frozenset('bookshop')
>>> s ^= t
>>> s
set(['c', 'b', 'e', 'k'])
```

7.8 Built-in Functions

7.8.1 Standard Type Functions

len()

The len() BIF for sets returns cardinality (or the number of elements) of the set passed in as the argument.

```
>>> s = set(u)
>>> s
set(['p', 'c', 'e', 'h', 's', 'o'])
>>> len(s)
6
```

7.8.2 Set Type Factory Functions

set() and frozenset()

The set() and frozenset() factory functions generate mutable and immutable sets, respectively. If no argument is provided, then an empty set is created. If one is provided, it must be an iterable, i.e., a sequence, an iterator, or an object that supports iteration such as a file or a dictionary.

```
>>> set()
set([])
>>> set([])
set([])
>>> set(())
set([])
>>> set('shop')
set(['h', 's', 'o', 'p'])
>>>
>>> frozenset(['foo', 'bar'])
frozenset(['foo', 'bar'])
```

7.9 Set Type Built-in Methods

7.9.1 Methods (All Set Types)

Table 7.4 Set Type Methods

Method Name	*Operation*
`s.issubset(t)`	Returns `True` if every member of *s* is in *t*, `False` otherwise
`s.issuperset(t)`	Returns `True` if every member of *t* is in *s*, `False` otherwise
`s.union(t)`	Returns a new set with the members of *s* or *t*
`s.intersection(t)`	Returns a new set with members of *s* and *t*
`s.difference(t)`	Returns a new set with members of *s* but not *t*
`s.symmetric_difference(t)`	Returns a new set with members of *s* or *t* but not both
`s.copy()`	Returns a new set that is a (shallow) copy of *s*

7.9.2 Methods (Mutable Sets Only)

Table 7.5 Mutable Set Type Methods

Method Name	*Operation*
`s.update(t)`	Updates *s* with elements added from *t*; in other words, *s* now has members of either *s* or *t*
`s.intersection_update(t)`	Updates *s* with members of both *s* and *t*
`s.difference_update(t)`	Updates *s* with members of *s* without elements of *t*
`s.symmetric_difference_update(t)`	Updates *s* with members of *s* or *t* but not both
`s.add(obj)`	Adds object *obj* to set *s*
`s.remove(obj)`	Removes object *obj* from set *s*; `KeyError` raised if *obj* is not an element of *s* (*obj* **not in** *s*)
`s.discard(obj)`	Removes object *obj* if *obj* is an element of *s* (*obj* **in** *s*)
`s.pop()`	Removes and returns an arbitrary object of *s*
`s.clear()`	Removes all elements from *s*