

Python 2.7 quick reference



John W. Shipman

2011-11-04 16:13

Abstract

A reference guide to most of the common features of the Python programming language, version 2.7.

This publication is available in Web form¹ and also as a PDF document². Please forward any comments to tcc-doc@nmt.edu.

Table of Contents

1. Introduction: What is Python?	5
2. Python 2.7 and Python 3.x	5
3. Starting Python	6
3.1. Using Python in Windows	6
3.2. Using Python in Linux	6
4. Line syntax	7
5. Names and keywords	7
6. Basic types	7
7. Numeric types	8
7.1. Type <code>int</code> : Integers	8
7.2. Type <code>long</code> : Extended-precision integers	9
7.3. Type <code>bool</code> : Boolean truth values	9
7.4. Type <code>float</code> : Floating-point numbers	10
7.5. Type <code>complex</code> : Imaginary numbers	10
8. Sequence types	11
8.1. Operations common to all the sequence types	12
9. Type <code>str</code> : Strings of 8-bit characters	14
9.1. String constants	14
9.2. Definition of "whitespace"	15
9.3. Methods on <code>str</code> values	16
9.4. The string <code>.format()</code> method	22
9.4.1. General form of a format code	23
9.4.2. The <i>name</i> part	24
9.4.3. The <i>conversion</i> part	25
9.4.4. The <i>spec</i> part	25
9.4.5. Formatting a field of variable length	29
9.5. The older string format operator	30
10. Type <code>unicode</code> : Strings of 32-bit characters	32
10.1. The UTF-8 encoding	33

¹ <http://www.nmt.edu/tcc/help/pubs/python27/web/>

² <http://www.nmt.edu/tcc/help/pubs/python27/python27.pdf>

11. Type <code>list</code> : Mutable sequences	35
11.1. Methods on lists	35
11.2. List comprehensions	38
12. Type <code>tuple</code> : Immutable sequences	39
13. The <code>bytes</code> type	40
13.1. Using the <code>bytes</code> type in 3.x conversion	41
14. The <code>bytearray</code> type	42
15. Types <code>set</code> and <code>frozenset</code> : Set types	43
15.1. Operations on mutable and immutable sets	44
15.2. Operations on mutable sets	47
16. Type <code>dict</code> : Dictionaries	49
16.1. Operations on dictionaries	49
16.2. Dictionary comprehensions	53
17. Type <code>file</code> : Input and output files	54
17.1. Methods on <code>file</code> objects	55
18. <code>None</code> : The special placeholder value	56
19. Operators and expressions	57
19.1. What is a predicate?	58
19.2. What is an iterable?	59
19.3. Duck typing, or: what is an interface?	59
19.4. What is the locale?	60
20. Basic functions	60
20.1. <code>abs()</code> : Absolute value	61
20.2. <code>all()</code> : Are all the elements of an iterable true?	61
20.3. <code>any()</code> : Are any of the members of an iterable true?	61
20.4. <code>bin()</code> : Convert to binary	61
20.5. <code>bool()</code> : Convert to Boolean	62
20.6. <code>bytearray()</code> : Create a byte array	62
20.7. <code>chr()</code> : Get the character with a given code	62
20.8. <code>cmp()</code> : Compare two values	63
20.9. <code>complex()</code> : Convert to <code>complex</code> type	63
20.10. <code>dict()</code> : Convert to a dictionary	64
20.11. <code>divmod()</code> : Quotient and remainder	64
20.12. <code>enumerate()</code> : Step through indices and values of an iterable	64
20.13. <code>file()</code> : Open a file	65
20.14. <code>filter()</code> : Extract qualifying elements from an iterable	65
20.15. <code>float()</code> : Convert to <code>float</code> type	66
20.16. <code>format()</code> : Format a value	66
20.17. <code>frozenset()</code> : Create a frozen set	66
20.18. <code>hex()</code> : Convert to base 16	67
20.19. <code>int()</code> : Convert to <code>int</code> type	67
20.20. <code>input()</code> : Read an expression from the user	68
20.21. <code>iter()</code> : Produce an iterator over a sequence	68
20.22. <code>len()</code> : Number of elements	69
20.23. <code>list()</code> : Convert to a list	69
20.24. <code>long()</code> : Convert to <code>long</code> type	69
20.25. <code>map()</code> : Apply a function to each element of an iterable	69
20.26. <code>max()</code> : Largest element of an iterable	70
20.27. <code>min()</code> : Smallest element of an iterable	71
20.28. <code>next()</code> : Call an iterator	71
20.29. <code>oct()</code> : Convert to base 8	72
20.30. <code>open()</code> : Open a file	72

20.31. <code>ord()</code> : Find the numeric code for a character	72
20.32. <code>pow()</code> : Exponentiation	72
20.33. <code>range()</code> : Generate an arithmetic progression as a list	73
20.34. <code>raw_input()</code> : Prompt and read a string from the user	73
20.35. <code>reduce()</code> : Sequence reduction	74
20.36. <code>reversed()</code> : Produce a reverse iterator	75
20.37. <code>round()</code> : Round to the nearest integral value	75
20.38. <code>set()</code> : Create an algebraic set	76
20.39. <code>sorted()</code> : Sort a sequence	76
20.40. <code>str()</code> : Convert to <code>str</code> type	77
20.41. <code>sum()</code> : Total the elements of a sequence	77
20.42. <code>tuple()</code> : Convert to a tuple	77
20.43. <code>type()</code> : Return a value's type	78
20.44. <code>unichr()</code> : Convert a numeric code to a Unicode character	78
20.45. <code>unicode()</code> : Convert to a Unicode string	78
20.46. <code>xrange()</code> : Arithmetic progression generator	78
20.47. <code>zip()</code> : Combine multiple sequences	79
21. Advanced functions	79
21.1. <code>basestring</code> : The string base class	79
21.2. <code>callable()</code> : Is this thing callable?	80
21.3. <code>classmethod()</code> : Create a class method	80
21.4. <code>delattr()</code> : Delete a named attribute	80
21.5. <code>dir()</code> : Display a namespace's names	81
21.6. <code>eval()</code> : Evaluate an expression in source form	82
21.7. <code>execfile()</code> : Execute a Python source file	82
21.8. <code>getattr()</code> : Retrieve an attribute of a given name	83
21.9. <code>globals()</code> : Dictionary of global name bindings	83
21.10. <code>hasattr()</code> : Does a value have an attribute of a given name?	84
21.11. <code>id()</code> : Unique identifier	84
21.12. <code>isinstance()</code> : Is a value an instance of some class or type?	84
21.13. <code>issubclass()</code> : Is a class a subclass of some other class?	85
21.14. <code>locals()</code> : Dictionary of local name bindings	86
21.15. <code>property()</code> : Create an access-controlled attribute	86
21.16. <code>reload()</code> : Reload a module	88
21.17. <code>repr()</code> : Representation	88
21.18. <code>setattr()</code> : Set an attribute	89
21.19. <code>slice()</code> : Create a slice instance	89
21.20. <code>staticmethod()</code> : Create a static method	90
21.21. <code>super()</code> : Superclass	90
21.22. <code>vars()</code> : Local variables	90
22. Simple statements	90
22.1. The assignment statement: <code>name = expression</code>	91
22.2. The <code>assert</code> statement: Verify preconditions	94
22.3. The <code>del</code> statement: Delete a name or part of a value	95
22.4. The <code>exec</code> statement: Execute Python source code	95
22.5. The <code>global</code> statement: Declare access to a global name	95
22.6. The <code>import</code> statement: Use a module	97
22.7. The <code>pass</code> statement: Do nothing	98
22.8. The <code>print</code> statement: Display output values	98
22.9. The <code>print()</code> function	98
23. Compound statements	99
23.1. Python's block structure	99

23.2. The break statement: Exit a for or while loop	100
23.3. The continue statement: Jump to the next cycle of a for or while	101
23.4. The for statement: Iteration over a sequence	101
23.5. The if statement: Conditional execution	102
23.6. The raise statement: Cause an exception	103
23.7. The return statement: Exit a function or method	105
23.8. The try statement: Anticipate exceptions	105
23.9. The with statement and context managers	107
23.10. The yield statement: Generate one result from a generator	108
24. def() : Defining your own functions	109
24.1. A function's local namespace	111
24.2. Iterators: Values that can produce a sequence of values	111
24.3. Generators: Functions that can produce a sequence of values	112
24.4. Decorators	113
25. Exceptions: Error signaling and handling	114
25.1. Definitions of exception terms	114
25.2. Life cycle of an exception	115
25.3. Built-in exceptions	116
26. Classes: Defining your own types	118
26.1. Old-style classes	121
26.1.1. Defining an old-style class	121
26.1.2. Instantiation of an old-style class: The constructor, <code>__init__()</code>	121
26.1.3. Attribute references in old-style classes	122
26.1.4. Method calls in an old-style class	123
26.1.5. Instance deletion: the destructor, <code>__del__()</code>	123
26.2. Life cycle of a new-style class	124
26.2.1. <code>__new__()</code> : New instance creation	124
26.2.2. Attribute access control in new-style classes	125
26.2.3. Properties in new-style classes: Fine-grained attribute access control	125
26.2.4. Conserving memory with <code>__slots__</code>	125
26.3. Special method names	126
26.3.1. Rich comparison methods	129
26.3.2. Special methods for binary operators	129
26.3.3. Unary operator special methods	130
26.3.4. Special methods to emulate built-in functions	130
26.3.5. <code>__call__()</code> : What to do when someone calls an instance	131
26.3.6. <code>__cmp__()</code> : Generalized comparison	131
26.3.7. <code>__contains__()</code> : The “in” and “not in” operators	132
26.3.8. <code>__del__()</code> : Destructor	132
26.3.9. <code>__delattr__()</code> : Delete an attribute	132
26.3.10. <code>__delitem__()</code> : Delete one item of a sequence	132
26.3.11. <code>__enter__</code> : Context manager initialization	133
26.3.12. <code>__exit__</code> : Context manager cleanup	133
26.3.13. <code>__format__</code> : Implement the <code>format()</code> function	133
26.3.14. <code>__getattr__()</code> : Handle a reference to an unknown attribute	134
26.3.15. <code>__getattribute__()</code> : Intercept all attribute references	134
26.3.16. <code>__getitem__()</code> : Get one item from a sequence or mapping	134
26.3.17. <code>__iter__()</code> : Create an iterator	134
26.3.18. <code>__nonzero__()</code> : True/false evaluation	135
26.3.19. <code>__repr__()</code> : String representation	135
26.3.20. <code>__reversed__()</code> : Implement the <code>reversed()</code> function	135
26.3.21. <code>__setattr__()</code> : Intercept all attribute changes	135
26.3.22. <code>__setitem__()</code> : Assign a value to one item of a sequence	135

26.4. Static methods	136
26.5. Class methods	136
27. <code>pdb</code> : The Python interactive debugger	137
27.1. Starting up <code>pdb</code>	137
27.2. Functions exported by <code>pdb</code>	137
27.3. Commands available in <code>pdb</code>	138
28. Commonly used modules	139
28.1. <code>math</code> : Common mathematical operations	139
28.2. <code>string</code> : Utility functions for strings	140
28.3. <code>random</code> : Random number generation	142
28.4. <code>time</code> : Clock and calendar functions	143
28.5. <code>re</code> : Regular expression pattern-matching	145
28.5.1. Characters in regular expressions	145
28.5.2. Functions in the <code>re</code> module	147
28.5.3. Compiled regular expression objects	148
28.5.4. Methods on a <code>MatchObject</code>	148
28.6. <code>sys</code> : Universal system interface	149
28.7. <code>os</code> : The operating system interface	150
28.8. <code>stat</code> : Interpretation of file status	152
28.9. <code>os.path</code> : File and directory interface	153

1. Introduction: What is Python?

Python is a recent, general-purpose, high-level programming language. It is freely available and runs pretty much everywhere.

- This document is a reference guide, not a tutorial. If you are new to Python programming, see *A Python programming tutorial*³.
- Complete documentation and free installs are available from the `python.org` homepage⁴.

This document does not describe every single feature of Python 2.7. A few interesting features that 99% of Python users will never need, such as metaclasses, are not described here. Refer to the official documentation for the full feature set.

2. Python 2.7 and Python 3.x

At this writing, both Python 2.7 and Python 3.2 are officially maintained implementations. The 3.0 release marked the first release in the development of Python that a new version was incompatible with the old one.

If you are using 2.x releases of Python, there is no hurry to convert to the 3.x series. Release 2.7 is guaranteed to be around for many years. Furthermore, there are tools to help you automate much of the conversion process. Notes throughout this document will discuss specific features of 2.7 that are intended to ease the transition.

- For a discussion of the changes between 2.7 and 3.0, see *What's New in Python*⁵.
- To see what changes must be made in your program to allow automatic conversion to Python 3.x, run Python with this flag:

³ <http://www.nmt.edu/tcc/help/pubs/lang/pytut/>

⁴ <http://www.python.org/>

⁵ <http://docs.python.org/whatsnew/>

```
python -3 yourprogram
```

- To convert your program to Python 3.x, first make a copy of the original program, then run this command:

```
python3-2to3 -w yourprogram
```

The `-w` flag replaces *yourprogram* with the converted 3.x version, and moves the original to "*yourprogram.bak*"

For full documentation of the Python 3.2 version, see the online documentation⁶.

3. Starting Python

You can use Python in two different ways:

- In "calculator" or "conversational mode", Python will prompt you for input with three greater-than signs (`>>>`). Type a line and Python will print the result. Here's an example:

```
>>> 2+2
4
>>> 1.0 / 7.0
0.14285714285714285
```

- You can also use Python to write a program, sometimes called a *script*.

3.1. Using Python in Windows

If you are using Python at the NM Tech Computer Center (TCC), you can get conversational mode from *Start* → *All Programs* → *ActiveState ActivePython 2.6* → *Python Interactive Shell*.

To write a program:

1. *Start* → *All Programs* → *ActiveState ActivePython 2.6* → *PythonWin Editor*.
2. Use *File* → *New*, select *Python Script* in the pop-up menu, and click *OK*. This will bring up an edit window.
3. Write your Python program in the edit window, then use *File* → *Save As...* to save it under some file name that ends in ".py". Use your **U:** drive. This drive is mounted everywhere at the TCC, and contains your personal files. It is backed up regularly.
4. To run your program, use *File* → *Run*. In the "Run Script" popup, enter the name of your program in the field labeled *Script File*, then click *OK*.

The output will appear in the "Interactive Window".

You may also run a Python script by double-clicking on it, provided that its name ends with ".py".

3.2. Using Python in Linux

To enter conversational mode on a Linux system, type this command:

```
python
```

⁶ <http://docs.python.org/py3k/>

Type *Control-D* to terminate the session.

If you write a Python script named *filename.py*, you can execute it using the command

```
python filename.py
```

Under Unix, you can also make a script self-executing by placing this line at the top:

```
#!/usr/bin/env python
```

You must also tell Linux that the file is executable by using the command “`chmod +x filename`”. For example, if your script is called *hello.py*, you would type this command:

```
chmod +x hello.py
```

4. Line syntax

The comment character is “`#`”; comments are terminated by end of line.

Long lines may be continued by ending the line with a backslash (`\`), but this is not necessary if there is at least one open “`(`”, “`[`”, or “`{`”.

5. Names and keywords

Python names (also called identifiers) can be any length and follow these rules:

- The first or only character must be a letter (uppercase or lowercase) or the underbar character, “`_`”.
- Any additional characters may be letters, underbars, or digits.

Examples: `coconuts`, `sirRobin`, `blanche_hickey_869`, `__secretWord`.

Case is significant in Python. The name “`Robin`” is not the same name as “`robin`”.

The names below are *keywords*, also known as reserved words. They have special meaning in Python and cannot be used as names or identifiers.

<code>and</code>	<code>def</code>	<code>finally</code>	<code>in</code>	<code>print</code>	<code>yield</code>
<code>as</code>	<code>del</code>	<code>for</code>	<code>is</code>	<code>raise</code>	
<code>assert</code>	<code>elif</code>	<code>from</code>	<code>lambda</code>	<code>return</code>	
<code>break</code>	<code>else</code>	<code>global</code>	<code>not</code>	<code>try</code>	
<code>class</code>	<code>except</code>	<code>if</code>	<code>or</code>	<code>with</code>	
<code>continue</code>	<code>exec</code>	<code>import</code>	<code>pass</code>	<code>while</code>	

6. Basic types

In programming, you manipulate *values* using *operators*. For example, in the expression “`1+2`”, the addition operator (`+`) is operating on the values 1 and 2 to produce the sum, 3. The Python operators are described in Section 19, “Operators and expressions” (p. 57), but let’s look first at Python’s way of representing values.

Every Python value must have a type. For example, the type of the whole number 1 is `int`, short for “integer.”

Here is a table summarizing most of the commonly-used Python types.

Table 1. Python's common types

Type name	Values	Examples
<code>int</code>	Integers in the range [-2147483648, 2147483647]. See Section 7.1, “Type <code>int</code> : Integers” (p. 8).	42, -3, 1000000
<code>long</code>	Integers of any size, limited only by the available memory. See Section 7.2, “Type <code>long</code> : Extended-precision integers” (p. 9).	42L, -3L, 1000000000000000L
<code>bool</code>	The two Boolean values <code>True</code> and <code>False</code> . See Section 7.3, “Type <code>bool</code> : Boolean truth values” (p. 9).	<code>True</code> , <code>False</code>
<code>float</code>	Floating-point numbers; see Section 7.4, “Type <code>float</code> : Floating-point numbers” (p. 10).	3.14159, -1.0, 6.0235e23
<code>complex</code>	Complex numbers. If the idea of computing with the square root of -1 bothers you, just ignore this type, otherwise see Section 7.5, “Type <code>complex</code> : Imaginary numbers” (p. 10).	(3.2+4.9j), (0+3.42e-3j)
<code>str</code>	Strings of 8-bit characters; see Section 9, “Type <code>str</code> : Strings of 8-bit characters” (p. 14). Strings can be empty: write such as a string as <code>""</code> or <code>''</code> .	'Sir Robin', "xyz", "I'd've"
<code>unicode</code>	Strings of 32-bit Unicode characters; see Section 10, “Type <code>unicode</code> : Strings of 32-bit characters” (p. 32).	u'Fred', u'\u03fa'
<code>list</code>	A mutable sequence of values; see Section 11, “Type <code>list</code> : Mutable sequences” (p. 35).	['dot', 'dash']; []
<code>tuple</code>	An immutable sequence of values; see Section 12, “Type <code>tuple</code> : Immutable sequences” (p. 39).	('dot', 'dash'); (); ("singleton",)
<code>dict</code>	Use <code>dict</code> values (dictionaries) to structure data as look-up tables; see Section 16, “Type <code>dict</code> : Dictionaries” (p. 49).	{'go':1, 'stop':2}; {}
<code>bytearray</code>	A mutable sequence of 8-bit bytes; see Section 14, “The <code>bytearray</code> type” (p. 42).	<code>bytearray('Bletchley')</code>
<code>file</code>	A file being read or written; see Section 17, “Type <code>file</code> : Input and output files” (p. 54).	<code>open('/etc/motd')</code>
<code>None</code>	A special, unique value that may be used where a value is required but there is no obvious value. See Section 18, “None: The special placeholder value” (p. 56).	<code>None</code>

7. Numeric types

Python has a number of different types used for representing numbers.

7.1. Type `int`: Integers

Python values of type `int` represent integers, that is, whole numbers in the range $[-2^{31}, 2^{31}-1]$, roughly plus or minus two billion.

You can represent a value in octal (base 8) by preceding it with “0o”. Similarly, use a leading “0x” to represent a value in hexadecimal (base 16), or “0b” for binary. Examples in conversational mode:

```
>>> 999+1
1000
>>> 0o77
63
>>> 0xff
255
>>> 0b1001
9
```

Note

The 0o and 0b prefixes work only in Python versions 2.6 and later. In 2.5 and earlier versions, any number starting with “0” was considered to be octal. This functionality is retained in the 2.6+ versions, but will not work in the Python 3.x versions.

To convert other numbers or character strings to type `int`, see Section 20.19, “`int()`: Convert to `int` type” (p. 67).

If you perform operations on `int` values that result in numbers that are too large, Python automatically converts them to `long` type; see Section 7.2, “Type `long`: Extended-precision integers” (p. 9).

7.2. Type `long`: Extended-precision integers

Values of `long` type represent whole numbers, but they may have many more than the nine or ten digits allowed by `int` type. In practice, the number of digits in a `long` value is limited only by processor memory size.

To write a `long`-type constant, use the same syntax as for `int`-type constants, but place a letter `L` immediately after the last digit. Also, if an operation on `int` values results in a number too large to represent as an `int`, Python will automatically converted it to type `long`.

```
>>> 100 * 100
10000
>>> 100L * 100L
10000L
>>> 10000000000*10000000000
10000000000000000000L
>>> 0xffffL
65535L
```

To convert a value of a different numeric type or a string of characters to a `long` value, see Section 20.24, “`long()`: Convert to `long` type” (p. 69).

7.3. Type `bool`: Boolean truth values

A value of `bool` type represents a Boolean (true or false) value. There are only two values, written in Python as “True” and “False”.

Internally, `True` is represented as 1 and `False` as 0, and they can be used in numeric expressions as those values.

Here's an example. In Python, the expression “ $a < b$ ” compares two values a and b , and returns `True` if a is less than b , `False` if a is greater than or equal to b .

```
>>> 2 < 3
True
>>> 3 < 2
False
>>> True+4
5
>>> False * False
0
```

These values are considered `False` wherever true/false values are expected, such as in an `if` statement:

- The `bool` value `False`.
- Any numeric zero: the `int` value `0`, the `float` value `0.0`, the `long` value `0L`, or the `complex` value `0.0j`.
- Any empty sequence: the `str` value `''`, the `unicode` value `u''`, the empty `list` value `[]`, or the empty `tuple` value `()`.
- Any empty mapping, such as the empty `dict` (dictionary) value `{}`.
- The special value `None`.

All other values are considered `True`. To convert any value to a Boolean, see Section 20.5, “`bool()`: Convert to Boolean” (p. 62).

7.4. Type `float`: Floating-point numbers

Values of this type represent real numbers, with the usual limitations of IEEE-754 floating point type: it cannot represent very large or very small numbers, and the precision is limited to only about 15 digits. For complete details on the IEEE-754 standard and its limitations, see the Wikipedia article⁷.

A floating-point constant may be preceded by a “+” or “-” sign, followed by a string of one or more digits containing a decimal point (“.”).

For very large or small numbers, you may express the number in exponential notation by appending a letter “e” followed by a power of ten (which may be preceded by a sign).

For example, Avogadro's Number gives the number of atoms of carbon in 12 grams of carbon¹², and is written as 6.0221418×10^{23} . In Python that would be “`6.0221418e23`”.

Please note that calculations involving `float` type are approximations. In calculator mode, Python will display the numbers to their full precision, so you may see a number that is very close to what you expect, but not exact.

```
>>> 1.0/7.0
0.14285714285714285
>>> -2*-4.2e37
8.4000000000000004e+37
```

7.5. Type `complex`: Imaginary numbers

Mathematically, a complex number is a number of the form $A+Bi$ where i is the imaginary number, equal to the square root of -1.

⁷ http://en.wikipedia.org/wiki/IEEE_754-1985

Complex numbers are quite commonly used in electrical engineering. In that field, however, because the symbol i is used to represent current, they use the symbol j for the square root of -1. Python adheres to this convention: a number followed by “j” is treated as an imaginary number. Python displays complex numbers in parentheses when they have a nonzero real part.

```
>>> 5j
5j
>>> 1+2.56j
(1+2.5600000000000001j)
>>> (1+2.56j)*(-1-3.44j)
(7.8064-6j)
```

Unlike Python's other numeric types, complex numbers are a composite quantity made of two parts: the real part and the imaginary part, both of which are represented internally as `float` values. You can retrieve the two components using attribute references. For a complex number `C`:

- `C.real` is the real part.
- `C.imag` is the imaginary part as a `float`, not as a `complex` value.

```
>>> a=(1+2.56j)*(-1-3.44j)
>>> a
(7.8064-6j)
>>> a.real
7.8064
>>> a.imag
-6.0
```

To construct a `complex` value from two `float` values, see Section 20.9, “`complex()`: Convert to complex type” (p. 63).

8. Sequence types

The next four types described (`str`, `unicode`, `list` and `tuple`) are collectively referred to as *sequence* types.

Each sequence value represents an ordered set in the mathematical sense, that is, a collection of things in a specific order.

Python distinguishes between *mutable* and *immutable* sequences:

- An immutable sequence can be created or destroyed, but the number, sequence, and values of its elements cannot change.
- The values of a mutable sequence can be changed. Any element can be replaced or deleted, and new elements can be added at the beginning, the end, or in the middle.

There are four sequence types, but they share most of the same operations.

- Section 8.1, “Operations common to all the sequence types” (p. 12).
- Section 9, “Type `str`: Strings of 8-bit characters” (p. 14) (immutable).
- Section 10, “Type `unicode`: Strings of 32-bit characters” (p. 32) (immutable).
- Section 11, “Type `list`: Mutable sequences” (p. 35) (mutable).
- Section 12, “Type `tuple`: Immutable sequences” (p. 39) (immutable).

8.1. Operations common to all the sequence types

These functions work on values of the four sequence types: `int`, `unicode`, `tuple`, and `list`.

- Section 20.22, “`len()`: Number of elements” (p. 69).
- Section 20.26, “`max()`: Largest element of an iterable” (p. 70).
- Section 20.27, “`min()`: Smallest element of an iterable” (p. 71).

These operators apply to sequences.

S_1+S_2

Concatenation—for two sequences S_1 and S_2 of the same type, a new sequence containing all the elements from S_1 followed by all the elements of S_2 .

```
>>> "vi" + "car"
'vicar'
>>> [1,2,3]+[5,7,11,13]+[15]
[1, 2, 3, 5, 7, 11, 13, 15]
>>> ('roy', 'g')+('biv',)
('roy', 'g', 'biv')
```

$S*n$

For a sequence S and a positive integer n , the result is a new sequence containing all the elements of S repeated n times.

```
>>> 'worra'*8
'worrworraworrworraworrworraworrworra'
>>> [0]*4
[0, 0, 0, 0]
>>> (True, False)*5
(True, False, True, False, True, False, True, False, True, False)
```

$x \text{ in } S$

Is any element of a sequence S equal to x ?

For convenience in searching for substrings, if the sequence to be searched is a string, the x operand can be a multi-character string. In that case, the operation returns `True` if x is found anywhere in S .

```
>>> 1 in [2,4,6,0,8,0]
False
>>> 0 in [2,4,6,0,8,0]
True
>>> 'a' in 'banana'
True
>>> 3.0 in (2.5, 3.0, 3.5)
True
>>> "baz" in "rowrbazzle"
True
```

$x \text{ not in } S$

Are all the elements of a sequence S not equal to x ?

```
>>> 'a' not in 'banana'
False
```

```
>>> 'x' not in 'banana'
True
```

S[i]

Subscripting: retrieve the *i*th element of *S*, *counting from zero*. If *i* is greater than or equal to the number of elements of *S*, an `IndexError` exception is raised.

```
>>> 'Perth'[0]
'p'
>>> 'Perth'[1]
'e'
>>> 'Perth'[4]
'h'
>>> 'Perth'[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> ('red', 'yellow', 'green')[2]
'green'
```

S[i:j]

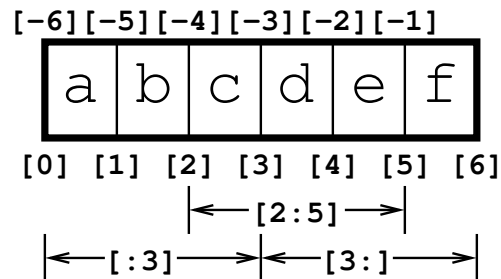
Slicing: For a sequence *S* and two integers *i* and *j*, return a new sequence with copies of the elements of *S* between *positions i* and *j*.

The values used in slicing refer to the positions *between* elements, where position zero is the position *before* the first element; position 1 is between the first and second element; and so on.

You can also specify positions relative to the end of a sequence. Position -1 is the position before the last element; -2 is the position before the second-to-last element; and so on.

You can omit the starting position to obtain a slice starting at the beginning. You can omit the ending position to get all the elements through the last.

For example, here is a diagram showing three slices of the string 'abcdef'.



```
>>> 'abcdef'[2:5]
'cde'
>>> 'abcdef'[:3]
'abc'
>>> 'abcdef'[3:]
'def'
>>> (90, 91, 92, 93, 94, 95)[2:5]
(92, 93, 94)
```

S[i:j:k]

You can use a slice expression like this to select every *k*th element. Examples:

```

>>> teens = range(13,20)
>>> teens
[13, 14, 15, 16, 17, 18, 19]
>>> teens[::2]
[13, 15, 17, 19]
>>> teens[1::2]
[14, 16, 18]
>>> teens[1:5]
[14, 15, 16, 17]
>>> teens[1:5:2]
[14, 16]

```

9. Type `str`: Strings of 8-bit characters

Python has two string types. Type `str` holds strings of zero or more 8-bit characters, while `unicode` strings provide full support of the expanded Unicode character set; see Section 10, “Type `unicode`: Strings of 32-bit characters” (p. 32).

In addition to the functions described in Section 8.1, “Operations common to all the sequence types” (p. 12), these functions apply to strings:

- Section 20.34, “`raw_input()`: Prompt and read a string from the user” (p. 73).
- Section 20.40, “`str()`: Convert to `str` type” (p. 77).
- Section 20.45, “`unicode()`: Convert to a Unicode string” (p. 78).

9.1. String constants

There are many forms for string constants:

- `'...'`: You may enclose the string in single-quotes.
- `"..."`: You may instead enclose it in double-quotes. Internally, there is absolutely no difference. To include a double-quote character inside the string, use the escape sequence `"\""`.

In conversational mode, Python will generally display values using single-quotes. If the string contains single-quotes but no double-quotes, Python will display the value using double-quotes. If the string contains both, the value will be displayed in single-quotes, with single-quote characters inside the value displayed as the escape sequence `"\""`.

- `'''...'''`: You may enclose your string between three single quotes in a row. The difference is that you can continue such a string over multiple lines, and the line breaks will be included in the string as newline characters.
- `"""..."""`: You can use three sets of double quotes. As with three sets of single quotes, line breaks are allowed and preserved as `"\n"` characters. If you use these triply-quoted strings in conversational mode, continuation lines will prompt you with `...` .

```

>>> 'Penguin'
'Penguin'
>>> "ha'penny"
"ha'penny"
>>> "Single ' and double\" quotes"
'Single \' and double" quotes'
>>> '''
'''

```

```

''
>>> ""
''
>>> s=''This string
... contains two lines.
>>> t=""This string
... contains
... three lines.

```

In addition, you can use any of these *escape sequences* inside a string constant (see Wikipedia⁸ for more information on the ASCII code).

Table 2. String escape sequences

<code>\newline</code>	A backslash at the end of a line is ignored.
<code>\\</code>	Backslash (\)
<code>\'</code>	Closing single quote (')
<code>\"</code>	Double-quote character (")
<code>\n</code>	Newline (ASCII LF or linefeed)
<code>\b</code>	Backspace (in ASCII, the BS character)
<code>\f</code>	Formfeed (ASCII FF)
<code>\r</code>	Carriage return (ASCII CR)
<code>\t</code>	Horizontal tab (ASCII HT)
<code>\v</code>	Vertical tab (ASCII VT)
<code>\ooo</code>	The character with octal code <i>ooo</i> , e.g., <code>'\177'</code> .
<code>\xhh</code>	The character with hexadecimal value <i>hh</i> , e.g., <code>'\xFF'</code> .

Raw strings: If you need to use a lot of backslashes inside a string constant, and doubling them is too confusing, you can prefix any string with the letter `r` to suppress the interpretation of escape sequences. For example, `'\\'` contains two backslashes, but `r'\\'` contains four. Raw strings are particularly useful with Section 28.5, “re: Regular expression pattern-matching” (p. 145).

9.2. Definition of “whitespace”

In Python, these characters are considered whitespace:

Escape sequence	ASCII ⁹ name	English name
<code>' '</code>	SP	space
<code>'\n'</code>	NL	newline
<code>'\r'</code>	CR	carriage return
<code>'\t'</code>	HT	horizontal tab
<code>'\f'</code>	FF	form feed
<code>'\v'</code>	VT	vertical tab

⁸ <http://en.wikipedia.org/wiki/ASCII>

⁹ <http://en.wikipedia.org/wiki/ASCII>

9.3. Methods on str values

These methods are available on any string value *S*.

S.capitalize()

Return *S* with its first character capitalized (if a letter).

```
>>> 'e e cummings'.capitalize()
'E e cummings'
>>> '---abc---'.capitalize()
'---abc---'
```

S.center(*w*)

Return *S* centered in a string of width *w*, padded with spaces. If $w \leq \text{len}(S)$, the result is a copy of *S*. If the number of spaces of padding is odd, the extra space will be placed after the centered value. Example:

```
>>> 'x'.center(4)
' x '
```

S.count(*t* [, *start* [, *end*]])

Return the number of times string *t* occurs in *S*. To search only a slice *S*[*start*:*end*] of *S*, supply *start* and *end* arguments.

```
>>> 'banana'.count('a')
3
>>> 'bananana'.count('na')
3
>>> 'banana'.count('a', 3)
2
>>> 'banana'.count('a', 3, 5)
1
```

S.decode (*encoding*)

If *S* contains an encoded Unicode string, this method will return the corresponding value as `unicode` type. The *encoding* argument specifies which decoder to use; typically this will be the string `'utf_8'` for the UTF-8 encoding. For discussion and examples, see Section 10.1, “The UTF-8 encoding” (p. 33).

S.endswith(*t* [, *start* [, *end*]])

Predicate to test whether *S* ends with string *t*. If you supply the optional *start* and *end* arguments, it tests whether the slice *S*[*start*:*end*] ends with *t*.

```
>>> 'bishop'.endswith('shop')
True
>>> 'bishop'.endswith('bath and wells')
False
>>> 'bishop'[3:5]
'ho'
>>> 'bishop'.endswith('o', 3, 5)
True
```

S.expandtabs([*tabsize*])

Returns a copy of *S* with all tabs replaced by one or more spaces. Each tab is interpreted as a request to move to the next “tab stop”. The optional *tabsize* argument specifies the number of spaces between tab stops; the default is 8.

Here is how the function actually works. The characters of *S* are copied to a new string *T* one at a time. If the character is a tab, it is replaced by enough tabs so the new length of *T* is a multiple of the tab size (but always at least one space).

```
>>> 'X\tY\tZ'.expandtabs()
'X     Y     Z'
>>> 'X\tY\tZ'.expandtabs(4)
'X    Y    Z'
>>> 'a\tbb\tccc\tddd\ttttt\ttttt\ttttt'.expandtabs(4)
'a   bb  ccc ddd   ttttt  ttttt'
```

S.find(*t* [, *start* [, *end*]])

If string *t* is not found in *S*, return -1; otherwise return the index of the first position in *S* that matches *t*.

The optional *start* and *end* arguments restrict the search to slice *S*[*start*:*end*].

```
>>> 'banana'.find('an')
1
>>> 'banana'.find('ape')
-1
>>> 'banana'.find('n', 3)
4
>>> 'council'.find('c', 1, 4)
-1
```

.format(p*, ***kw*)**

See Section 9.4, “The string .format() method” (p. 22).

S.index(*t* [, *start* [, *end*]])

Works like .find(), but if *t* is not found, it raises a `ValueError` exception.

```
>>> 'council'.index('co')
0
>>> 'council'.index('phd')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

S.isalnum()

Predicate that tests whether *S* is nonempty and all its characters are alphanumeric.

```
>>> ''.isalnum()
False
>>> 'abc123'.isalnum()
True
>>> '&*$#&*( )abc123'.isalnum()
False
```

S.isalpha()

Predicate that tests whether *S* is nonempty and all its characters are letters.

```
>>> 'abc123'.isalpha()
False
>>> 'MaryRecruiting'.isalpha()
True
```

```
>>> ''.isalpha()
False
```

S.isdigit()

Predicate that tests whether *S* is nonempty and all its characters are digits.

```
>>> 'abc123'.isdigit()
False
>>> ''.isdigit()
False
>>> '2415'.isdigit()
True
```

S.islower()

Predicate that tests whether *S* is nonempty and all its letters are lowercase (non-letter characters are ignored).

```
>>> ''.islower()
False
>>> 'abc123'.islower()
True
>>> 'ABC123'.islower()
False
```

S.isspace()

Predicate that tests whether *S* is nonempty and all its characters are whitespace characters.

```
>>> ''.isspace()
False
>>> '\t\n\r'.isspace()
True
>>> 'killer \t \n rabbit'.isspace()
False
```

S.istitle()

A predicate that tests whether *S* has “title case”. In a title-cased string, uppercase characters may appear only at the beginning of the string or after some character that is not a letter. Lowercase characters may appear only after an uppercase letter.

```
>>> 'abc def GHI'.istitle()
False
>>> 'Abc Def Ghi'.istitle()
True
```

S.isupper()

Predicate that tests whether *S* is nonempty and all its letters are uppercase letters (non-letter characters are ignored).

```
>>> 'abcDEF'.isupper()
False
>>> '123GHI'.isupper()
True
>>> ''.isupper()
False
```

S.join(L)

L must be an iterable that produces a sequence of strings. The returned value is a string containing the members of the sequence with copies of the delimiter string *S* inserted between them.

One quite common operation is to use the empty string as the delimiter to concatenate the elements of a sequence.

Examples:

```
>>> '/' .join(['never', 'pay', 'plan'])
'never/pay/plan'
>>> '(***)'.join ( ('Property', 'of', 'the', 'zoo') )
'Property(***)of(***)the(***)zoo'
>>> '' .join(['anti', 'dis', 'establish', 'ment', 'arian', 'ism'])
'antidisestablishmentarianism'
```

S.ljust(w)

Return a copy of *S* left-justified in a field of width *w*, padded with spaces. If $w \leq \text{len}(S)$, the result is a copy of *S*.

```
>>> "Ni".ljust(4)
'Ni  '
```

S.lower()

Returns a copy of *S* with all uppercase letters replaced by their lowercase equivalent.

```
>>> "I like SHOUTING!".lower()
'i like shouting!'
```

S.lstrip([c])

Return *S* with all leading characters from string *C* removed. The default value for *C* is a string containing all the whitespace characters.

```
>>> ' \t \n Run \t \n away ! \n \t ' .rstrip()
'Run \t \n away ! \n \t '
'***Done***'.rstrip('*')
'Done***'
>>> "(*)(*)(*Undone*)".rstrip ( " )(*" )
'Undone*')
```

S.partition(d)

Searches string *S* for the first occurrence of some delimiter string *d*. If *S* contains the delimiter, it returns a tuple (*pre*, *d*, *post*), where *pre* is the part of *S* before the delimiter, *d* is the delimiter itself, and *post* is the part of *S* after the delimiter.

If the delimiter is not found, this method returns a 3-tuple (*S*, '', '').

```
>>> "Daffy English kniggets!".partition(' ')
('Daffy', ' ', 'English kniggets!')
>>> "Daffy English kniggets!".partition('/')
('Daffy English kniggets!', '/', '')
>>> "a*b***c*d".partition("***")
('a*b', '***', '*c*d')
```

S.replace(*old*, *new*[, *max*])

Return a copy of *S* with all occurrences of string *old* replaced by string *new*. Normally, all occurrences are replaced; if you want to limit the number of replacements, pass that limit as the *max* argument.

```
>>> 'Frenetic'.replace('e', 'x')
'Frnxxtic'
>>> 'Frenetic'.replace('e', '###')
'Fr###n###tic'
>>> 'banana'.replace('an', 'erzerk')
'berzerkerzerka'
>>> 'banana'.replace('a', 'x', 2)
'bxnxna'
```

S.rfind(*t*[, *start*[, *end*]])

Like `.find()`, but if *t* occurs in *S*, this method returns the *highest* starting index.

```
>>> 'banana'.find('a')
1
>>> 'banana'.rfind('a')
5
```

S.rindex(*t*[, *start*[, *end*]])

Similar to `S.index()`, but it returns the *last* index in *S* where string *t* is found. It will raise a `ValueError` exception if the string is not found.

```
>>> "Just a flesh wound.".index('s')
2
>>> "Just a flesh wound.".rindex('s')
10
>>> "Just a flesh wound.".rindex('xx')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: substring not found
```

S.rjust(*w*[, *fill*])

Return a copy of *S* right-justified in a field of width *w*, padded with spaces. If $w \leq \text{len}(S)$, the result is a copy of *S*.

To pad values with some character other than a space, pass that character as the optional second argument.

```
>>> '123'.rjust(5)
'  123'
>>> '123'.rjust(5, '*')
'**123'
```

S.rpartition(*d*)

Similar to `S.partition()`, except that it finds the *last* occurrence of the delimiter.

```
>>> "Daffy English kniggets!".rpartition(' ')
('Daffy English', ' ', 'kniggets!')
>>> "a*b***c*d".rpartition("***")
('a*b*', '***', 'c*d')
```

S.rsplit(*d* [, *max*])

Similar to `S.split(d [, max])`, except that if there are more fields than *max*, the split fields are taken from the end of the string instead of from the beginning.

```
>>> "I am Zoot's identical twin sister, Dingo.".rsplit(None, 2)
["I am Zoot's identical twin", 'sister,', 'Dingo.']
```

S.rstrip(*c*)

Return *S* with all trailing characters from string *c* removed. The default value for *c* is a string containing all the whitespace characters.

```
>>> ' \t \n Run \t \n away ! \n \t '.rstrip()
' \t \n Run \t \n away !'
```

S.split(*d* [, *max*])

Returns a list of strings [*s*₀, *s*₁, ...] made by splitting *S* into pieces wherever the delimiter string *d* is found. The default is to split up *S* into pieces wherever clumps of one or more whitespace characters are found.

```
>>> "I'd annex \t \r the Sudetenland" .split()
["I'd", 'annex', 'the', 'Sudetenland']
>>> '3/crunchy frog/ Bath & Wells'.split('/')
['3', 'crunchy frog', ' Bath & Wells']
>>> '//Norwegian Blue/'.split('/')
['', '', 'Norwegian Blue', '']
>>> 'never<*>pay<*>plan<*>'.split('<*>')
['never', 'pay', 'plan', '']
```

The optional *max* argument limits the number of pieces removed from the front of *S*. The resulting list will have no more than *max*+1 elements.

To use the *max* argument while splitting the string on clumps of whitespace, pass `None` as the first argument.

```
>>> 'a/b/c/d/e'.split('/', 2)
['a', 'b', 'c/d/e']
>>> 'a/b'.split('/', 2)
['a', 'b']
>>> "I am Zoot's identical twin sister, Dingo.".split(None, 2)
['I', 'am', "Zoot's identical twin sister, Dingo."]
```

S.splitlines(*keepends*)

Splits *S* into lines and returns a list of the lines as strings. Discards the line separators unless the optional *keepends* argument is true.

```
>>> """Is that
... an ocarina?""" .splitlines()
['Is that', 'an ocarina?']
>>> """What is your name?
... Sir Robin of Camelot.""" .splitlines(True)
['What is your name?\n', 'Sir Robin of Camelot.']
```

S.startswith(*t* [, *start* [, *end*]])

Predicate to test whether *S* starts with string *t*. Otherwise similar to `.endswith()`.

```
>>> "bishop".startswith('bish')
True
>>> "bishop".startswith('The')
False
```

S.strip([c])

Return *S* with all leading and trailing characters from string *c* removed. The default value for *c* is a string containing all the whitespace characters.

```
>>> ' \t \n Run \t \n away ! \n \t '.strip()
'Run \t \n away !'
```

S.swapcase()

Return a copy of *S* with each lowercase character replaced by its uppercase equivalent, and vice versa.

```
>>> "abcDEF".swapcase()
'ABCdef'
```

S.title()

Returns the characters of *S*, except that the first letter of each word is uppercased, and other letters are lowercased.

```
>>> "huge...tracts of land".title()
'Huge...Tracts Of Land'
```

S.translate(new[, drop])

This function is used to translate or remove each character of *S*. The *new* argument is a string of exactly 256 characters, and each character *x* of the result is replaced by `new[ord(x)]`.

If you would like certain characters removed from *S* before the translation, provide a string of those characters as the *drop* argument.

For your convenience in building the special 256-character strings used here, see the definition of the `maketrans()` function of Section 28.2, “string: Utility functions for strings” (p. 140), where you will find examples.

S.upper()

Return a copy of *S* with all lowercase characters replaced by their uppercase equivalents.

```
>>> 'I like shouting'.upper()
'I LIKE SHOUTING'
```

S.zfill(w)

Return a copy of *S* left-filled with '0' characters to width *w*.

```
>>> '12'.zfill(9)
'000000012'
```

9.4. The string .format() method

The `.format()` method of the `str` type is an extremely convenient way to format text exactly the way you want it.

Note

This method was added in Python 2.6.

Quite often, we want to embed data values in some explanatory text. For example, if we are displaying the number of nematodes in a hectare, it is a lot more meaningful to display it as "There were 37.9 nematodes per hectare" than just "37.9". So what we need is a way to mix constant text like "nematodes per hectare" with values from elsewhere in your program.

Here is the general form:

```
template.format( $p_0$ ,  $p_1$ , ...,  $k_0=v_0$ ,  $k_1=v_1$ , ...)
```

The *template* is a string containing a mixture of one or more *format codes* embedded in constant text. The `format` method uses its arguments to substitute an appropriate value for each format code in the template.

The arguments to the `.format()` method are of two types. The list starts with zero or more positional arguments p_i , followed by zero or more keyword arguments of the form $k_i=v_i$, where each k_i is a name with an associated value v_i .

Just to give you the general flavor of how this works, here's a simple conversational example. In this example, the format code "{0}" is replaced by the first positional argument (49), and "{1}" is replaced by the second positional argument, the string "okra".

```
>>> "We have {0} hectares planted to {1}.".format(49, "okra")
'We have 49 hectares planted to okra.'
>>>
```

In the next example, we supply the values using keyword arguments. The arguments may be supplied in any order. The keyword names must be valid Python names (see Section 5, "Names and keywords" (p. 7)).

```
>>> "{monster} has now eaten {city}.".format(
...     city='Tokyo', monster='Mothra')
'Mothra has now eaten Tokyo'
```

You may mix references to positional and keyword arguments:

```
>>> "The {structure} sank {0} times in {1} years.".format(
...     3, 2, structure='castle')
'The castle sank 3 times in 2 years.'
```

If you need to include actual "{" and "}" characters in the result, double them, like this:

```
>>> "There are {0} members in set {{a}}.".format(15)
'There are 15 members in set {a}.'
```

9.4.1. General form of a format code

Here is the general form of a format code, where optional parts in [brackets], and actual characters are in "double quotes":

```
"{" [name] ["!" conversion] [":" spec] "}"
```

- For the *name* portion, see Section 9.4.2, "The *name* part" (p. 24).

- For the *conversion* part, see Section 9.4.3, “The *conversion* part” (p. 25).
- For the *spec* part, see Section 9.4.4, “The *spec* part” (p. 25).

9.4.2. The *name* part

The *name* part of a format code specifies the source of the value to be formatted here. Numbers refer to positional arguments passed to the `.format()` method, starting at 0 for the first argument. You may also use any Python name to refer to one of the keyword arguments.

- If the associated argument is an iterable, you may append an expression of this form to retrieve one of its elements:

```
"[" index "]"
```

For example:

```
>>> signal=['red', 'yellow', 'green']
>>> signal[2]
'green'
>>> "The light is {0[2]}!".format(signal)
'The light is green!'
```

- If the associated argument has attributes, you can append an expression of this form to refer to that attribute:

```
"." name
```

For example:

```
>>> import string
>>> string.digits
'0123456789'
>>> "Our digits are '{s.digits}'".format(s=string)
"Our digits are '0123456789'."
```

In general, you can use any combination of these features. For example:

```
>>> "The sixth digit is '{s.digits[5]}'.format(s=string)
"The sixth digit is '5'"
```

Starting with Python 2.7, you may omit all of the numbers that refer to positional arguments, and they will be used in the sequence they occur. For example:

```
>>> "The date is {}-{}-{}".format(2012, 5, 1)
'The date is 2012-5-1.'
```

If you use this convention, you must omit all those numbers. You can, however, omit all the numbers and still use the keyword names feature:

```
>>> "Can I have {} pounds to {excuse}?".format(
...     50, excuse='mend the shed')
'Can I have 50 pounds to mend the shed?'
```

9.4.3. The *conversion* part

Following the *name* part of a format code, you can use one of these two forms to force the value to be converted by a standard function:

!s	str()
!r	repr()

Here's an example:

```
>>> "{}".format('Don\t')
'Don\t'
>>> "{!r}".format('Don\t')
'"Don\t"'
```

9.4.4. The *spec* part

After the *name* and *conversion* parts of a format code, you may use a colon (":") and a format specifier to supply more details about how to format the related value.

Here is the general form of a format specifier.

```
":" [[fill] align] [sign] ["#"] ["0"] [width] [","] [". " prec] [type]
```

fill

You may specify any fill character except “””. This character is used to pad a short value to the specified length. It may be specified only in combination with an *align* character.

align

Specifies how to align values that are not long enough to occupy the specified length. There are four values:

<	Left-justify the value. This is the default alignment for string values.
>	Right-justify the value. This is the default alignment for numbers.
^	Center the value.
=	For numbers using a <i>sign</i> specifier, add the padding between the sign and the rest of the value.

Here are some examples of the use of *fill* and *align*.

```
>>> "{:>8}".format(13)
'      13'
>>> "{:>8}".format('abc')
'      abc'
>>> "{:*>8}".format('abc')
'*****abc'
>>> "{:*<8}".format('abc')
'abc*****'
>>> "{:>5d}".format(14)
'     14'
>>> "{:#>5d}".format(14)
'###14'
>>> "{:<6}".format('Git')
'Git  '
```

```
>>> "{:*<6}".format('Git')
'Git***'
>>> "{:=^8}".format('Git')
'==Git==='
>>> "{:=-9d}".format(-3)
'_.*****3'
```

sign

This option controls whether an arithmetic sign is displayed. There are three possible values:

+	Always display a sign: + for positive, - for negative.
-	Display - only for negative values.
(one space)	Display one space for positive values, - for negative.

Here are some examples of use of the sign options.

```
>>> '{} {}'.format(17, -17)
'17 -17'
>>> '{:5} {:5}'.format(17, -17)
' 17  -17'
>>> '{:<5} {:<5}'.format(17, -17)
'17  -17 '
>>> '{:@<5} {:@<5}'.format(17, -17)
'17@@@ -17@'
>>> '{:@>5} {:@>5}'.format(17, -17)
'@@@17 @@-17'
>>> '{:@^5} {:@^5}'.format(17, -17)
'@17@@ @-17@'
>>> '{:@+5} {:@+5}'.format(17, -17)
'@+17@ @-17@'
>>> '{:@^-5} {:@^-5}'.format(17, -17)
'@17@@ @-17@'
>>> '{:@^ 5} {:@^ 5}'.format(17, -17)
'@ 17@ @-17@'
```

"#"

This option selects the “alternate form” of output for some types.

- When formatting integers as binary, octal, or hexadecimal, the alternate form adds “0b”, “0o”, or “0x” before the value, to show the radix explicitly.

```
>>> "{:4x}".format(255)
' ff'
>>> "{:#4x}".format(255)
'0xff'
>>> "{:9b}".format(62)
' 111110'
>>> "{:#9b}".format(62)
' 0b111110'
>>> "{:<#9b}".format(62)
'0b111110 '
```

- When formatting float, complex, or Decimal values, the “#” option forces the result to contain a decimal point, even if it is a whole number.

```

>>> "{:5.0f}".format(36)
'   36'
>>> "{:#5.0f}".format(36)
'  36.'
>>> from decimal import Decimal
>>> w=Decimal(36)
>>> "{:g}".format(w)
'36'
>>> "{:#g}".format(w)
'36.'

```

"0"

To fill the field with left zeroes, place a "0" at this position in your format code.

```

>>> "{:5d}".format(36)
'   36'
>>> "{:05d}".format(36)
'00036'
>>> "{:021.15}".format(1.0/7.0)
'00000.142857142857143'

```

width

Place a number at this position to specify the total width of the displayed value.

```

>>> "Beware the {}!".format('Penguin')
'Beware the Penguin!'
>>> "Beware the {:11}!".format('Penguin')
'Beware the Penguin      !'
>>> "Beware the {:>11}!".format('Penguin')
'Beware the           Penguin!'

```

","

Place a comma at this position in your format code to display commas between groups of three digits in whole numbers.

Note

This feature was added in Python 2.7.

```

>>> "{:,d}".format(12345678901234)
'12,345,678,901,234'
>>> "{:,f}".format(1234567890123.456789)
'1,234,567,890,123.456787'
>>> "{:25,f}".format(98765432.10987)
'          98,765,432.109870'

```

"." precision

Use this part to specify the number of digits after the decimal point.

```

>>> from math import pi
>>> "{}".format(pi)
'3.141592653589793'

```

```
>>> "{:.3}".format(pi)
'3.14'
>>> "{:25, .3f}".format(1234567890123.456789)
'1,234,567,890,123.457'
```

type

This code specifies the general type of format used. The default is to convert the value of a string as if using the `str()` function. Refer to the table below for allowed values.

b	Format an integer in binary.
c	Given a number, display the character that has that code.
d	Display a number in decimal (base 10).
e	Display a <code>float</code> value using the exponential format.
E	Same as <code>e</code> , but use a capital "E" in the exponent.
f	Format a number in fixed-point form.
g	General numeric format: use either <code>f</code> or <code>g</code> , whichever is appropriate.
G	Same as "g", but uses a capital "E" in the exponential form.
n	For formatting numbers, this format uses the current local setting to insert separator characters. For example, a number that Americans would show as "1,234.56", Europeans would show it as "1.234,56".
o	Display an integer in octal format.
x	Display an integer in hexadecimal (base 16). Digits greater than 9 are displayed as lowercase characters.
X	Display an integer in hexadecimal (base 16). Digits greater than 9 are displayed as uppercase characters.
%	Display a number as a percentage: its value is multiplied by 100, followed by a "%" character.

Examples:

```
>>> "{:b}".format(9)
'1001'
>>> "{:08b}".format(9)
'00001001'
>>> "{:c}".format(97)
'a'
>>> "{:d}".format(0xff)
'255'
>>> from math import pi
>>> "{:e}".format(pi*1e10)
'3.141593e+10'
>>> "{:E}".format(pi*1e10)
'3.141593E+10'
>>> "{:f}".format(pi)
'3.141593'
>>> "{:g}".format(pi)
'3.14159'
>>> "{:g}".format(pi*1e37)
'3.14159e+37'
>>> "{:G}".format(pi*1e37)
```

```
'3.14159E+37'
>>> "{:o}".format(255)
'377'
>>> "{:#o}".format(255)
'0o377'
>>> "{:x}".format(105199)
'19aef'
>>> "{:X}".format(105199)
'19AEF'
>>> "{:<#9X}".format(105199)
'0X19AEF'
>>> "{:%}".format(0.6789)
'67.890000%'
>>> "{:15.3%}".format(0.6789)
'        67.890%'
```

9.4.5. Formatting a field of variable length

Sometimes you need to format a field using a length that is available only once the program is running. To do this, you can use a number or name in `{braces}` *inside* a format code at the *width* position. This item then refers to either a positional or keyword argument to the `.format()` method as usual.

Here's an example. Suppose you want to format a number `n` using `d` digits. Here are examples showing this with and without left-zero fill:

```
>>> n = 42
>>> d = 8
>>> "{0:{1}d}".format(42, 8)
'    42'
>>> "{0:0{1}d}".format(42, 8)
'00000042'
>>>
```

You can, of course, also use keyword arguments to specify the field width. This trick also works for variable precision.

```
"{count:0{width}d}".format(width=8, count=42)
'00000042'
>>>
```

The same technique applies to substituting any of the pieces of a format code.

```
>>> "{:&<14,d}".format(123456)
'123,456&&&&&&&&&'
>>> "{1:{0}{2}{3},{4}}".format('&', 123456, '<', 14, 'd')
'123,456&&&&&&&&&'
>>> "{:@^14,d}".format(1234567)
'@@1,234,567@@@'
>>> "{n:{fil}{al}{w},{kind}}".format(
...     kind='d', w=14, al='^', fil='@', n=1234567)
'@@1,234,567@@@'
```

9.5. The older string format operator

Python versions before 2.6 did not have the string `.format()` method described in Section 9.4, “The string `.format()` method” (p. 22). Instead, string formatting used this general form:

```
f % v
```

where `f` is a template string and `v` specifies the value or values to be formatted using that template. If multiple values are to be formatted, `v` must be a tuple.

The template string may contain any mixture of ordinary text and *format codes*. A format code always starts with a percent (%) symbol. See Table 4, “Format codes” (p. 31).

The result of a format operation consists of the ordinary characters from the template with values substituted within them wherever a format code occurs. A conversational example:

```
>>> print "We have %d pallets of %s today." % (49, "kiwis")
We have 49 pallets of kiwis today.
```

In the above example, there are two format codes. Code “%d” means “substitute a decimal number here,” and code “%s” means “substitute a string value here”. The number 49 is substituted for the first format code, and the string “kiwis” replaces the second format code.

In general, format codes have this form:

```
 %[p][m[.n]]c
```

Table 3. Parts of the format operator

<i>p</i>	An optional prefix; see Table 5, “Format code prefixes” (p. 31).
<i>m</i>	Specifies the total desired field width. The result will never be shorter than this value, but may be longer if the value doesn't fit; so, “%5d” % 1234 yields “ 1234”, but “%2d” % 1234 yields “1234”. If the value is negative, values are left-aligned in the field whenever they don't fill the entire width.
<i>n</i>	For <code>float</code> values, this specifies the number of digits after the decimal point.
<i>c</i>	Indicates the type of formatting.

Here are the format type codes, `c` in the general expression above:

Table 4. Format codes

%s	Format a string. For example, '%-3s' % 'xy' yields 'xy '; the width (-3) forces left alignment.
%d	Decimal conversion. For example, '%3d' % -4 yields the string ' -4'.
%e	Exponential format; allow four characters for the exponent. Examples: '%08.1e' % 1.9783 yields '0002.0e+00'.
%E	Same as %e, but the exponent is shown as an uppercase E.
%f	For float type. E.g., '%4.1f' % 1.9783 yields ' 2.0'.
%g	General numeric format. Use %f if it fits, otherwise use %e.
%G	Same as %G, but an uppercase E is used for the exponent if there is one.
%o	Octal (base 8). For example, '%o' % 13 yields '15'.
%x	Hexadecimal (base 16). For example, '%x' % 247 yields 'f7'.
%X	Same as %x, but capital letters are used for the digits A-F. For example, '%04X' % 247 yields '00F7'; the leading zero in the length (04) requests that Python fill up any empty leading positions with zeroes.
%c	Convert an integer to the character with the corresponding ASCII ¹⁰ code. For example, '%c' % 0x61 yields the string 'a'.
%%	Places a percent sign (%) in the result. Does not require a corresponding value. Example: "Energy at %d%%." % 88 yields the value 'Energy at 88%.'.

Table 5. Format code prefixes

+	For numeric types, forces the sign to appear even for positive values.
-	Left-justifies the value in the field.
0	For numeric types, use zero fill. For example, '%04d' % 2 produces the value '0002'.
#	With the %o (octal) format, append a leading "0"; with the %x (hexadecimal) format, append a leading "0x"; with the %g (general numeric) format, append all trailing zeroes. Examples:
	<pre> >>> '%4o' % 127 ' 177' >>> '%#4o' % 127 '0177' >>> '%x' % 127 '7f' >>> '%#x' % 127 '0x7f' >>> '%10.5g' % 0.5 ' 0.5' >>> '%#10.5g' % 0.5 ' 0.50000' </pre>

You can also use the string format operator % to format a set of values from a dictionary *D* (see Section 16, "Type dict: Dictionaries" (p. 49)):

```
f % D
```

¹⁰ <http://en.wikipedia.org/wiki/ASCII>

In this form, the general form for a format code is:

```
%(k)[p][m[.n]]c
```

where k is a key in dictionary D , and the rest of the format code is as in the usual string format operator. For each format code, the value of $D[k]$ is used. Example:

```
>>> named = {'last': 'Poe', 'first': 'Aloysius'}
>>> 'Dear %(first)s %(last)s:' % named
'Dear Aloysius Poe:'
```

10. Type unicode: Strings of 32-bit characters

With the advent of the Web as medium for worldwide information interchange, the Unicode character set has become vital. For general background on this character set, see the Unicode homepage¹¹.

To get a Unicode string, prefix the string with `u`. For example:

```
u'klarn'
```

is a five-character Unicode string.

To include one of the special Unicode characters in a string constant, use these escape sequences:

<code>\xHH</code>	For a code with the 8-bit hexadecimal value HH .
<code>\uHHHH</code>	For a code with the 16-bit hexadecimal value $HHHH$.
<code>\UHHHHHHHH</code>	For a code with the 32-bit hexadecimal value $HHHHHHHH$.

Examples:

```
>>> u'Klarn.'
u'Klarn.'
>>> u'Non-breaking-\xa0-space.'
u'Non-breaking-\xa0-space.'
>>> u'Less-than-or-equal symbol: \u2264'
u'Less-than-or-equal symbol: \u2264'
>>> u"Phoenician letter 'wau': \U00010905"
u"Phoenician letter 'wau': \U00010905"
>>> len(u'\U00010905')
1
```

All the operators and methods of `str` type are available with `unicode` values.

Additionally, for a Unicode value U , use this method to encode its value as a string of type `str`:

`U.encode (encoding[, error]`

Return the value of U as type `str`. The *encoding* argument is a string that specifies the encoding method. In most cases, this will be `'utf_8'`. For discussion and examples, see Section 10.1, "The UTF-8 encoding" (p. 33).

The optional *error* string specifies what to do with characters that do not have exact equivalents. For example, if you are converting to the ASCII¹² character set, the *encoding* argument is `'ascii'`. Values of the *error* argument are given in the table below.

¹¹ <http://www.unicode.org/>

¹² <http://en.wikipedia.org/wiki/ASCII>

'strict'	Raise a <code>UnicodeError</code> exception if any character has no ASCII equivalent. This is the default behavior.
'ignore'	Leave out characters that have no equivalent.
'replace'	Substitute a '?' for each character that has no equivalent.
'xmlcharrefreplace'	Use the XML character entity escape sequence for characters with no ASCII equivalent. The general form of this sequence is "&#N;", where <i>N</i> is the decimal value of the character's code point. This feature is very handy for generating internationalized Web pages.
'backslashreplace'	Use Python backslash escape sequences to represent characters with no equivalent.

Here are some examples to demonstrate `error` argument values.

```
>>> s = u"a\u262ez"
>>> len(s)
3
>>> s
u'a\u262ez'
>>> s.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character u'\u262e'
in position 1: ordinal not in range(128)
>>> s.encode('ascii', 'ignore')
'az'
>>> s.encode('ascii', 'replace')
'a?z'
>>> s.encode('ascii', 'xmlcharrefreplace')
'a&#9774;z'
>>> hex(9774)
'0x262e'
>>> t = s.encode('ascii', 'backslashreplace')
>>> t
'a\\u262eb'
>>> print t
a\u262eb
>>> len(t)
8
>>> t[1]
'\\'
```

10.1. The UTF-8 encoding

How, you might ask, do we pack 32-bit Unicode characters into 8-bit bytes? Quite prevalent on the Web and the Internet generally is the UTF-8 encoding, which allows any of the Unicode characters to be represented as a string of one or more 8-bit bytes.

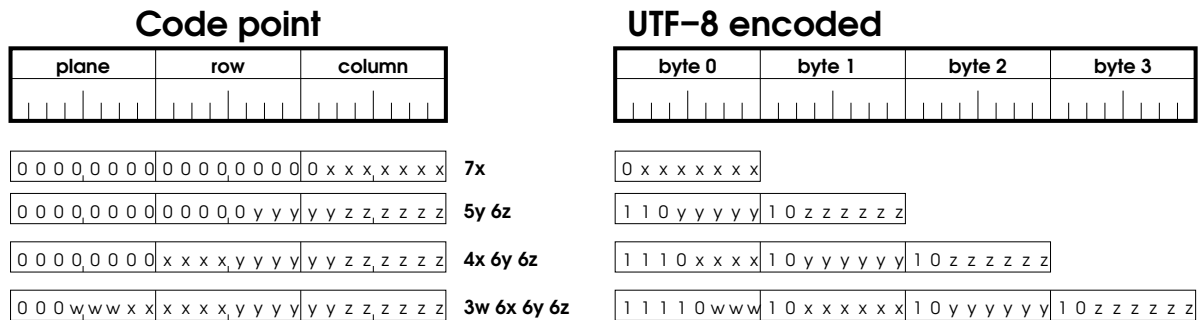
First, some definitions:

- A *code point* is a number representing a unique member of the Unicode character set.

- The Unicode code points are visualized as a three-dimensional structure made of *planes*, each of which has a range of 65536 code points organized as 256 *rows* of 256 *columns* each.

The low-order eight bits of the code point select the column; the next eight more significant bits select the row; and the remaining most significant bits select the plane.

This diagram shows how UTF-8 encoding works. The first 128 code points (hexadecimal 00 through 7F) are encoded as in normal 7-bit ASCII¹³, with the high-order bit always 0. For code points above hex 7F, all bytes have the high-order (0x80) bit set, and the bits of the code point are distributed through two, three, or four bytes, depending on the number of bits needed to represent the code point value.



To encode a Unicode string *U*, use this method:

```
U.encode('utf_8')
```

To decode a regular `str` value *S* that contains a UTF-8 encoded value, use this method:

```
S.decode('utf_8')
```

Examples:

```
>>> tilde='~'
>>> tilde.encode('utf_8')
'~'
>>> u16 = u'\u0456'
>>> s = u16.encode('utf_8')
>>> s
'\xd1\x96'
>>> s.decode('utf_8')
u'\u0456'
>>> u32 = u'\U000E1234'
>>> s = u32.encode('utf_8')
>>> s
'\xf3\xa1\x88\xb4'
>>> s.decode('utf_8')
u'\U000e1234'
```

UTF-8 is not the only encoding method. For more details, consult the documentation for the Python module `codecs`¹⁴.

¹³ <http://en.wikipedia.org/wiki/ASCII>
¹⁴ <http://docs.python.org/library/codecs.html>

11. Type `list`: Mutable sequences

To form values into a sequence, use Python's `list` type if you are going to change, delete, or add values to the sequence. For a discussion of when to use `list` and when to use `tuple`, see Section 12, “Type `tuple`: Immutable sequences” (p. 39).

To create a list, enclose a list of zero or more comma-separated values inside square brackets, “[...]”. Examples:

```
[ ]
["baked beans"]
[23, 30.9, 'x']
```

You can also create a list by performing specific operations on each element of some sequence; see Section 11.2, “List comprehensions” (p. 38).

Lists support all the operations described under Section 8.1, “Operations common to all the sequence types” (p. 12). Methods available on lists are discussed in Section 11.1, “Methods on lists” (p. 35).

There are a number of functions that can be used with lists as well:

- Section 20.2, “`all()`: Are all the elements of an iterable true?” (p. 61).
- Section 20.3, “`any()`: Are any of the members of an iterable true?” (p. 61).
- Section 20.8, “`cmp()`: Compare two values” (p. 63).
- Section 20.12, “`enumerate()`: Step through indices and values of an iterable” (p. 64)
- Section 20.14, “`filter()`: Extract qualifying elements from an iterable” (p. 65).
- Section 20.21, “`iter()`: Produce an iterator over a sequence” (p. 68).
- Section 20.22, “`len()`: Number of elements” (p. 69).
- Section 20.23, “`list()`: Convert to a list” (p. 69).
- Section 20.25, “`map()`: Apply a function to each element of an iterable” (p. 69).
- Section 20.26, “`max()`: Largest element of an iterable” (p. 70).
- Section 20.27, “`min()`: Smallest element of an iterable” (p. 71).
- Section 20.33, “`range()`: Generate an arithmetic progression as a list” (p. 73).
- Section 20.35, “`reduce()`: Sequence reduction” (p. 74).
- Section 20.36, “`reversed()`: Produce a reverse iterator” (p. 75).
- Section 20.39, “`sorted()`: Sort a sequence” (p. 76).
- Section 20.41, “`sum()`: Total the elements of a sequence” (p. 77).
- Section 20.46, “`xrange()`: Arithmetic progression generator” (p. 78).
- Section 20.47, “`zip()`: Combine multiple sequences” (p. 79).

11.1. Methods on lists

For any list value L , these methods are available.

`L.append(x)`

Append a new element x to the end of list L . Does not return a value.

```
>>> colors = ['red', 'green', 'blue']
>>> colors.append('indigo')
>>> colors
['red', 'green', 'blue', 'indigo']
```

`L.count(x)`

Return the number of elements of L that compare equal to x .

```
>>> [59, 0, 0, 0, 63, 0, 0].count(0)
5
>>> ['x', 'y'].count('Fomalhaut')
0
```

L.extend(S)

Append another sequence *S* to *L*.

```
>>> colors
['red', 'green', 'blue', 'indigo']
>>> colors.extend(['violet', 'pale puce'])
>>> colors
['red', 'green', 'blue', 'indigo', 'violet', 'pale puce']
```

L.index(x[, start[, end]])

If *L* contains any elements that equal *x*, return the index of the first such element, otherwise raise a `ValueError` exception.

The optional *start* and *end* arguments can be used to search only positions within the slice `L[start:end]`.

```
>>> colors
['red', 'green', 'blue', 'indigo', 'violet', 'pale puce']
>>> colors.index('blue')
2
>>> colors.index('taupe')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
>>> M=[0, 0, 3, 0, 0, 3, 3, 0, 0, 3]
>>> M.index(3)
2
>>> M.index(3, 4, 8)
5
>>> M.index(3, 0, 2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.index(x): x not in list
```

L.insert(i,x)

Insert a new element *x* into list *L* just before the *i*th element, shifting all higher-number elements to the right. No value is returned.

```
>>> colors
['red', 'green', 'blue', 'indigo', 'violet', 'pale puce']
>>> colors[1]
'green'
>>> colors.insert(1, "yellow")
>>> colors
['red', 'yellow', 'green', 'blue', 'indigo', 'violet', 'pale puce']
```

L.pop([i])

Remove and return the element with index *i* from *L*. The default value for *i* is -1, so if you pass no argument, the last element is removed.

```

>>> colors
['red', 'yellow', 'green', 'blue', 'indigo', 'violet', 'pale puce']
>>> tos = colors.pop()
>>> tos
'pale puce'
>>> colors
['red', 'yellow', 'green', 'blue', 'indigo', 'violet']
>>> colors[4]
'indigo'
>>> dye = colors.pop(4)
>>> dye
'indigo'
>>> colors
['red', 'yellow', 'green', 'blue', 'violet']

```

L.remove(x)

Remove the first element of *L* that is equal to *x*. If there aren't any such elements, raises `ValueError`.

```

>>> colors
['red', 'yellow', 'green', 'blue', 'violet']
>>> colors.remove('yellow')
>>> colors
['red', 'green', 'blue', 'violet']
>>> colors.remove('cornflower')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list
>>> notMuch = [0, 0, 3, 0]
>>> notMuch.remove(0)
>>> notMuch
[0, 3, 0]
>>> notMuch.remove(0)
>>> notMuch
[3, 0]
>>> notMuch.remove(0)
>>> notMuch
[3]
>>> notMuch.remove(0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: list.remove(x): x not in list

```

L.reverse()

Reverses the elements of *L* *in place*. Does not return a result. Compare Section 20.36, “`reversed()`: Produce a reverse iterator” (p. 75).

```

>>> colors
['red', 'green', 'blue', 'violet']
>>> colors.reverse()
>>> colors
['violet', 'blue', 'green', 'red']

```

`L.sort(cmp[,key[,reverse]])`

Sort list *L* in place. Does not return a result. Compare Section 20.39, “sorted(): Sort a sequence” (p. 76).

The reordering is guaranteed to be *stable*—that is, if two elements are considered equal, their order after sorting will not change.

While sorting, Python will use the built-in `cmp()` function to compare elements; see Section 20.8, “`cmp()`: Compare two values” (p. 63). You may provide, as the first argument to the `.sort()` method, your own *comparator function* to compare elements. This function must have the same calling sequence and return value convention as the built-in `cmp()` function: it must take two arguments, and return a negative number if the first argument precedes the second, a positive number if the second argument precedes the first, or zero if they are considered equal.

You may also provide a “key extractor function” that is applied to each element to determine its key. This function must take one argument and return the value to be used as the sort key. If you want to provide a key extractor function but not a comparator function, pass `None` as the first argument to the method.

Additionally, you may provide a third argument of `True` to sort the sequence in descending order; the default behavior is to sort into ascending order.

```
>>> temps=[67, 73, 85, 93, 92, 78, 95, 100, 104]
>>> temps.sort()
>>> temps
[67, 73, 78, 85, 92, 93, 95, 100, 104]
>>> def reverser(n1, n2):
...     '''Comparison function to use reverse order.
...     '''
...     return cmp(n2, n1)
...
>>> temps.sort(reverser)
>>> temps
[104, 100, 95, 93, 92, 85, 78, 73, 67]
>>> def unitsDigit(n):
...     '''Returns only the units digit of n.
...     '''
...     return n % 10
...
>>> temps.sort(None, unitsDigit)
>>> temps
[100, 92, 93, 73, 104, 95, 85, 67, 78]
>>> temps.sort(None, None, True)
>>> temps
[104, 100, 95, 93, 92, 85, 78, 73, 67]
```

11.2. List comprehensions

You can use a form called a *list comprehension* to create a list. The general form is:

```
[ e
  for v1 in s1
  for v2 in s2
  ...
  if c ]
```

where *e* is some expression, followed by one or more `for` clauses, optionally followed by an `if` clause.

The result is a list containing all the values of expression *e* after all the nested `for` loops have been run; the `for` loops have the same structure as in Section 23.4, “The `for` statement: Iteration over a sequence” (p. 101). If there is an “`if`” clause, it determines which values of *e* are added to the list: if the `if` condition is true, the value is added, otherwise it is not added.

This is perhaps easiest to explain with a few examples. In the first example, we construct a list containing the cubes of the numbers from 1 to 10, inclusive. The `for` loop generates the numbers 1, 2, ..., 10, and then the expression “`x**3`” cubes each one and appends it to the resulting list.

```
>>> range(1, 11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [ x**3 for x in range(1,11) ]
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

In the next example, we use two `for` loops. The outer loop generates the sequence [1, 2, 3], and the inner loop generates the sequence [50, 51]. The expression “`x*1000 + y`” is computed for each of the resulting six value sets for *x* and *y*, and the result is appended to the list being built.

```
>>> [ x*1000 + y
...   for x in range(1,4)
...   for y in range(50, 52) ]
[1050, 1051, 2050, 2051, 3050, 3051]
```

In the next example, there are two nested loops, each generating the sequence [0, 1, 2]. For each of the nine trips through the inner loop, we test the values of *x* and *y* and discard the cases where they are equal. The expression “(*y*, *x*)” combines the two values into a 2-tuple.

```
>>> [ (y, x)
...   for y in range(3)
...   for x in range(3)
...   if x != y ]
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

12. Type tuple: Immutable sequences

For representing a sequence of values, Python has two similar *container types*: `list` and `tuple`.

An instance of a container type is basically a value that has other values inside it; we call the contained values *elements*.

So, when should you use a list, and when a tuple? In many contexts, either will work. However, there are important differences.

- Values of type `list` are mutable; that is, you can delete or add elements, or change the value of any of the elements inside the list.

Lists cannot be used in certain contexts. For example, you can't use a list as the key in a dictionary.

```
>>> d={}
>>> d[(23,59)] = 'hike'
>>> d[[46,19]] = 'hut'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
```

- Values of type `tuple` are immutable. Once you have assembled a tuple, you cannot add or delete elements or change the value of an element inside the tuple.

Among the reasons to use a tuple instead of a list:

- Tuples are allowed in certain contexts where lists are not, such as the right-hand argument of the string format operator, or as a key in a dictionary.
- If your program is in danger of running out of processor memory, tuples are slightly more efficient in their memory usage.

Write a literal tuple as a sequence of values in parentheses, separated by commas. There may be any number of values of any type or any mixture of types. There may be zero values.

To write a tuple with exactly one value, you must use this special syntax:

```
(value,)
```

That is, you must provide a comma before the closing “)”, in order to show that it is a tuple, and not just a parenthesized expression. Note especially the last two examples below:

```
>>> ()
()
>>> ('farcical', 'aquatic', 'ceremony')
('farcical', 'aquatic', 'ceremony')
>>> ('Ni',)
('Ni',)
>>> ('Ni')
'Ni'
```

You may also convert any iterable into a tuple using Section 20.42, “`tuple()`: Convert to a tuple” (p. 77).

The tuple type does not have comprehensions (see Section 11.2, “List comprehensions” (p. 38)), but you can get the equivalent by applying the `tuple()` function to a list comprehension. Here is an example:

```
>>> tuple([x**2 for x in (2,3,8)])
(4, 9, 64)
>>>
```

Tuples also support the `.index()` and `.count()` methods as described in Section 11.1, “Methods on lists” (p. 35).

13. The bytes type

To understand why Python version 2.6 and beyond have a `bytes` type, it is necessary to review a little history.

Most early computing used 7- and 8-bit character codes, but these character sets are very limited. In particular, life was difficult for Francophone countries when “è” and “é” are very different letters. The 32-bit character set of the Unicode standard¹⁵ is the current preferred practice, and provides enough characters to last a good while into the future.

Text handling in the Python 2.x releases was awkward due to the presence of two different types for representing character data: `str` and `unicode`. Consequently, in the upcoming major incompatible 3.x releases, all character data will be represented internally by 32-bit characters.

¹⁵ <http://www.unicode.org/>

Therefore, in Python 2.6 the `bytes` type was added to aid transition to the 3.0 family, which has a separate `bytes` type for 8-bit character strings. In the 3.x versions, a `bytes` value is a sequence of zero or more unsigned 8-bit integers, each in the range 0–255, inclusive.

In Python 2.6 and subsequent versions, the `bytes` type is a synonym for `str`. The `bytes()` function works exactly like the `str()` function.

```
>>> s=bytes(987)
>>> s
'987'
>>> type(s)
<type 'str'>
```

Use this type where your program expects 8-bit characters, and it will ease the transition to Python 3.x, because the semi-automated translation process will know that values of `bytes` type are intended for sequences of 8-bit characters.

13.1. Using the `bytes` type in 3.x conversion

Versions 2.6+ support a new notation: to create a literal of type `bytes`, place a “b” just before the opening quote.

```
>>> s = b'abc'
>>> s
'abc'
>>> type(s)
<type 'str'>
```

Such literals are exactly like regular string literals. The difference comes when you convert your program to the 3.x versions. In Python 3.x, a string of the form `b' . . . '` will have type `bytes`, which will be different than the `str` (32-bit character) type in 3.x.

One step in converting your 2.x programs to 3.x is to add this `import` before all the other imports in your program:

```
from __future__ import unicode_literals
```

In programs that start with this declaration, all string literals will automatically be considered `unicode` type without using the `u' . . . '` prefix. This means you may also include escape sequences of the form `'\uXXXX'`, each of which designates a 16-bit Unicode code point as four hexadecimal digits `XXXX`.

Here is a demonstration of the difference. Before the `import`, the `\u` escape is not recognized, and the value has type `str`. Afterwards, the return value is type `unicode`

```
>>> s = '\u2672'
>>> len(s)
6
>>> s
'\\u2672'
>>> type(s)
<type 'str'>
>>> from __future__ import unicode_literals
>>> t = '\u2672'
>>> len(t)
```

```
1
>>> type(t)
<type 'unicode'>
>>> t
u'\u2672'
```

14. The bytearray type

New in Python 2.6 is the `bytearray` type. Each instance is a sequence of 8-bit bytes, each of which is an unsigned integer in the range $0 \leq 255$. Unlike the `str` type, however, `bytearray` values are mutable: you can delete, insert, or replace arbitrary values or slices.

As with the features described in Section 13, “The `bytes` type” (p. 40), this type is intended to ease the transition to Python 3.x versions. Use it for situations where you are handling sequences of 8-bit bytes that are **not** intended as textual representations, such as raw binary data.

Values of this type support almost all of the operators and methods of the `str` type (with the exception of `.encode()` and `.format()` methods). They also support these methods of the `list` type: `.extend()`, `.insert()`, `.pop()`, `.remove()`, `.reverse()`. You can also replace values using either integers or the `b'...'` (`bytes`) literals.

Some examples:

```
>>> s=bytearray('abcdef')
>>> s
bytearray(b'abcdef')
>>> type(s)
<type 'bytearray'>
>>> s[3]
100
>>> s.insert(0, b'^')
>>> s
bytearray(b'^abcdef')
>>> s.reverse()
>>> s
bytearray(b'fedcba^')
>>> s[2:6]
bytearray(b'dcba')
>>> s[2:6] = b'#'
>>> s
bytearray(b'fe#^')
>>> s[0]=63
>>> s
bytearray(b'?e#^')
>>>
```

The `bytearray` type also has a static method named `.fromhex()` that creates a `bytearray` value from a Unicode string containing hexadecimal characters (which may be separated by spaces for legibility).

```
>>> ao = bytearray.fromhex(u'00 ff')
>>> ao
bytearray(b'\x00\xff')
```

```
>>> ao[1]
255
```

15. Types `set` and `frozenset`: Set types

Mathematically speaking, a set is an unordered collection of zero or more distinct elements. Python has two set types that represent this mathematical abstraction. Use these types when you care only about whether something is a member of the set or not, and you don't need them to be in any specific order.

The elements of a Python set must be immutable. In particular, you can't have list or dictionary elements in a set.

Most operations on sets work with both `set` and `frozenset` types.

- Values of type `set` are mutable: you can add or delete members.

There are two ways to create a mutable set.

- In all Python versions of the 2.x series, the `set(S)` function operates on a sequence `S` and returns a mutable set containing the unique elements of `S`. The argument is optional; if omitted, you get a new, empty set.

```
>>> s1 = set([1, 1, 1, 9, 1, 8, 9, 8, 3])
set([8, 1, 3, 9])
>>> s1 = set([1, 1, 1, 9, 1, 8, 9, 8, 3])
>>> s2 = set()
>>> s1
set([8, 1, 3, 9])
>>> s2
set([])
>>> print len(s1), len(s2)
4 0
>>> s3 = set("notlob bolton")
>>> s3
s3
set([' ', 'b', 'l', 'o', 'n', 't'])
```

- Starting in Python 2.7, you can create a set by simply enclosing one or more elements within braces `{...}` separated by commas.

```
s1 = {1, 1, 1, 9, 1, 8, 9, 8, 3}
>>> s1
set([8, 9, 3, 1])
```

Note the wording “one or more:” an empty pair of braces “`{}`” is an empty dictionary, not an empty set.

- A `frozenset` value is immutable: you can't change the membership, but you can use a `frozenset` value in contexts where `set` values are not allowed. For example, you can use a `frozenset` as a key in a dictionary, but you can't use a `set` value as a dictionary key.

To create a `set` or `frozenset`, see Section 20.38, “`set()`: Create an algebraic set” (p. 76) and Section 20.17, “`frozenset()`: Create a frozen set” (p. 66).

A number of functions that work on sequences also work on sets. In each case, the set is converted to a list before being passed to the function.

- Section 20.2, “`all()`: Are all the elements of an iterable true?” (p. 61). Predicate to test whether all members of a set are `True`.
- Section 20.3, “`any()`: Are any of the members of an iterable true?” (p. 61). Predicate to test whether any member of a set is true.
- Section 20.14, “`filter()`: Extract qualifying elements from an iterable” (p. 65). Returns a list of the elements that pass through a filtering function.
- Section 20.21, “`iter()`: Produce an iterator over a sequence” (p. 68). Returns an iterator that will visit every element of the set.
- Section 20.22, “`len()`: Number of elements” (p. 69). Returns the length (cardinality) of the set.
- Section 20.23, “`list()`: Convert to a list” (p. 69). Returns the elements of the set as a list.
- Section 20.25, “`map()`: Apply a function to each element of an iterable” (p. 69). Returns a list containing the result of the application of a function to each element of a set.
- Section 20.26, “`max()`: Largest element of an iterable” (p. 70). Returns the largest element of a set.
- Section 20.27, “`min()`: Smallest element of an iterable” (p. 71). Returns the smallest element of a set.
- Section 20.35, “`reduce()`: Sequence reduction” (p. 74). Returns the result of the application of a given function pairwise to all the elements of a set.
- Section 20.39, “`sorted()`: Sort a sequence” (p. 76). Returns a list containing the sorted elements of the set.

Another new feature in Python 2.7 is the *set comprehension*. This is similar to the feature described in Section 11.2, “List comprehensions” (p. 38). Here is the general form:

```
{ e
  for v1 in s1
  for v2 in s2
  ...
  if c }
```

As with a list comprehension, you use one or more `for` clauses to iterate over sets of values, and the expression `e` is evaluated for every combination of the values in the sequences s_i . If there is no “`if`” clause, or if the “`if`” condition evaluates as `True`, the value is added to the sequence from which a set is then constructed.

Here is an example. Function `takeUppers()` takes one string argument and returns a set of the unique letters in that string, uppcased. The `for` clause iterates over the characters in the argument `s`; the `if` clause discards characters that aren't letters; and the `.upper()` method converts lowercase letters to uppercase.

```
>>> def takeUpper(s):
...     return { c.upper()
...              for c in s
...              if c.isalpha() }
...
>>> takeUpper("A a|ccCc^#zZ")
set(['A', 'C', 'Z'])
```

15.1. Operations on mutable and immutable sets

These operations are supported by both `set` and `frozenset` types:

`x in S`

Predicate that tests whether element `x` is a member of set `S`.

```
>>> 1 in set([0,1,4])
True
>>> 99 in set([0,1,4])
False
```

x not in S

Predicate that tests whether element *x* is *not* a member of set *S*.

```
>>> 1 not in set([0,1,4])
False
>>> 99 not in set([0,1,4])
True
```

S1 == S2

Predicate that tests whether sets *S1* and *S2* have exactly the same members.

```
>>> set('bedac') == set('abcde')
True
>>> set('bedac') == set('bedack')
False
```

S1 != S2

Predicate that tests whether sets *S1* and *S2* have different members.

```
>>> set('bedac') != set('abcde')
False
>>> set('bedac') != set('bedack')
True
```

S1 < S2

Predicate that tests whether *S1* is a proper subset of *S2*; that is, all the elements of *S1* are also members of *S2*, but there is at least one element of *S2* that is not in *S1*.

```
>>> set('ab') < set('ab')
False
>>> set('ab') < set('abcde')
True
```

S1 > S2

Predicate that tests whether *S1* is a proper superset of *S2*; that is, all the elements of *S2* are also members of *S1*, but there is at least one element of *S1* that is not in *S2*.

```
>>> set('ab') > set('ab')
False
>>> set('abcde') > set('cd')
True
```

S.copy()

Return a new set of the same type as *S*, containing all the same elements.

```
>>> s1=set('aeiou')
>>> s2=s1
>>> s3=s1.copy()
>>> s1.add('y')
>>> s1
```

```
set(['a', 'e', 'i', 'o', 'u', 'y'])
>>> s2
set(['a', 'e', 'i', 'o', 'u', 'y'])
>>> s3
set(['a', 'i', 'e', 'u', 'o'])
```

S1.difference(S2)

Returns a new set of the same type as *S1*, containing only those values found in *S1* but *not* found in *S2*. The *S2* argument may be a set or a sequence.

```
>>> set('roygbiv').difference('rgb')
set(['i', 'o', 'v', 'y'])
```

S1 - S2

Same as *S1.difference(S2)*, except that *S2* must be a set.

```
>>> set('roygbiv') - set('rgb')
set(['i', 'y', 'o', 'v'])
```

S1.intersection(S2)

Returns a new set, of the same type as *S1*, containing only the elements found both in *S1* and *S2*. *S2* may be a set or a sequence.

```
>>> set([1,2,3,5,7,11]).intersection(set([1,3,5,7,9]))
set([1, 3, 5, 7])
>>> set([1,3,5]).intersection( (2,4,6,8) )
set([])
```

S1 & S2

Same as *S1.intersection(S2)*, but *S2* must be a set.

S1.issubset(S2)

Predicate that tests whether every element of *S1* is also in *S2*. *S2* may be a set or a sequence.

```
>>> set([1,2]).issubset(set([2,4,1,8]))
True
>>> set([2,4,1,8]).issubset(set([1,2]))
False
>>> set(['r', 'g', 'b']) <= set(['r', 'o', 'y', 'g', 'b', 'i', 'v'])
True
```

S1 <= S2

Same as *S1.issubset(S2)*, but *S2* must be a set.

S1.issuperset(S2)

Predicate that tests whether every element of *S2* is also in *S1*. *S2* may be a set or a sequence.

```
>>> set([1,2]).issuperset(set([2,4,1,8]))
False
>>> set([2,4,1,8]).issuperset(set([1,2]))
True
```

S1 >= S2

Same as *S1.issuperset(S2)*.

`S1.symmetric_difference(S2)`

Returns a new set of the same type as *S1*, containing only elements found in *S1* or *S2*, but not found in both. The *S2* argument may be a set or a sequence.

```
>>> set('aeiou').symmetric_difference('etaoin')
set(['n', 'u', 't'])
```

`S1 ^ S2`

Same as `S1.symmetric_difference(S2)`, but *S2* must be a set.

`S1.union(S2)`

Returns a new set, with the same type as *S1*, containing all the elements found in either *S1* or *S2*.

The *S2* argument may be a set or a sequence.

```
>>> set([1,2]).union(set([1,3,7]))
set([1, 2, 3, 7])
>>> set([1,2]).union( (8,2,4,5) )
set([8, 1, 2, 4, 5])
```

`S1 | S2`

Same as `S1.union(S2)`.

15.2. Operations on mutable sets

The operations described in this section apply to `set` (mutable) values, but may not be used with `frozenset` (immutable) values.

`S.add(x)`

Add element *x* to set *S*. Duplicate elements will be ignored.

```
>>> pbr=set(['USA', 'Brazil', 'Canada'])
>>> pbr.add('Australia')
>>> pbr
set(['Brazil', 'Canada', 'Australia', 'USA'])
>>> pbr.add('USA')
>>> pbr
set(['Brazil', 'Canada', 'Australia', 'USA'])
```

`S.clear()`

Remove all the elements from set *S*.

```
>>> pbr
set(['Brazil', 'USA'])
>>> pbr.clear()
>>> pbr
set([])
```

`S.discard(x)`

If set *S* contains element *x*, remove that element from *S*.

If *x* is not in *S*, it is not considered an error; compare `S.remove(x)`.

```
>>> pbr
set(['Brazil', 'Australia', 'USA'])
>>> pbr.discard('Swaziland')
```

```
>>> pbr
set(['Brazil', 'Australia', 'USA'])
>>> pbr.discard('Australia')
>>> pbr
set(['Brazil', 'USA'])
```

S1.difference_update(S2)

Modify set *S1* by removing any values found in *S2*. Value *S2* may be a set or a sequence.

```
>>> s1=set('roygbiv')
>>> s1.difference_update('rgb')
>>> s1
set(['i', 'o', 'v', 'y'])
```

S1 -= S2

Same as *S1.difference_update(S2)*, but *S2* must be a set.

S1.intersection_update(S2)

Modify set *S1* so that it contains only values found in both *S1* and *S2*.

```
>>> s1=set('roygbiv')
>>> s1
set(['b', 'g', 'i', 'o', 'r', 'v', 'y'])
>>> s1.intersection_update('roy')
>>> s1
set(['y', 'r', 'o'])
```

S1 &= S2

Same as *S1.intersection_update(S2)*, but *S2* must be a set.

S.remove(x)

If element *x* is in set *S*, remove that element from *S*.

If *x* is not an element of *S*, this operation will raise a `KeyError` exception.

```
>>> pbr
set(['Brazil', 'Canada', 'Australia', 'USA'])
>>> pbr.remove('Canada')
>>> pbr
set(['Brazil', 'Australia', 'USA'])
>>> pbr.remove('Swaziland')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Swaziland'
```

S1.symmetric_difference_update(S2)

Remove from *S1* any elements found in both *S1* and *S2*. Value *S2* may be a set or a sequence.

```
>>> s1=set('abcd')
>>> s1.symmetric_difference_update('cdefg')
>>> s1
set(['a', 'b', 'e', 'g', 'f'])
```

S1 ^= S2

Same as *S1.symmetric_difference_update(S2)*, but *S2* must be a set.

***S1*.update(*S2*)**

Add to *S1* any elements of *S2* not found in *S1*. The *S2* argument may be a set or a sequence.

```
>>> s1=set('rgb')
>>> s1
set(['r', 'b', 'g'])
>>> s1.update('roygbiv')
>>> s1
set(['b', 'g', 'i', 'o', 'r', 'v', 'y'])
```

S1* |= *S2

Same as *S1*.update(*S2*), but *S2* must be a set.

16. Type dict: Dictionaries

Python dictionaries are one of its more powerful built-in types. They are generally used for look-up tables and many similar applications.

A Python dictionary represents a set of zero or more ordered pairs (k_i, v_i) such that:

- Each k_i value is called a *key*;
- each key is unique and immutable; and
- the associated *value* v_i can be of any type.

Another term for this structure is *mapping*, since it maps the set of keys onto the set of values (in the algebraic sense).

To create a new dictionary, use this general form:

```
{  $k_0$ :  $v_0$ ,  $k_1$ :  $v_1$ , ... }
```

There can be any number of key-value pairs (including zero). Each key-value has the form " $k_i:v_i$ ", and pairs are separated by commas. Here are some examples of dictionaries:

```
{ }
{ 'Bolton': 'Notlob', 'Ipswich': 'Esher' }
{(1,1):48, (8,20): 52}
```

For efficiency reasons, the order of the pairs in a dictionary is arbitrary: it is essentially an unordered set of ordered pairs. If you display a dictionary, the pairs may be shown in a different order than you used when you created it.

```
>>> signals = {0:'red', 1: 'yellow', 2:'green'}
>>> signals
{2: 'green', 0: 'red', 1: 'yellow'}
```

16.1. Operations on dictionaries

These operations are available on any dictionary object *D*:

len(*D*)

Returns the number of key-value pairs in *D*.

$D[k]$

If dictionary D has a key whose value is equal to k , this operation returns the corresponding value for that key. If there is no matching key, it raises a `KeyError` exception.

```
>>> signals = {0: 'red', 1: 'yellow', 2: 'green'}
>>> signals[2]
'green'
>>> signals[88]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 88
```

$D[k] = v$

If dictionary D does not have a key-value pair whose key equals k , a new pair is added with key k and value v .

If D already has a key-value pair whose key equals k , the value of that pair is replaced by v .

k in D

A predicate that tests whether D has a key equal to k .

```
>>> roster={1:'Pat', 2:'Ray', 3:'Min'}
>>> 3 in roster
True
>>> 88 in roster
False
```

k not in D

A predicate that tests whether D does *not* have a key equal to k .

```
>>> roster={1:'Pat', 2:'Ray', 3:'Min'}
>>> 3 not in roster
False
>>> 88 not in roster
True
```

`del D[k]`

In Python, `del` is a statement, not a function; see Section 22.3, “The `del` statement: Delete a name or part of a value” (p. 95).

If dictionary D has a key-value pair whose key equals k , that key-value pair is deleted from D . If there is no matching key-value pair, the statement will raise a `KeyError` exception.

```
>>> rgb = {'red': '#ff0000', 'green': '#00ff00', 'blue': '#0000ff'}
>>> del rgb['red']
>>> rgb
{'blue': '#0000ff', 'green': '#00ff00'}
>>> del rgb['cerise']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'cerise'
```

$D.get(k, x)$

If dictionary D has a key equal to x , it returns the corresponding value, that is, it is the same as the expression “ $D[x]$ ”.

However, if D has no key-value pair for key k , this method returns the default value x . The second argument is optional; if omitted, and D has no key equal to k , it returns `None`.

```
>>> roster={1:'Pat', 2:'Ray', 3:'Min'}
>>> roster.get(2)
'Ray'
>>> v = roster.get(8)
>>> print v
None
>>> roster.get(2, 'Not found')
'Ray'
>>> roster.get(8, 'Not found')
'Not found'
```

D .has_key(k)

A predicate that returns `True` if D has a key k .

```
>>> signals = {0: 'red', 1: 'yellow', 2: 'green'}
>>> signals.has_key(1)
True
>>> signals.has_key(88)
False
```

D .items()

Returns the contents of dictionary D as a list of two-element tuples (k, v) , in no particular order.

```
>>> signals = {0: 'red', 1: 'yellow', 2: 'green'}
>>> signals.items()
[(0, 'red'), (1, 'yellow'), (2, 'green')]
```

D .iteritems()

Returns an iterator that generates the values from dictionary D as a sequence of two-element tuples (k, v) . See Section 24.2, “Iterators: Values that can produce a sequence of values” (p. 111).

```
>>> roster={1:'Pat', 2:'Ray', 3:'Min'}
>>> rosterScan = roster.iteritems()
>>> for n, name in rosterScan:
...     print "{0:04d}: {1}".format(n, name)
...
0001: Pat
0002: Ray
0003: Min
```

D .iterkeys()

Returns an iterator that generates the keys from dictionary D . See Section 24.2, “Iterators: Values that can produce a sequence of values” (p. 111).

```
>>> roster={1:'Pat', 2:'Ray', 3:'Min'}
>>> nScan = roster.iterkeys()
>>> for n in nScan:
...     print n,
...
1 2 3
```

D.itervalues()

Returns an iterator that generates the values from dictionary *D*. See Section 24.2, “Iterators: Values that can produce a sequence of values” (p. 111).

```
>>> roster={1:'Pat', 2:'Ray', 3:'Min'}
>>> nameScan = roster.itervalues()
>>> for name in nameScan:
...     print name,
...
Pat Ray Min
```

D.keys()

Returns a list of the key values in dictionary *D*, in no particular order.

```
>>> signals = {0: 'red', 1: 'yellow', 2: 'green'}
>>> signals.keys()
[1, 0, 2]
```

D.popitem()

Returns an arbitrary entry from dictionary *D* as a (*key*, *value*) tuple, and also removes that entry. If *D* is empty, raises a `KeyError` exception.

```
>>> roster={1:'Pat', 2:'Ray', 3:'Min'}
>>> roster.popitem()
(1, 'Pat')
>>> roster
{2: 'Ray', 3: 'Min'}
```

D.setdefault(*k*, *x*)

If dictionary *D* has a key equal to *k*, this method returns the corresponding value *D*[*k*].

If *D* has no key equal to *k*, the method returns the default value *x*. However, unlike the `.get()` method, it *also* creates a new key-value pair (*k*, *x*) in *D*.

As with the `.get()` method, the second argument is optional, and defaults to the value `None`.

D.values()

Returns a list of the values from key-value pairs in dictionary *D*, in no particular order. However, if you call both the `.items()` and `.values()` methods of a dictionary without changing that dictionary's contents between those calls, Python guarantees that the ordering of the two results will be the same.

```
>>> signals = {0: 'red', 1: 'yellow', 2: 'green'}
>>> signals.values()
['yellow', 'red', 'green']
>>> signals.keys()
[1, 0, 2]
```

D.update(*D*₂)

Merge the contents of dictionary *D*₂ into dictionary *D*. For any key-value pairs that have the same key in both *D* and *D*₂, the value for that key in *D* after this operation will be the value from *D*₂, not the value from *D*.

```
>>> roster={1:'Pat', 2:'Ray', 3:'Min'}
>>> newer={3:'Bev', 4:'Wes'}
>>> roster.update(newer)
```

```
>>> roster
{1: 'Pat', 4: 'Wes', 2: 'Ray', 3: 'Bev'}
>>> newer
{3: 'Bev', 4: 'Wes'}
```

16.2. Dictionary comprehensions

New in Python 2.7 are dictionary comprehensions: a construct that allows you to build a dictionary dynamically, somewhat like Section 11.2, “List comprehensions” (p. 38). Here is the general form:

```
{ ek: ev
  for v1 in s1
  for v2 in s2
  ...
  if c }
```

As with list comprehensions, you provide one or more `for` clauses and an optional `if` clause. For all possible combinations of the values in the `for` clauses that have a true value for the `if` clause, two expressions e_k and e_v are evaluated, and a new dictionary entry is added with key e_k and value e_v .

Here is an example. The Wikipedia article on the game of Scrabble¹⁶ gives the Scrabble score for each letter of the alphabet.

What we would like is a dictionary whose keys are letters, and each related value is the score. However, the Wikipedia article shows the score values grouped by score: the 1's together, the 2's together, and so on. So, to make it easy to check that we have entered the right score values and letters, we can use a list of tuples, where the first element of each tuple is the score and the second element is a string of all the letters with that score. We can then convert that list to the desired dictionary using a dictionary comprehension.

```
>>> scrabbleTuples = [ (1, "EAOINRTLSU"), (2, "DG"), (3, "BCMP"),
...                   (4, "FHVWY"), (5, "K"), (8, "JX"), (10, "QZ")]
>>> scrabbleMap = { letter: score
...                for score, letterList in scrabbleTuples
...                for letter in letterList }
>>> scrabbleMap['A']
1
>>> scrabbleMap['Z']
10
```

Evaluating the set comprehension proceeds as follows.

1. In the first `for` clause, the first tuple is unpacked, setting `score` to 1 and `letterList` to "EAOINRTLSU".
2. In the second `for` clause, `letter` is set to "E".
3. A new entry is added to the dictionary, with key "E" and value 1.
4. In the second `for` clause, `letter` is set to "A".
5. A new entry is added with key "A" and value 1.

Execution proceeds in this manner until all the `for` clauses are complete. Then the name `scrabbleMap` is bound to the resulting dictionary.

¹⁶ http://en.wikipedia.org/wiki/Scrabble_letter_distributions

17. Type file: Input and output files

To open a file, use this general form:

```
f = open(name[, mode[, bufsize]])
```

name

The path name of the file to be opened, as a string.

mode

An optional string specifying what you plan to do with the file. If omitted, you will get read access to the file. In general the value consists of three parts:

- General mode, one of:

r	Read access. The file must already exist. You will not be allowed to write to it.
w	Write access. If there is no file by this name, a new one will be created.
<div style="border: 1px solid gray; padding: 5px;">Important <i>If there is an existing file, it will be deleted!</i></div>	
a	Append access. If there is a file by this name, your initial position will be at the end of the file, and you will be allowed to write (and read). If there is no file by this name, a new one will be created. On some systems, all writes to a file with append mode are added at the end of the file, regardless of the current file position.

- If you plan to modify the file, append a “+” next.

For example, mode “r+” puts you at the beginning of an existing file and allows you to write to the file anywhere.

Mode “w+” is the same as “w”: it deletes an existing file if there is any, then creates a new file and gives you write access.

Mode “a+” allows you to write new data at the end of an existing file; if no file by this name exists, it will create a new one.

- If you are handling binary data, as opposed to lines of text, add “b” at the end of the *mode* string.
- For modes beginning with ‘r’, you may append a capital ‘U’ to request universal newline treatment. This is handy when you are reading files made on a platform with different line termination conventions.

When reading lines from a file opened in this way, any line terminator (‘\n’, ‘\r’, or ‘\r\n’) will appear in the return value as the standard ‘\n’. Also, files so opened will have an attribute named `.newlines`; this attribute will be `None` initially, but after any line terminators have been read, it will be a tuple containing all the different line terminator strings seen so far.

bufsize

Buffer size: this affects when physical device writes are done, compared to write operations that your program performs.

- In most cases you will probably want to omit this argument. The default is to use line buffering for terminal-type devices, or some system default for other devices.
- Use 0 to force unbuffered operation. This may be inefficient, but any file writes are performed immediately.

- Use 1 for line buffering: output lines are written whenever you write a line terminator such as `'\n'`.
- Use larger values to specify the actual size of the buffer.
- Use a negative value to request the system defaults.

If you are reading text files, and you don't want to worry about the variety of line termination protocols, you may use a *mode* value of `"U"` for "universal line terminator mode." In this mode, input lines may be terminated with either carriage return (`'\r'`), newline (`'\n'`), or both, but the lines you receive will always be terminated with a single newline. (Exception: If the last line is unterminated, the string you get will also be unterminated.)

There are a number of potential error conditions. For modes starting with `"r"`, the file must exist before you open it. Also, you must have access according to the underlying operating system. For example, in Linux environments, you must have read access to read a file, and you must have write access to modify or delete a file. These sorts of failures will raise an `IOError` exception.

A file is its own iterator (see Section 24.2, "Iterators: Values that can produce a sequence of values" (p. 111)). Hence, if you have a file `inFile` opened for reading, you can use a `for` loop that looks like this to iterate over the lines of the file:

```
for line in inFile:
    ...
```

The variable `line` will be set to each line of the file in turn. The line terminator character (if any) will be present in that string.

Other aspects of files:

- Every open file has a *current position*. Initially, this will `0L` if you opened it for reading or writing, or the size of the file if you opened it with append access. Each write or read operation moves this position by the amount read or written. You can also query or set the file position; see Section 17.1, "Methods on file objects" (p. 55).
- Files may use a technique called *buffering*. Because physical access to some storage media (such as disk drives) takes a relatively long time, Python may employ a storage area called a *buffer* as a holding area for data being input or output.

For example, if you are writing data to a disk file, Python may keep the data in the file's buffer area until the buffer is full and only then actually write it to the physical disk. There are various techniques for controlling this behavior; see Section 17.1, "Methods on file objects" (p. 55).

17.1. Methods on file objects

Use these methods on an open file instance *F*.

***F*.close()**

Close file *F*. Any unwritten data in the buffer will be flushed. No further operations will be allowed on the file unless it is reopened with the `open()` function.

***F*.flush()**

For buffered files, you can use this method to make sure that all data written to the file has been physically transmitted to the storage medium. Closing a file will also flush the buffers. Avoid using this method unless you really need it, as it may make your program less efficient.

***F*.isatty()**

A predicate that tests whether *F* is a terminal; "tty" is an ancient term that originally meant "Teletype", but has come to mean any terminal or simulated terminal.

***F*.read(*n*)**

Read the next *n* characters from *F* and return them as a string.

If there are fewer than *n* characters remaining after your current position, you will get all remaining characters. If you are at the end of the file, you will get back an empty string ('').

The argument is optional. If omitted, you will get the entire remaining contents of the file as a string.

***F*.readline(*maxlen*)**

Read the next text line from *F* and return it as a string, including the line terminator if any.

If you need to limit the maximum size of incoming lines, pass that size limit as the optional *maxlen* argument. The default is to return a line of any size (subject to memory limitations). If the line exceeds *maxlen*, the file position will be left pointing to the first unread character of that line.

***F*.readlines()**

Read all remaining lines from *F* and return them as a list of strings, including line terminator characters if any.

***F*.seek(*offset*, *whence*)**

Change the file's current position. The value of *whence* determines how the *offset* value is used to change the position:

- 0: Set the position to *offset* bytes after the beginning of the file.
- 1: Move the current position *offset* bytes toward the end of the file.
- 2: Move the current position *offset* bytes relative to the end of the file.

For example, for a file *f*, this operation would set the position to 4 bytes before the end of the file:

```
f.seek(-4, 2)
```

***F*.tell()**

This method returns the current file position relative to the beginning as a `long` value.

```
>>> f=open('roundtable', 'w')
>>> f.write('Bedevere')
>>> f.tell()
8L
>>> f.seek(2L)
>>> f.tell()
2L
```

***F*.truncate([*pos*])**

Remove any contents of *F* past position *pos*, which defaults to the current position.

***F*.write(*s*)**

Write the contents of string *s* to file *F*. This operation will not add terminator characters; if you want newlines in your file, include them in the string *s*.

***F*.writelines(*S*)**

For a sequence *S* containing strings, write all those strings to *F*. No line terminators will be added; you must provide them explicitly if you want them.

18. None: The special placeholder value

Python has a unique value called `None`. This special null value can be used as a placeholder, or to signify that some value is unknown.

In conversational mode, any expression that evaluates to `None` is not printed. However, if a value of `None` is converted to a string, the result is the string `'None'`; this may happen, for example, in a `print` statement.

```
>>> x = None
>>> x
>>> print x
None
```

The value `None` is returned from any function that executes a `return` statement with no value, or any function after it executes its last line if that last line is not a `return` statement.

```
>>> def useless():
...     print "Useless!"
...
>>> useless()
Useless!
>>> z=useless()
Useless!
>>> z
>>> print z
None
>>> def boatAnchor():
...     pass
...
>>> x=boatAnchor()
>>> x
>>> print x
None
```

19. Operators and expressions

Python's operators are shown here from highest precedence to lowest, with a ruled line separating groups of operators with equal precedence:

Table 6. Python operator precedence

<code>(E)</code>	Parenthesized expression or tuple.
<code>[E, ...]</code>	List.
<code>{key:value, ...}</code>	Dictionary or set.
<code>`...`</code>	Convert to string representation.
<code>x.attribute</code>	Attribute reference.
<code>x[...]</code>	Subscript or slice; see Section 8.1, “Operations common to all the sequence types” (p. 12).
<code>f(...)</code>	Call function <i>f</i> .
<code>x**y</code>	<i>x</i> to the <i>y</i> power.
<code>-x</code>	Negation.
<code>~x</code>	Bitwise not (one's complement).
<code>x*y</code>	Multiplication.
<code>x/y, x//y</code>	Division. The “//” form discards the fraction from the result. For example, “13.9//5.0” returns the value 2.0.
<code>x%y</code>	Modulo (remainder of <i>x/y</i>).
<code>x+y</code>	Addition, concatenation.
<code>x-y</code>	Subtraction.
<code>x<<y</code>	<i>x</i> shifted left <i>y</i> bits.
<code>x>>y</code>	<i>x</i> shifted right <i>y</i> bits.
<code>x&y</code>	Bitwise and.
<code>x^y</code>	Bitwise exclusive or.
<code>x y</code>	Bitwise or.
<code>x<y, x<=y, x>y, x>=y, x!=y, x==y</code>	Comparisons. These operators are all predicates; see Section 19.1, “What is a predicate?” (p. 58).
<code>x in y, x not in y</code>	Test for membership.
<code>x is y, x is not y</code>	Test for identity.
<code>not x</code>	Boolean “not.”
<code>x and y</code>	Boolean “and.”
<code>x or y</code>	Boolean “or.”

19.1. What is a predicate?

We use the term *predicate* to mean any Python function that tests some condition and returns a Boolean value.

For example, `x < y` is a predicate that tests whether *x* is less than *y*. For example, `5 < 500` returns `True`, while `5 >= 500` returns `False`.

19.2. What is an iterable?

To *iterate over* a sequence means to visit each element of the sequence, and do some operation for each element.

In Python, we say that a value is an *iterable* when your program can iterate over it. In short, an iterable is a value that represents a sequence of one more values.

All instances of Python's sequence types are iterables. These types may be referred to as *container types*: a `unicode` string is a container for 32-bit characters, and lists and tuples are general-purpose containers that can contain any sequence.

One of the most common uses for an iterable is in a `for` statement, where you want to perform some operation on a sequence of values. For example, if you have a tuple named `celsiuses` containing Celsius temperatures, and you want to print them with their Fahrenheit equivalents, and you have written a function `cToF()` that converts Celsius to Fahrenheit, this code does it:

```
>>> def cToF(c): return c*9.0/5.0 + 32.0
...
>>> celsiuses = (0, 20, 23.6, 100)
>>> for celsius in celsiuses:
...     print "{0:.1f} C = {1:.1f} F".format(celsius, cToF(celsius))
...
0.0 C = 32.0 F
20.0 C = 68.0 F
23.6 C = 74.5 F
100.0 C = 212.0 F
```

However, Python also supports mechanisms for *lazy evaluation*: a piece of program that acts like a sequence, but produces its contained values one at a time.

Keep in mind that the above code works exactly the same if `celsiuses` is an iterator (see Section 24.2, “Iterators: Values that can produce a sequence of values” (p. 111)). You may find many uses for iterators in your programs. For example, `celsiuses` might be a system that goes off and reads an actual thermometer and returns the readings every ten seconds. In this application, the code above doesn't care where `celsiuses` gets the values, it cares only about how to convert and print them.

19.3. Duck typing, or: what is an interface?

When I see a bird that walks like a duck and swims like a duck and quacks like a duck,
I call that bird a duck.

—James Whitcomb Riley

The term *duck typing* comes from this quote. In programming terms, this means that the important thing about a value is what it can do, not its type. As the excellent Wikipedia article on duck typing¹⁷ says, “Simply stated: provided you can perform the job, we don't care who your parents are.”

One common example of duck typing is in the Python term “file-like object”. If you open a file for reading using the `open()` function, you get back a value of type `file`:

```
>>> inFile = open('input')
>>> type(inFile)
<type 'file'>
```

¹⁷ http://en.wikipedia.org/wiki/Duck_typing

Let's suppose that you write a function called `numberIt()` that takes a readable `file` as an argument and prints the lines from a file preceded by five-digit line numbers. Here's the function and an example of its use:

```
>>> def numberIt(f):
...     for lineNo, line in enumerate(f):
...         print "{0:05d} {1}".format(lineNo, line.rstrip())
...
>>> numberIt(inFile)
00000 Kant
00001 Heidegger
00002 Hume
```

The way you have written the `numberIt()` function, it works for files, but it also works for any iterable.

Thus, when you see the statement that some Python feature works with a “file-like object,” that means that the object must have an interface like that of the `file` type; Python doesn't care about the type, just the operations that it supports.

In practice, the `enumerate()` function works with any iterable, so your function will also work with any iterable:

```
>>> numberIt(['Kant', 'Heidegger', 'Hume'])
00000 Kant
00001 Heidegger
00002 Hume
```

So in Python when we say that we expect some value to have an *interface*, we mean that it must provide certain methods or functions, but the actual type of the value is immaterial.

More formally, when we say that a value supports the *iterable interface*, that value must provide either of the following features:

- A `.__getitem__()` method as described in Section 26.3.16, “`.__getitem__()`: Get one item from a sequence or mapping” (p. 134).
- A `.__iter__()` method as described in Section 26.3.17, “`.__iter__()`: Create an iterator” (p. 134).

19.4. What is the locale?

In order to accommodate different character encodings, your system may have a *locale* setting that specifies a preferred character set.

In the USA, most systems use the ASCII¹⁸ encoding. Good application code should be written in a way that does not depend on this encoding to deal with cultural issues.

For general information on handling locale issues, see the documentation for the `locale` module¹⁹.

20. Basic functions

Python has a lot of built-in functions. This section describes the ones that most people use most of the time. If you are interested in exploring some of the more remote corners of Python, see Section 21, “Advanced functions” (p. 79).

¹⁸ <http://en.wikipedia.org/wiki/ASCII>

¹⁹ <http://docs.python.org/library/locale.html>

20.1. `abs ()`: Absolute value

To find the absolute value of a number x :

```
abs(x)
```

If x is negative, the function returns $-x$; otherwise it returns x .

For complex values, the function returns the magnitude, that is, the square root of $(x.\text{real}^{**2}+x.\text{imag}^{**2})$.

```
>>> abs(-33)
33
>>> abs(33)
33
>>> abs(0)
0
>>> abs(complex(1,5))
5.0990195135927845
```

20.2. `all ()`: Are all the elements of an iterable true?

A predicate that tests whether all the elements of some iterable are considered `True`. If any elements are not already type `bool`, they are converted to Boolean values using the `bool ()` built-in function.

```
>>> all([True, 14, (88,99)])
True
>>> all([True, 14, (88,99), None])
False
```

20.3. `any ()`: Are any of the members of an iterable true?

This function, applied to some iterable, is a predicate that tests whether any of the elements of that iterable are `True`. If any element is not already type `bool`, it is converted to a Boolean value using the `bool ()` built-in function.

```
>>> noneTrue = (0, 0.0, (), None)
>>> any(noneTrue)
False
>>> someTrue = (0, 0.0, (88,), 'penguin')
>>> any(someTrue)
True
```

20.4. `bin ()`: Convert to binary

This function takes an integer argument and returns a string that represents that number in binary (base 2) starting with `'0b'`.

```
>>>
bin(7)
'0b111'
```

```
>>> bin(257)
'0b100000001'
```

20.5. `bool()`: Convert to Boolean

This function takes any value x and converts it to a Boolean (true or false) value.

For the list of values that are considered True or False, see Section 7.3, “Type `bool`: Boolean truth values” (p. 9). Examples:

```
>>> bool(0)
False
>>> bool(0.0)
False
>>> bool(0L)
False
>>> bool(0j)
False
>>> bool('')
False
>>> bool([])
False
>>> bool(())
False
>>> bool({})
False
>>> bool(None)
False
>>> bool(1)
True
>>> bool(15.9)
True
>>> bool([0])
True
>>> bool((None,))
True
>>> bool({None: False})
True
```

20.6. `bytearray()`: Create a byte array

See Section 14, “The `bytearray` type” (p. 42).

20.7. `chr()`: Get the character with a given code

For arguments n in the range 0 to 255, this function returns a one-character string containing the character that has code n . Compare Section 20.31, “`ord()`: Find the numeric code for a character” (p. 72).

```
>>> chr(65)
'A'
```

```
>>> chr(0)
'\x00'
```

20.8. `cmp()`: Compare two values

This function compares the values of two arguments x and y :

```
cmp(x, y)
```

The return value is:

- A negative number if x is less than y .
- Zero if x is equal to y .
- A positive number if x is greater than y .

The built-in `cmp()` function will typically return only the values -1, 0, or 1. However, there are other places that expect functions with the same calling sequence, and those functions may return other values. It is best to observe only the sign of the result.

```
>>> cmp(2,5)
-1
>>> cmp(5,5)
0
>>> cmp(5,2)
1
>>> cmp('aardvark', 'aardwolf')
-1
```

20.9. `complex()`: Convert to complex type

To create a complex number from a real part R and a complex part I :

```
complex(R, I)
```

Both arguments are optional.

- With two arguments, both arguments must be numbers of any numeric type.

With one numeric argument, it returns a complex number with real part R and an imaginary part of zero.

To convert a complex number in string form, pass that string as the only argument. If the string is not a valid complex number, the function raises a `ValueError` exception.

- If called with no arguments, it returns a complex zero.

Examples:

```
>>> complex(-4.04, 3.173)
(-4.04+3.173j)
>>> complex(-4.04)
(-4.04+0j)
>>> complex()
0j
>>> c1=4+5j
>>> c2=6+7j
```

```
>>> complex(c1, c2) # Equals (4+5j)+(6+7j)j = 4+5j+6j-7
(-3+11j)
>>> c1 + c2*1.0j
(-3+11j)
```

20.10. dict(): Convert to a dictionary

This function creates a new dictionary from its arguments. The general form is:

```
dict(v, k0=v0, k1=v1, ...)
```

That is, there may be one optional positional argument or any number of keyword arguments.

- If you supply no arguments, you get a new, empty dictionary.
- If one positional argument is supplied, it must be a iterable containing two-element iterables. Each two-element iterable becomes one key-value pair of the result.

```
>>> dict()
{}
>>> dict ( [ (0, 'stop'), (1, 'go') ] )
{0: 'stop', 1: 'go'}
>>> dict (('y', 'boy'), ('x', 'girl'))
{'y': 'boy', 'x': 'girl'}
```

- If you supply any keyword arguments, each keyword becomes a key in the resulting dictionary, and that argument's value becomes the corresponding value of that key-value pair.

```
>>> dict(bricks='sleep', keith='maniac', rj='gumby')
{'bricks': 'sleep', 'keith': 'maniac', 'rj': 'gumby'}
```

20.11. divmod(): Quotient and remainder

```
divmod(x, y)
```

Sometimes you want both the quotient and remainder when dividing x by y . This function returns a tuple (q, r) , where q is the quotient and r is the remainder.

If either x or y is a `float`, the returned value q is the whole part of the quotient, and the returned r is computed as $x - (r*d)$.

Examples:

```
>>> divmod(13, 5)
(2, 3)
>>> divmod(1.6, 0.5)
(3.0, 0.10000000000000009)
```

20.12. enumerate(): Step through indices and values of an iterable

Given an iterable S , `enumerate(S)` produces an iterator that iterates over the pairs of values $(i, S[i])$, for i having the values in `range(len(S))`. For more information on iterators, see Section 24.2, "Iterators: Values that can produce a sequence of values" (p. 111).

```

>>> L = ['Ministry', 'of', 'Silly', 'Walks']
>>> for where, what in enumerate(L):
...     print "[{0}] {1}".format(where, what)
...
[0] Ministry
[1] of
[2] Silly
[3] Walks

```

If you would like the numbers to start at a different origin, pass that origin as the second argument to the `enumerate()` function. You will still get all the elements of the sequence, but the numbers will start at the value you provide. (Python 2.6 and later versions only.)

```

>>> for where, what in enumerate(L, 1):
...     print "[{0}] {1}".format(where, what)
...
[1] Ministry
[2] of
[3] Silly
[4] Walks

```

20.13. `file()`: Open a file

This function is identical to the `open()` function; for details, see Section 17, “Type `file`: Input and output files” (p. 54).

20.14. `filter()`: Extract qualifying elements from an iterable

This function is useful for removing some of the elements of an iterable. You must provide a filtering function that takes one argument and returns a `bool` value. Here is the calling sequence:

```
filter(f, S)
```

The filtering function *f* is the first argument. It is applied to every element of some iterable *S*. The result is a new sequence containing only those elements *x* of *S* for which *f*(*x*) returned `True`.

- If *f* is a string or tuple, the result has the same type, otherwise the result is a `list`.
- If *f* is `None`, you get a sequence of the true elements of *S*. In this case, the filtering function is effectively the `bool()` function.

```

>>> def isOdd(x):
...     if (x%2) == 1: return True
...     else: return False
...
>>> filter(isOdd, [88, 43, 65, -11, 202])
[43, 65, -11]
>>> filter(isOdd, (1, 2, 4, 6, 9, 3, 3))
(1, 9, 3, 3)
>>> def isLetter(c):
...     return c.isalpha()
...
>>> filter(isLetter, "01234abcdeFGHIJ*(&!^)")

```

```
'abcdeFGHIJ'
>>> maybes = [0, 1, (), (2,), 0.0, 0.25]
>>> filter(None, maybes)
[1, (2,), 0.25]
>>> filter(bool, maybes)
[1, (2,), 0.25]
```

20.15. float(): Convert to float type

Converts a value to type `float`. The argument must be a number, or a string containing a numeric value in string form (possibly surrounded by whitespace). If the argument is not a valid number, this function will raise a `ValueError` exception. If no argument is given, it will return `0.0`.

```
>>> float()
0.0
>>> float(17)
17.0
>>> float(' 3.1415 ')
3.1415000000000002
>>> print float('6.0221418e23')
6.0221418e+23
>>> float('142x')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for float(): 142x
```

20.16. format(): Format a value

This function converts a value to a formatted representation. The general form is:

```
format(value[, spec])
```

- For built-in types, the *spec* has the syntax as that described in Section 9.4, “The string `.format()` method” (p. 22) between “:” and the closing “}”.

```
>>> x = 34.56
>>> format(x, '9.4f')
' 34.5600'
>>> '{0:9.4f}'.format(x)
' 34.5600'
```

- You can define how this function works for a user-defined class by defining the special method described in Section 26.3.13, “`__format__`: Implement the `format()` function” (p. 133).
- If the *spec* argument is omitted, the result is the same as `str(value)`.

20.17. frozenset(): Create a frozen set

This function is used to create a new `frozenset` value: an immutable set. General form:

```
frozenset(S)
```

This function converts an existing iterable *S* to a `frozenset`. The argument is optional; if omitted, you get a frozen empty set.

```
>>> frozenset()
frozenset([])
>>> frozenset('aeiou')
frozenset(['a', 'i', 'e', 'u', 'o'])
>>> frozenset([0, 0, 0, 44, 0, 44, 18])
frozenset([0, 18, 44])
```

For more information, see Section 15, “Types `set` and `frozenset`: Set types” (p. 43).

20.18. `hex()`: Convert to base 16

Given an integer, this function returns a string displaying that value in hexadecimal (base 16).

```
>>> hex(15)
'0xf'
>>> hex(255)
'0xff'
>>> hex(256)
'0x100'
>>> hex(1325178541275812780L)
'0x1263fadcb8b713ac'
```

See also Section 9.4, “The string `.format()` method” (p. 22): hexadecimal conversion is supported by specifying a *type* code of `'x'` or `'X'`.

20.19. `int()`: Convert to `int` type

To convert a number of a different type to `int` type, or to convert a string of characters that represents a number:

```
int(ns)
```

where *ns* is the value to be converted. If *ns* is a `float`, the value will be truncated, discarding the fraction.

If you want to convert a character string *s*, expressed in a radix (base) other than 10, to an `int`, use this form, where *b* is an integer in the range [2, 36] that specifies the radix.

```
int(s, b)
```

Examples:

```
>>> int(43L)
43
>>> int(True)
1
>>> int(False)
0
>>> int(43.89)
43
>>> int("69")
```

```

69
>>> int('77', 8)
63
>>> int('7ff', 16)
2047
>>> int('10101', 2)
21

```

20.20. `input()`: Read an expression from the user

This function asks the user to type something, then evaluates it as a Python expression. Here is the general form:

```
input([prompt])
```

If you supply a string as the optional *prompt* argument, that string will be written to the user before the input is read.

In any case, the result is the value of the expression. Of course, if the user types something that isn't a valid Python expression, the function will raise an exception.

```

>>> input()
2+2
4
>>> print "The answer was '{0}'".format(input())
2+3*4
The answer was '14'.
>>> print "The answer was '{0}'".format(input("Type an expression:"))
Type an expression:2+3*4
The answer was '14'.
>>> print input()
1/0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero

```

20.21. `iter()`: Produce an iterator over a sequence

Given a sequence *S*, this function returns an iterator that generates the elements of the sequence. For more information, see Section 24.2, "Iterators: Values that can produce a sequence of values" (p. 111).

```

>>> listWalker = iter ( [23, 47, 'hike'] )
>>> for x in listWalker: print x,
...
23 47 hike

```

In general, the calling sequence is:

```
iter(s[, sentinel])
```

- If the *sentinel* argument is omitted, the first argument must be either a sequence value that implements the `.__getitem__()` method or an instance of a class that has the `.__iter__()` method.

- If you provide a *sentinel* argument, the *s* argument must be callable. The iterator returned will call *s()* with no arguments and generate the values it returns until the return value equals *sentinel*, at which point it will raise `StopIteration`.

20.22. `len()`: Number of elements

Given a sequence or dictionary, this function returns the number of elements.

```
>>> len('')
0
>>> len([23, 47, 'hike'])
3
>>> len({1: 'foot', 2: 'shoulder', 'feather': 'rare'})
3
```

20.23. `list()`: Convert to a list

This function creates a new list. The argument *x* may be any iterable (see Section 19.2, “What is an iterable?” (p. 59)).

```
>>> list(('Bentham', 'Locke', 'Hobbes'))
['Bentham', 'Locke', 'Hobbes']
>>> list("Bruce")
['B', 'r', 'u', 'c', 'e']
>>> list((42,))
[42]
>>> list()
[]
```

20.24. `long()`: Convert to long type

This function works exactly the same way as Section 20.19, “`int()`: Convert to `int` type” (p. 67), except that it produces a result of type `long`.

```
>>> long(43)
43L
>>> long(43.889)
43L
>>> long('12345678901234567890123457890')
12345678901234567890123457890L
>>> long('potrzebie456', 36)
3381314581245790842L
```

20.25. `map()`: Apply a function to each element of an iterable

The purpose of this function is to perform some operation on each element of an iterable. It returns a list containing the result of those operations. Here is the general form:

```
map(f, S)
```

- *f* is a function that takes one argument and returns a value.

- *S* is any iterable.

```
>>> def add100(x):
...     return x+100
...
>>> map(add100, (44,22,66))
[144, 122, 166]
```

To apply a function with multiple arguments to a set of sequences, just provide multiple iterables as arguments, like this.

```
>>> def abc(a, b, c):
...     return a*10000 + b*100 + c
...
>>> map(abc, (1, 2, 3), (4, 5, 6), (7, 8, 9))
[10407, 20508, 30609]
```

If you pass `None` as the first argument, Python uses the identity function to build the resulting list. This is useful if you want to build a list of tuples containing items from two or more iterables.

```
>>> map(None, range(3))
[0, 1, 2]
>>> map(None, range(3), 'abc', [44, 55, 66])
[(0, 'a', 44), (1, 'b', 55), (2, 'c', 66)]
```

20.26. `max()`: Largest element of an iterable

Given an iterable *S* that contains at least one element, `max(S)` returns the largest element of the sequence.

```
>>> max('blimey')
'y'
>>> max([-505, -575, -144, -288])
-144
>>> max([])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: max() arg is an empty sequence
```

You can also pass multiple arguments, and `max()` will return the largest. In the example below, 'cheddar' is the largest because lowercase letters have higher codes than uppercase letters.

```
>>> max('Gumby', 'Lambert', 'Sartre', 'cheddar')
'cheddar'
```

If you want to redefine the comparison function, you may provide a keyword argument `key=f`, where *f* is a function that takes one argument and returns a value suitable for comparisons. In this example, we use the `.upper()` method of the `str` class to compare the uppercased strings, then return the original string whose uppercased value is largest.

```
>>> max('Gumby', 'Lambert', 'Sartre', 'cheddar', key=str.upper)
'Sartre'
```

20.27. `min()`: Smallest element of an iterable

Given an iterable S containing at least one element, `max(S)` returns the largest element of S .

```
>>> min('blimey')
'b'
>>> min ( [-505, -575, -144, -288] )
-575
```

You may also pass multiple arguments, and the `min()` function will return the smallest.

```
>>> min(-505, -575, -144, -288)
-575
```

If you would like to use a different function to define the ordering, specify that function as a keyword argument `key=f`, where f is a function that takes one argument and returns a value suitable for comparisons. In this example, we want to order the values based on their inverse.

```
>>> def rev(x):
...     return -x
...
>>> min(-505, -575, -144, -288, key=rev)
-144
```

20.28. `next()`: Call an iterator

This function attempts to get the next value from some iterator I (see Section 24.2, “Iterators: Values that can produce a sequence of values” (p. 111)). (New in version 2.6.)

```
next(I[, default])
```

- If the iterator produces another value, that value is returned by this function.
- If the iterator is exhausted and you provide a *default* value, that value is returned.
- If the iterator is exhausted and you do not provide a default value, the `next()` function raises a `StopIteration` exception.

Here is an example.

```
>>> it = iter(xrange(0,2))
>>> next(it, 'Done')
0
>>> next(it, 'Done')
1
>>> next(it, 'Done')
'Done'
>>> next(it)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

20.29. oct () : Convert to base 8

Given a number n , `oct(n)` returns a string containing an octal (base 8) representation of n . Consistent with Python's convention that any number starting with a zero is considered octal, the result of this function will always have a leading zero.

```
>>> oct(0)
'0'
>>> oct(127)
'0177'
```

See also Section 9.4, “The string `.format()` method” (p. 22): octal conversion is supported by specifying *type* code 'o'.

20.30. open () : Open a file

Open a file. For the calling sequence, see Section 17, “Type file: Input and output files” (p. 54).

20.31. ord () : Find the numeric code for a character

Given a string s containing a single character, `ord(s)` returns that character's numeric code. Compare Section 20.7, “`chr()`: Get the character with a given code” (p. 62).

```
>>> ord('A')
65
>>> ord('\x00')
0
>>> ord(u'\u262e')
9774
>>> hex(9774)
'0x262e'
```

20.32. pow () : Exponentiation

There are two ways to compute x^y in Python. You can write it as “ $x**y$ ”. There is also a function that does the same thing:

```
pow(x, y)
```

For integer arithmetic, the function also has a three-argument form that computes $x^y \% z$, but more efficiently than if you used that expression:

```
pow(x, y, z)
```

Examples:

```
>>> 2**4
16
>>> pow(2,4)
16
>>> pow(2.5, 4.5)
61.763235550163657
>>> (2**9)%3
```

```
2
>>> pow(2,9,3)
2
```

20.33. range () : Generate an arithmetic progression as a list

This function generates a list containing the values of an arithmetic progression, that is, a sequence of numbers such that the difference between adjacent numbers is always the same. There are three forms:

range(*n*)

Returns the list $[0, 1, 2, \dots, n-1]$. Note that the result never includes the value n .

range(*start*, *stop*)

Returns the list $[start, start+1, start+2, \dots, stop-1]$. The result never includes the *stop* value.

range(*start*, *stop*, *step*)

Returns the list $[start, start+step, start+2*step, \dots]$, up to but not including the value of *stop*. The value of *step* may be negative.

Examples:

```
>>> range(4)
[0, 1, 2, 3]
>>> range(4,9)
[4, 5, 6, 7, 8]
>>> range(10,104,10)
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
>>> range(5, -1, -1)
[5, 4, 3, 2, 1, 0]
```

20.34. raw_input () : Prompt and read a string from the user

To prompt for keyboard entry, use this function:

```
raw_input(p)
```

The argument *p* is a prompt string that is written to standard output. Then a line is read from standard input and returned as a string, without its trailing newline character.

```
>>> party = raw_input('Party affiliation: ')
Party affiliation: Slightly silly.
>>> party
'Slightly silly.'
```

If the user signals end of input (e.g., with *Control-D* under Unix), the function raises an `E0FError` exception.

20.35. reduce () : Sequence reduction

The idea of *reduction* comes from the world of functional programming. There is a good introductory article on this concept in Wikipedia²⁰. In simple terms, a function of two arguments is applied repeatedly to the elements of an iterable to build up a final value.

- The idea of a “sum of elements of a sequence” is a reduction of those elements using “+” as the function. For example, the +-reduction of [2, 3, 5] is 2+3+5 or 10.
- Similarly, the product of a series of numbers is a reduction using the “*” operator: the multiply reduction of [2, 3, 5] is 2*3*5 or 30.

There are two general forms:

```
reduce(f, S)
reduce(f, S, I)
```

- *f* is a function that takes two arguments and returns a value.
- *S* is an iterable.

The result depends on the number of elements in *S*, and whether the initial value *I* is supplied. Let's look first at the case where argument *I* is not supplied.

- If *S* has only one element, the result is *S*[0].
- If *S* has two elements, the result is *f*(*S*[0], *S*[1]).
- If *S* has three elements, the result is *f*(*f*(*S*[0], *S*[1]), *S*[2]).
- If *S* has four or more elements, *f* is applied first to *S*[0] and *S*[1], then to that result and *S*[2], and so on until all elements are reduced to a single value.
- If *S* is empty and no initial value was provided, the function raises a `TypeError` exception.

If an initial value *I* is provided, the result is the same as `reduce(f, [I]+list(S))`.

Some examples:

```
>>> def x100y(x,y):
...     return x*100+y
...
>>> reduce(x100y, [15])
15
>>> reduce(x100y, [1,2])
102
>>> reduce(x100y, [1,2,3])
10203
>>> reduce(x100y, (), 44)
44
>>> reduce(x100y, [1], 2)
201
>>> reduce(x100y, [1,2], 3)
30102
>>> reduce(x100y, [1,2,3], 4)
4010203
>>> reduce(x100y, [])
Traceback (most recent call last):
```

²⁰ [http://en.wikipedia.org/wiki/Fold_\(higher-order_function\)](http://en.wikipedia.org/wiki/Fold_(higher-order_function))

```
File "<stdin>", line 1, in <module>
TypeError: reduce() of empty sequence with no initial value
```

20.36. reversed () : Produce a reverse iterator

This function, applied to a sequence *S*, returns an iterator that generates the elements of *S* in reverse order (see Section 24.2, "Iterators: Values that can produce a sequence of values" (p. 111)).

```
>>> L=[22,44,88]
>>> backL = reversed(L)
>>> for i in backL:
...     print i,
...
88 44 22
```

The allowable values for *S* include all the types described in Section 8, "Sequence types" (p. 11). It also works for any class provided one of these two conditions is met:

- The method described in Section 26.3.20, "`__reversed__()`: Implement the `reversed()` function" (p. 135).
- If the class has no `__reversed__()` method, `reversed()` will still work provided that instances act like a sequence; that is, the class has a `__len__()` method and a `__getitem__()` method.

20.37. round () : Round to the nearest integral value

To find the integral value nearest to some value *x*, use this function:

```
round(x)
```

The value is returned as a `float`. In the case that the fractional part of *x* is exactly one-half, the returned value is the integer farther from zero. Examples:

```
>>> round ( 4.1 )
4.0
>>> round(4.9)
5.0
>>> round(4.5)
5.0
>>> round(-4.1)
-4.0
>>> round(-4.9)
-5.0
>>> round(-4.5)
-5.0
```

You can also provide an optional second argument that specifies how many digits to retain after the decimal.

```
>>> from math import pi
>>> round(pi)
3.0
>>> print round(pi,1)
3.1
```

```
>>> print round(pi,2)
3.14
>>> print round(pi, 4)
3.1416
>>> round(pi,30)
3.1415926535897931
```

20.38. set(): Create an algebraic set

Use this function to create a `set` value. Here is the general form:

```
set(S)
```

The optional argument `S` is any iterable; the return value is a new `set` instance containing the unique values from `S`. When called with no arguments, this function returns a new, empty set. Examples:

```
>>> empty = set()
>>> empty
set([])
>>> len(empty)
0
>>> swallows=set(['African', 'European'])
>>> swallows
set(['European', 'African'])
>>> len(swallows)
2
>>> set ( (0, 0, 0, 58, 0, 0, 58, 17) )
set([0, 17, 58])
```

For more information about sets, see Section 15, “Types `set` and `frozenset`: Set types” (p. 43).

20.39. sorted(): Sort a sequence

This function, applied to any iterable `S`, produces a new list containing the elements of `S` in ascending order (or some other order you specify).

Here is the general form:

```
sorted(S[, cmp[, key[, reverse]])
```

The `cmp`, `key`, and `reverse` arguments are optional, and have the same meaning as in the `.sort()` method of the `list` type (see Section 11.1, “Methods on lists” (p. 35)).

```
>>> L = ['geas', 'clue', 'Zoe', 'Ann']
>>> sorted(L)
['Ann', 'Zoe', 'clue', 'geas']
>>> def ignoreCase(x,y):
...     return cmp(x.upper(), y.upper())
...
>>> sorted(L, ignoreCase)
['Ann', 'clue', 'geas', 'Zoe']
>>> sorted(L, None, str.upper)
['Ann', 'clue', 'geas', 'Zoe']
```

```
>>> L
['geas', 'clue', 'Zoe', 'Ann']
```

In the first example above, 'Zoe' precedes 'clue', because all uppercase letters are considered to be less than all lowercase letters. The second example shows the use of a *cmp* argument to sort strings as if they were all uppercase; the third example shows how to achieve the same result using the `.upper()` method of the `str` class as the *key* argument. Note in the last line that the original list `L` is unchanged.

20.40. `str()`: Convert to `str` type

To convert any value *x* to a string, use this general form:

```
str(x)
```

For example:

```
>>> str(17)
'17'
>>> str({'boy': 'Relmond', 'girl': 'Wirdley'})
"{'boy': 'Relmond', 'girl': 'Wirdley'}"
```

For general information, see Section 9, "Type `str`: Strings of 8-bit characters" (p. 14).

20.41. `sum()`: Total the elements of a sequence

This function, applied to an iterable *S*, returns the sum of its elements. There are two general forms:

```
sum(S)
sum(S, I)
```

In the second form, the summing process starts with the initial value *I*. Examples:

```
>>> L=[1,2,3,4]
>>> sum(L)
10
>>> sum(L,1000)
1010
>>> sum((), 1000)
1000
```

20.42. `tuple()`: Convert to a tuple

To convert some iterable *S* to a tuple, use this general form:

```
tuple(s)
```

The result will be a new tuple with the same elements as *S* in the same order. For general information, see Section 12, "Type `tuple`: Immutable sequences" (p. 39).

To create an empty tuple, omit the argument. Examples:

```
>>> tuple()
()
```

```

>>> tuple ( ['swallow', 'coconut'] )
('swallow', 'coconut')
>>> tuple ( 'shrubbery' )
('s', 'h', 'r', 'u', 'b', 'b', 'e', 'r', 'y')
>>> tuple ( ['singleton'] )
('singleton',)

```

20.43. `type()`: Return a value's type

This function can be applied to any value. It returns a *type object* corresponding to the type of that value.

For built-in types, the type object is the same as the name of the type: `int`, `str`, `list`, and so on. To test whether a value x is some type T , you can use the predicate “`type(x) is T`”.

If you display a type object in conversational mode, it will look like “<type 'T'>”. Examples:

```

>>> type(i)
<type 'int'>
>>> type(i) is int
True
>>> type([2,4,8]) is list
True

```

20.44. `unichr()`: Convert a numeric code to a Unicode character

Given a number n , this function returns the Unicode character that has code point n . For more on Unicode, see Section 10, “Type unicode: Strings of 32-bit characters” (p. 32).

```

>>> unichr(0)
u'\x00'
>>> unichr(ord('A'))
u'A'
>>> unichr(0x3046)
u'\u3046'
>>> unichr(0xe0047)
u'\U000e0047'

```

20.45. `unicode()`: Convert to a Unicode string

Use this function to convert a string to type `unicode`. For more information about Unicode, see Section 10, “Type unicode: Strings of 32-bit characters” (p. 32).

```

>>> unicode('Pratt')
u'Pratt'
>>> unicode()
u''

```

20.46. `xrange()`: Arithmetic progression generator

The `xrange()` function has exactly the same arguments as the `range()` function (see Section 20.33, “`range()`: Generate an arithmetic progression as a list” (p. 73)).

The difference is that `xrange()` is a generator (see Section 24.3, “Generators: Functions that can produce a sequence of values” (p. 112)), while `range()` actually builds a list for its result. This means you can use `xrange()` in situations where you want to generate a large series of the values from an arithmetic progression, but you don't have enough memory to build that series as a list.

```
>>> for i in xrange(2000000000):
...     print i,
...     if i > 8:
...         break
...
0 1 2 3 4 5 6 7 8 9
>>> for i in range(2000000000):
...     print i,
...     if i > 8:
...         break
...
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
MemoryError
```

20.47. `zip()`: Combine multiple sequences

The purpose of this function is to build a list of tuples from two or more iterables of the same length. Here is the general form:

```
zip(S0, S1, S2, ...)
```

Each S_i must be in iterable. The result is a list $[T_0, T_1, \dots]$, where each T_i is the tuple $(S_0[i], S_1[i], S_2[i], \dots)$.

Here are some examples.

```
>>> L1=[1,2,3,4]
>>> L2=['a', 'b', 'c', 'd']
>>> zip(L1, L2)
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd')]
>>> L3=[10.0, 20.0, 30.0, 40.0]
>>> zip(L1, L2, L3)
[(1, 'a', 10.0), (2, 'b', 20.0), (3, 'c', 30.0), (4, 'd', 40.0)]
```

21. Advanced functions

This section describes Python functions that most people will never need. However, the advanced Python programmer may find some of them quite useful.

21.1. `basestring`: The string base class

The class `basestring` is the parent class for the two string types, `str` and `unicode`. It exists not because you can call it (you can't), but for type testing. To test whether some value s is an instance of either type of string, use this predicate:

```
isinstance(s, basestring)
```

See Section 21.12, “`isinstance()`: Is a value an instance of some class or type?” (p. 84).

21.2. `callable()`: Is this thing callable?

This predicate tests whether some value x can be called as a function.

```
callable(x)
```

Class names can be called to create an instance of the class. Instances can be called if they define a `__call__()` special method; see Section 26.3.5, “`__call__()`: What to do when someone calls an instance” (p. 131).

```
>>> def someFunction():
...     pass
...
>>> callable(someFunction)
True
>>> callable(len)
True
>>> callable(int)
True
>>> callable(42)
False
```

21.3. `classmethod()`: Create a class method

The purpose of the `classmethod()` function is to convert a method into a *class method*. For a discussion of the purpose and usage of class methods, see Section 26.5, “Class methods” (p. 136).

There are two ways to declare a class method within a class:

- You can use the function decorator syntax to declare that `classmethod` is a decorator for your method. Precede the method definition with a line reading:

```
@classmethod
def methodName(cls, ...):
    method body
```

- In some older versions of Python without the decorator syntax, you can still declare a class method by placing a line after the method definition, at the same indentation level as the method's `def` statement, having this form:

```
methodName = classmethod(methodName)
```

21.4. `delattr()`: Delete a named attribute

Use this function to delete an attribute named A of some instance I . It does not return a value. Here is the general form:

```
delattr(I, A)
```

For example, if an instance `seabiscuit` has a `rider` attribute, this statement would delete that attribute:

```
delattr(seabiscuit, 'rider')
```

If the instance has no such attribute, this function will raise an `AttributeError` exception.

21.5. `dir()`: Display a namespace's names

The purpose of the `dir()` function is to find out what names are defined in a given namespace, and return a list of those names. If called without arguments, it returns a list of the names defined in the local namespace. This function can be very handy for finding out what items are in a module or class.

Certain special names are found in most or all namespaces:

- `__doc__` is present in every namespace. Initially `None`, you can store documentation there.
- `__name__` is the name of the current module (minus the `.py` extension). In the top-level script or in conversational mode, this name is set to the string `'__main__'`.
- `__builtins__` is a list of the names of all built-in functions and variables.

In this example sequence, we'll show you what is in the global namespace just after starting up Python. Then we'll import the `math` module and display its names.

```
>>> dir()
['_builtins__', '__doc__', '__name__']
>>> x=5; forkTail='Tyrannus'
>>> dir()
['_builtins__', '__doc__', '__name__', 'forkTail', 'x']
>>> print __doc__
None
>>> print __name__
__main__
>>> import math
>>> print math.__name__
math
>>> dir(math)
['_doc__', '__file__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil',
 'cos', 'cosh', 'degrees', 'e', 'exp', 'fabs', 'floor', 'fmod', 'frexp',
 'hypot', 'ldexp', 'log', 'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>> print math.__doc__
This module is always available. It provides access to the
mathematical functions defined by the C standard.
>>> print math.log10.__doc__
log10(x) -> the base 10 logarithm of x.
>>> print __builtins__
<module '__builtin__' (built-in)>
>>> for k, name in enumerate(dir(__builtins__)):
...     if (k%4)==3: print
...     print name,
...
ArithmeticError AssertionError AttributeError
BaseException BufferError BytesWarning DeprecationWarning
EOFError Ellipsis EnvironmentError Exception
False FloatingPointError FutureWarning GeneratorExit
IOError ImportError ImportWarning IndentationError
```

```

IndexError KeyError KeyboardInterrupt LookupError
MemoryError NameError None NotImplemented
NotImplementedError OSError OverflowError PendingDeprecationWarning
ReferenceError RuntimeError RuntimeError StandardError
StopIteration SyntaxError SyntaxWarning SystemError
SystemExit TabError True TypeError
UnboundLocalError UnicodeDecodeError UnicodeEncodeError UnicodeError
UnicodeTranslateError UnicodeWarning UserWarning ValueError
Warning ZeroDivisionError __debug__
__doc__ __import__ __name__ __package__
abs all any apply
basestring bin bool buffer
bytearray bytes callable chr
classmethod cmp coerce compile
complex copyright credits delattr
dict dir divmod enumerate
eval execfile exit file
filter float format frozenset
getattr globals hasattr hash
help hex id input
int intern isinstance isinstance
iter len license list
locals long map max
min next object oct
open ord pow print
property quit range raw_input
reduce reload repr reversed
round set setattr slice
sorted staticmethod str sum
super tuple type unichr
unicode vars xrange zip

```

21.6. eval(): Evaluate an expression in source form

This function evaluates a Python expression from a string. Example:

```

>>> cent=100
>>> eval('cent**3')
1000000

```

If you want to evaluate the expression using different name environments, refer to the official documentation²¹. For related features, see also Section 21.7, “execfile(): Execute a Python source file” (p. 82) and Section 22.4, “The exec statement: Execute Python source code” (p. 95).

21.7. execfile(): Execute a Python source file

To execute a sequence of Python statements in some file *F*, use this function:

```
execfile(F)
```

²¹ <http://docs.python.org/library/functions.html>

The function returns `None`. For additional features that allow you to control the environment of the executed statements, see the official documentation²².

21.8. `getattr()`: Retrieve an attribute of a given name

Use this function to retrieve an attribute of an instance *I*, where the attribute's name is a string *S*.

```
getattr(I, s[, default])
```

If *I* has no attribute whose name matches *S*:

- If you supplied the optional *default* value, that value is returned.
- If you don't supply a *default* value and there is no such attribute in *I*, this function will raise an `AttributeError` exception.

Example:

```
>>> class C:
...     def __init__(self, flavor):
...         self.flavor = flavor
...
>>> c=C('garlicky')
>>> getattr(c, 'flavor')
'garlicky'
>>> getattr(c, 'aroma', 'bland')
'bland'
>>> getattr(c, 'aroma')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: C instance has no attribute 'aroma'
```

21.9. `globals()`: Dictionary of global name bindings

This function takes no arguments and returns a dictionary that represents the current global namespace. The keys of this dictionary are globally defined names, and each corresponding value is the value for that name. This example starts from a fresh execution of Python in conversational mode, so the global namespace has only the three special names discussed in Section 21.5, “`dir()`: Display a namespace's names” (p. 81).

```
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__'}
>>> finch = 'Fleep'
>>> globals()
{'__builtins__': <module '__builtin__' (built-in)>, '__name__': '__main__', 'finch': 'Fleep'}
```

The special name `__builtins__` is bound to a module; name `__name__` is bound to the string `'__main__'`; and `__doc__` is bound to `None`. Note that defining a new name adds an entry to the result of `globals()`.

²² <http://docs.python.org/library/functions.html>

21.10. `hasattr()`: Does a value have an attribute of a given name?

This predicate tests to see if some instance *I* has an attribute whose name is given by some string *S*:

```
hasattr(I, s)
```

If this function returns `True`, you can be sure that the instance has an attribute named *S*. However, if it returns `False`, attempts to access an attribute may still succeed, if the class provides dynamic attributes; see Section 26.3.14, “`__getattr__()`: Handle a reference to an unknown attribute” (p. 134). Example:

```
>>> class C:
...     def __init__(self, disc):
...         self.disk = disc
...
>>> c=C('five')
>>> hasattr(c, 'disk')
True
>>> hasattr(c, 'disc')
False
>>> hasattr(c, 'jukebox')
False
>>> c.jukebox = 'Nine'
>>> hasattr(c, 'jukebox')
True
```

21.11. `id()`: Unique identifier

This function, given any Python value, returns an integer value that uniquely identifies it. In most implementations, it is the value's physical memory address.

```
>>> i = 20
>>> id(i)
137727456
```

21.12. `isinstance()`: Is a value an instance of some class or type?

Use this predicate to test whether some instance *I* is an instance of some class *C* (or an instance of any ancestor class from which *C* inherits). The general form:

```
isinstance(I, C)
```

The second argument may be a class name, a type object, or a tuple of class names and type objects. If a tuple, the function will test the instance against each of the class names or type objects.

```
>>> class C1:
...     pass
...
>>> class C2(C1):
...     pass
...
>>> c2=C2()
>>> isinstance(c2,C2)
True
```

```

>>> isinstance(c2,C1)
True
>>> isinstance(c2,int)
False
>>> isinstance(1,type(55))
True
>>> isinstance(1, (int, float, long))
True
>>> isinstance('Ni', (int, float, long))
False

```

A most useful built-in Python class is `basestring`, which is the ancestor class of both `str` and `unicode` types. It is intended for cases where you want to test whether something is a string but you don't care whether it is `str` or `unicode`.

```

>>> isinstance(42, str)
False
>>> isinstance('x', str)
True
>>> isinstance(u'x', str)
False
>>> isinstance('x', basestring)
True
>>> isinstance(u'x', basestring)
True

```

21.13. `issubclass()`: Is a class a subclass of some other class?

To test whether some class `C1` is a subclass of another class `C2`, use this predicate:

```
issubclass(C1, C2)
```

Examples:

```

>>> class Polygon:
...     pass
...
>>> class Square(Polygon):
...     pass
...
>>> issubclass(Square, Polygon)
True
>>> issubclass(Polygon, Square)
False
>>> issubclass(Square, Square)
True
>>> issubclass(unicode, basestring)
True
>>> issubclass(str, basestring)
True

```

For more information about the built-in `basestring` class, see Section 21.12, “`isinstance()`: Is a value an instance of some class or type?” (p. 84).

21.14. `locals()`: Dictionary of local name bindings

This function returns a dictionary containing the names and values of all variables in the local namespace. An example:

```
>>> def f(a, b=1):
...     c=2
...     print locals()
...
>>> f(5)
{'a': 5, 'c': 2, 'b': 1}
```

For related functions, see Section 21.5, “`dir()`: Display a namespace's names” (p. 81) and Section 21.9, “`globals()`: Dictionary of global name bindings” (p. 83).

21.15. `property()`: Create an access-controlled attribute

The purpose of this function is to create a *property* of a class. A property looks and acts like an ordinary attribute, except that you provide methods that control access to the attribute.

There are three kinds of attribute access: read, write, and delete. When you create a property, you can provide any or all of three methods that handle requests to read, write, or delete that attribute.

Here is the general method for adding a property named *p* to a new-style class *C*.

```
class C(...):
    def R(self):
        ...read method...
    def W(self, value):
        ...write method...
    def D(self):
        ...delete method...
    p = property(R, W, D, doc)
    ...
```

where:

- *R* is a *getter method* that takes no arguments and returns the effective attribute value. If omitted, any attempt to read that attribute will raise `AttributeError`.
- *W* is a *setter method* that takes one argument and sets the attribute to that argument's value. If omitted, any attempt to write that attribute will raise `AttributeError`.
- *D* is a *deleter method* that deletes the attribute. If omitted, any attempt to delete that attribute will raise `AttributeError`.
- *doc* is a documentation string that describes the attribute. If omitted, defaults to the documentation string of the *R* method if any, otherwise `None`.

To retrieve a property's documentation, use this form:

```
C.p.__doc__
```

where *C* is the class name and *p* is the property name.

As an example, here is a small class that defines a property named *x*:

```

class C(object):
    def __init__(self):
        self.__x=None
    def getx(self):
        print "+++ getx()"
        return self.__x
    def setx(self, v):
        print "+++ setx({0})".format(v)
        self.__x = v
    def delx(self):
        print "+++ delx()"
        del self.__x
    x=property(getx, setx, delx, "Me property 'x'.")

```

Assuming that class is defined, here is a conversational example.

```

>>> c=C()
>>> print c.x
+++ getx()
None
>>> print C.x.__doc__
Me property 'x'.
>>> c.x=15
+++ setx(15)
>>> c.x
+++ getx()
15
>>> del c.x
+++ delx()
>>> c.x
+++ getx()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in getx
AttributeError: 'C' object has no attribute '_C__x'

```

Starting with Python 2.6, this function can also be used as a decorator (see Section 24.4, “Decorators” (p. 113)). The decorated method is used as the getter method. Furthermore, the decorated method will itself have two decorators named `setter` and `deleter`; you can use these decorators to define setter and deleter methods.

For example, suppose you want to provide your class with a property named `state`, and youra getter method returns a private attribute named `._state`. You could define it like this:

```

@property
def state(self):
    '''The internal state property.'''
    return self._state

```

In this example, not only will the `.state()` method be the getter for this property, but the documentation string `'''The internal state property.'''` will be stored as the documentation string for the property.

Suppose further that you want to write a setter method that checks to make sure the argument is a positive number less than or equal to 2. To use the built-in `setter` method to write your setter, give

the function the same name as the property, and decorate it with `P.setter` where `P` is the name of the previously defined getter:

```
@state.setter
def state(self, k):
    if not (0 <= k <= 2):
        raise ValueError("Must be 0 through 2 inclusive!")
    else:
        self._state = k
```

Similarly, you can write a deleter method by decorating it with `P.deleter`:

```
@state.deleter
def state(self):
    del self._state
```

21.16. `reload()`: Reload a module

This function reloads a previously loaded module (assuming you loaded it with the syntax `import moduleName`). It is intended for conversational use, where you have edited the source file for a module and want to test it without leaving Python and starting it again. General form:

```
reload(moduleName)
```

The `moduleName` is the actual name of the module, not a string containing its name. For example, if you have imported a module like this:

```
import parrot
```

you would say `reload(parrot)`, not `reload('parrot')`.

21.17. `repr()`: Representation

The `repr` function attempts to convert any value to a string. Unlike the `str()` function, it attempts to display the value in Python source form, that is, in a form suitable for passing to `eval()` (see Section 21.6, `eval(): Evaluate an expression in source form` (p. 82)). It works the same as the ``...`` operator. Examples:

```
>>> s='Wensleydale'
>>> print s
Wensleydale
>>> print str(s)
Wensleydale
>>> print repr(s)
'Wensleydale'
>>> print `s`
'Wensleydale'
```

To specify the behavior of the `repr()` when it is applied to an instance of a user-defined class, see Section 26.3.19, `__repr__(): String representation` (p. 135).

21.18. `setattr()`: Set an attribute

This function is the inverse of Section 21.8, “`getattr()`: Retrieve an attribute of a given name” (p. 83): it sets the value of some attribute whose name is *A* from an instance *I* to a new value *V*:

```
setattr(I, A, V)
```

Example:

```
>>> class C5:
...     def __init__(self, x):
...         self.x = x
...
>>> c=C5(14)
>>> c.x
14
>>> setattr(c, 'x', 19)
>>> c.x
19
>>> setattr(c, 'violateEncapsulation', True)
>>> c.violateEncapsulation
True
```

As the last lines above show, you can use this function to create attributes that didn't even exist before. However, this is often considered bad style, as it violates the principle of encapsulation²³.

21.19. `slice()`: Create a slice instance

This function creates an instance of type `slice`. A slice instance *I* can be used to index a sequence *S* in an expression of the form *S*[*I*]. Here is the general form:

```
slice(start, limit, step)
```

The result is a slice that is equivalent to `start:limit:step`. Use `None` to get the default value for any of the three arguments.

Examples:

```
>>> r = range(9)
>>> r
[0, 1, 2, 3, 4, 5, 6, 7, 8]
>>> r[::2]
[0, 2, 4, 6, 8]
>>> r[slice(None, None, 2)]
[0, 2, 4, 6, 8]
>>> r[3:7]
[3, 4, 5, 6]
>>> r[slice(3,7)]
[3, 4, 5, 6]
>>> r[1::2]
[1, 3, 5, 7]
>>> odds = slice(1, None, 2)
>>> r[odds]
```

²³ [http://en.wikipedia.org/wiki/Encapsulation_\(object-oriented_programming\)](http://en.wikipedia.org/wiki/Encapsulation_(object-oriented_programming))

```
[1, 3, 5, 7]
>>> 'roygbiv'[odds]
'ogi'
```

21.20. `staticmethod()`: Create a static method

The purpose of the `staticmethod` function is to convert a method into a *static method*. See Section 26.4, “Static methods” (p. 136) for definitions and usage.

Typically you will declare a static method using the decorator syntax, like this:

```
@staticmethod
def methodName(...):
    method body
```

An alternative is to place a line like this *after* the method's definition (at the same indentation level as its `def`):

```
methodName = staticmethod(methodName)
```

21.21. `super()`: Superclass

The purpose of this function is to retrieve the superclass of a given type or object. Superclasses are beyond the scope of this document; see the online documentation for built-in functions²⁴ (scroll down).

21.22. `vars()`: Local variables

This function returns a dictionary that represents a symbol table: its keys are variable names, and each key's corresponding value is its bound value. The official documentation warns you not to change anything in this dictionary, or Bad Things Will Happen.

There are two forms of call:

`vars()`

Returns the local symbol table.

`vars(ns)`

Returns the symbol table of a namespace *ns*, where a namespace can be a module, an instance, or a class.

Compare the similar functions Section 21.5, “`dir()`: Display a namespace's names” (p. 81), Section 21.14, “`locals()`: Dictionary of local name bindings” (p. 86), and Section 21.9, “`globals()`: Dictionary of global name bindings” (p. 83).

22. Simple statements

Python statement types are divided into two groups. Simple statements, that are executed sequentially and do not affect the flow of control, are described first. Compound statements, which may affect the sequence of execution, are discussed in Section 23, “Compound statements” (p. 99).

²⁴ <http://docs.python.org/library/functions.html>

Here, for your convenience, is a table of all the Python statement types, and the sections where they are described. The first one, the assignment statement, does not have an initial keyword: an assignment statement is a statement of the form “*variable = expression*”.

Assignment	Section 22.1, “The assignment statement: <i>name = expression</i> ” (p. 91).
<code>assert</code>	Section 22.2, “The <code>assert</code> statement: Verify preconditions” (p. 94).
<code>break</code>	Section 23.2, “The <code>break</code> statement: Exit a <code>for</code> or <code>while</code> loop” (p. 100).
<code>continue</code>	Section 23.3, “The <code>continue</code> statement: Jump to the next cycle of a <code>for</code> or <code>while</code> ” (p. 101).
<code>del</code>	Section 22.3, “The <code>del</code> statement: Delete a name or part of a value” (p. 95).
<code>elif</code>	Section 23.5, “The <code>if</code> statement: Conditional execution” (p. 102) and Section 23.8, “The <code>try</code> statement: Anticipate exceptions” (p. 105).
<code>else</code>	Section 23.5, “The <code>if</code> statement: Conditional execution” (p. 102).
<code>except</code>	Section 23.8, “The <code>try</code> statement: Anticipate exceptions” (p. 105).
<code>exec</code>	Section 22.4, “The <code>exec</code> statement: Execute Python source code” (p. 95).
<code>finally</code>	Section 23.8, “The <code>try</code> statement: Anticipate exceptions” (p. 105).
<code>for</code>	Section 23.4, “The <code>for</code> statement: Iteration over a sequence” (p. 101).
<code>from</code>	Section 22.6, “The <code>import</code> statement: Use a module” (p. 97).
<code>global</code>	Section 22.5, “The <code>global</code> statement: Declare access to a global name” (p. 95).
<code>if</code>	Section 23.5, “The <code>if</code> statement: Conditional execution” (p. 102).
<code>import</code>	Section 22.6, “The <code>import</code> statement: Use a module” (p. 97).
<code>pass</code>	Section 22.7, “The <code>pass</code> statement: Do nothing” (p. 98).
<code>print</code>	Section 22.8, “The <code>print</code> statement: Display output values” (p. 98).
<code>raise</code>	Section 23.6, “The <code>raise</code> statement: Cause an exception” (p. 103).
<code>return</code>	Section 23.7, “The <code>return</code> statement: Exit a function or method” (p. 105).
<code>try</code>	Section 23.8, “The <code>try</code> statement: Anticipate exceptions” (p. 105).
<code>yield</code>	Section 23.10, “The <code>yield</code> statement: Generate one result from a generator” (p. 108).

22.1. The assignment statement: *name = expression*

The purpose of Python's assignment statement is to associate names with values in your program. It is the only statement that does not start with a keyword. An assignment statement is a line containing at least one single equal sign (=) that is not inside parentheses.

Here is the general form of an assignment statement:

```
target0 = target1 = ... = expression
```

In most cases, there will be one *target* that is a name. Python will evaluate the *expression*, reducing it to a single value, and then *bind* that name to the that value.

A *binding* is an association between a name and a value. It is important to note that in Python, unlike many other languages, names themselves are not associated with a specific type. A name is just a label, and it can be bound to any value of any type at any time. In this example, name `x` is bound first to an `int` value 5, then to a `str` value 'Some string'.

```

>>> x = 5
>>> x
5
>>> x = 'Some string'
>>> print x
Some string

```

If a target name was already bound to a value, the name is unbound from that value before it is rebound to the new value. For each value in a running program, Python keeps track of how many names are bound to that value. When the value has no more names bound to it, the value's memory is automatically recycled. If the value is an instance of a class, its destructor may be called; see Section 26.3.8, “`__del__()`: Destructor” (p. 132).

There are several other forms of assignment statement.

$n_0 = n_1 = \dots = \text{expression}$

If you supply multiple target names, each target will be assigned the value of the *expression*. Example:

```

>>> i = j = errorCount = 0
>>> i
0
>>> j
0
>>> errorCount
0

```

$n_0, n_1, \dots = \text{expression}$

If the target is a comma-separated list of names, the *expression* must evaluate to an iterable with the same number of values as there are names. Each value in the *expression* is then bound to the corresponding name. Example:

```

>>> L = ["Bell's Blue", "male", 6]
>>> name, sex, age = L
>>> name
"Bell's Blue"
>>> sex
'male'
>>> age
6

```

This feature, called “unpacking,” generalizes to arbitrarily nested sequences within sequences. You may group targets inside parentheses (`(...)`) or brackets [`[...]`] to show the levels of nesting. Here is an example:

```

>>> s = [1, [2, 3, [4, 5], 6], 7]
>>> a, (b, c, [d, e], f), g = s
>>> print a,b,c,d,e,f,g
1 2 3 4 5 6 7

```

All the assignments are effectively simultaneous. Therefore, you can safely exchange the values of two variables using a statement like this:

```

v1, v2 = v2, v1

```

Examples:

```

>>> a=5; b=9998
>>> print a,b
5 9998
>>> a,b=b,a
>>> print a,b
9998 5
>>> c=432
>>> a,b,c = b,c,a
>>> print a,b,c
5 432 9998

```

`name[i] = expression`

If *name* is an iterable, the expression *i* must evaluate to an integer. The element after position *i* is replaced by the value of the *expression*.

```

>>> L = range(6)
>>> L
[0, 1, 2, 3, 4, 5]
>>> L[2]
2
>>> L[2] = 888
>>> L
[0, 1, 888, 3, 4, 5]

```

If *name* is a dictionary (or other mapping), and *name* does not have a key-value pair whose key equals *index*, a new key-value pair is added to *name* with key *i* and value *expression*.

```

>>> d={'pudding': 'figgy'}
>>> d
{'pudding': 'figgy'}
>>> d['tart'] = 'strawberry'
>>> d
{'pudding': 'figgy', 'tart': 'strawberry'}
>>> d["tart"] = "rat"
>>> d
{'pudding': 'figgy', 'tart': 'rat'}

```

As the last two lines show, if the dictionary already has a key-value pair for key *i*, the old value of that pair is replaced by the *expression* value.

`name[start:end] = S`

If *name* is a list or other mutable sequence, you can replace the elements of a slice of that sequence with the elements from some sequence *S*. (For an explanation of slicing, see Section 8.1, "Operations common to all the sequence types" (p. 12).) This may result in addition, deletion, or replacement of the elements of *name*. Some examples will give the flavor of this kind of assignment.

```

>>> L=range(6)
>>> L
[0, 1, 2, 3, 4, 5]
>>> L[2:4]
[2, 3]
>>> L[2:4] = [111, 222, 333, 444, 555]
>>> L
[0, 1, 111, 222, 333, 444, 555, 4, 5]
>>> L[3]

```

```

222
>>> L[3:3]
[]
>>> L[3:3] = [41.0, 42.0, 43.0]
>>> L
[0, 1, 111, 41.0, 42.0, 43.0, 222, 333, 444, 555, 4, 5]
>>> L[4:7]
[42.0, 43.0, 222]
>>> L[4:7] = ()
>>> L
[0, 1, 111, 41.0, 333, 444, 555, 4, 5]

```

Note

The “=” signs in an assignment is *not* an operator, as it is in some other languages. You cannot assign a value to a name inside an expression; an assignment statement must stand alone.

```

>>> a = 5 + (a=7)
File "<stdin>", line 1
  a = 5 + (a=7)
              ^
SyntaxError: invalid syntax

```

Python also supports *augmented assignment*. In this form, you may place certain operators *before* the “=”. Here is the general form:

```
name operator= expression
```

An assignment of this general form has the same semantics as this form:

```
name = name operator expression
```

Supported *operator* symbols include:

```
+ - * / % ** >> << & ^ |
```

Examples:

```

>>> i = 1
>>> i += 3
>>> i
4
>>> i *= 5
>>> i
20

```

22.2. The assert statement: Verify preconditions

To check for “shouldn't happen” errors, you can use an `assert` statement:

```

assert e1
assert e1, e2

```

where e_1 is some condition that should be true. If the condition is false, Python raises an `AssertionError` exception (see Section 25, “Exceptions: Error signaling and handling” (p. 114)).

If a second expression e_2 is provided, the value of that expression is passed with the exception.

Assertion checking can be disabled by running Python with the `-O` (optimize) option.

22.3. The `del` statement: Delete a name or part of a value

The purpose of the `del` statement is to delete things. The general form is:

```
del  $L_0, L_1, \dots$ 
```

where each L_i is an item to be deleted. You can delete:

- A name. For example, the statement

```
del i, j
```

causes names `i` and `j` to become *unbound*, that is, undefined.

- An element or slice from a list. For example:

```
del L[5], M[-2:]
```

would delete the sixth element of list `L` and the last two elements of list `M`.

- One entry in a dictionary. For example, if `D` is a dictionary,

```
del D['color']
```

would delete from `D` the entry for key `'color'`.

22.4. The `exec` statement: Execute Python source code

To dynamically execute Python code, use a statement of this form:

```
exec  $E_0$  [in  $E_1$  [,  $E_2$ ]]
```

Expression E_0 specifies what to execute, and may be a string containing Python source code, an open file, or a code object. If E_1 is omitted, the code is executed in the local scope. If E_1 is given but E_2 is not, E_1 is a dictionary used to define the names in the global and local scopes. If E_2 is given, E_1 is a dictionary defining the global scope, and E_2 is a dictionary defining the local scope.

22.5. The `global` statement: Declare access to a global name

The purpose of the `global` statement is to declare that a function or method intends to change the value of a name from the global scope, that is, a name from outside the function.

When Python reads the definition of a function, it checks each name to see if that name's value may possibly be changed anywhere in the function—that is, if the name shows up on the left side of an assignment statement, or as the induction variable in a `for` loop, or in any other context where the name's value can be changed.

Such names are assumed to be local to the function unless you override this behavior by declaring that name in a `global` statement. Here is the general form:

```
global name1, name2, ...
```

Some conversational examples may help make this clear. Suppose you define a global variable `x`; you can use that name inside a function.

```
>>> x = 5
>>> def show1():
...     print x
...
>>> show1()
5
```

However, if you assign a value to `x` inside the function, the name `x` is now local to the function. It is said to *shadow* the global variable with the same name, and any changes to the value associated with that name inside the function will operate on a local copy, and will not affect the value of the global variable `x`.

```
>>> x = 5
>>> def show2():
...     x = 42
...     print x
...
>>> show2()
42
>>> x
5
```

But if you actually do want to change the value of the global variable inside the function, just declare it global like this:

```
>>> x = 5
>>> def show3():
...     global x
...     x = 42
...     print x
...
>>> show3()
42
>>> x
42
```

Notice what happens in this case:

```
>>> x = 5
>>> def show4():
...     print x, "Before"
...     x = 42
...     print x, "After"
...
>>> show4()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in show4
UnboundLocalError: local variable 'x' referenced before assignment
```

Because the line “`x = 42`” changes the value of `x`, and because it is not declared as a global, execution fails because the value of the local variable `x` is used before that variable has had a value assigned to it.

22.6. The `import` statement: Use a module

One of the cornerstones of Python's design philosophy is to keep the language relatively small and well-defined, and move all non-essential functionality to library modules. The `import` and `from` statements allow your programs to use items from these library modules.

- Your Python installation will come with a large collection of released modules.
- You can also create your own modules. Just place Python statements defining variables, functions, and classes into a file whose name ends in “.py”.

There are two different statements you can use to import items from a module:

- The `from` statement copies items from a module into your namespace. After importing an item in this way, you can refer to the item simply by its name.

General forms:

```
from moduleName import *
from moduleName import name1, name2, ...
```

The first form imports all the items from the module named *moduleName*. If you want to import only specific items, use the second form, and enumerate the names you want from that module.

- The `import` statement makes an entire module's content available to you as a separate namespace. To refer to some item named *N* in a module named *M*, use the dot notation, *M.N*.

Here is the general form:

```
import moduleName, ...
```

If you want to use some module *M* in this way, but you want to change the name to some different name *A*, use this form:

```
import M as A
```

Here are some examples that use the standard `math` module that is always available in a proper Python install. This module has functions such as `sqrt()` (square root), as well as variables such as `pi`. (Although π is a constant in the mathematical sense, the name `pi` is a variable in the Python sense.)

```
>>> from math import *
>>> sqrt(16)
4.0
>>> pi
3.1415926535897931
```

If you wanted only the `sqrt` function and the variable `pi`, this statement would do the job:

```
from math import sqrt, pi
```

Now some examples of the second form.

```
>>> import math
>>> math.sqrt(25)
5.0
```

```
>>> math.pi
3.1415926535897931
```

Suppose your program already used the name `math` for something else, but you still want to use functions from the `math` module. You can import it under a different name like this:

```
>>> import math as crunch
>>> crunch.sqrt(25)
5.0
>>> crunch.pi
3.1415926535897931
```

22.7. The `pass` statement: Do nothing

Python's `pass` statement is a placeholder. It does nothing.

Here's an example. Suppose you have a function named `arr()` that does something and then returns a `True` or `False` value. You want to keep calling this function until it returns a false value. This code would suffice:

```
while arr():
    pass
```

22.8. The `print` statement: Display output values

Use this statement to display values on the standard output stream. General form:

```
print thing1, thing2, ...[,]
```

Each *thing* must be a string, or a value that can be converted into a string by the `str()` function (see Section 20.40, “`str()`: Convert to `str` type” (p. 77)). These strings are written to the standard output stream, with one space between each value. A `print` statement by itself prints an empty line.

```
>>> print 4.3, 'Sir Robin', 1./7
4.3 Sir Robin 0.142857142857
>>> for i in range(4):
...     print i**4,
...
0 1 16 81
>>>
```

Normally, a newline is printed after the last value. However, you can suppress this behavior by appending a comma to the end of the list. For example, this statement:

```
print 'State your name:',
```

would print the string followed by one space and leave the cursor at the end of that line.

22.9. The `print()` function

In Python 3, `print` is a function, not a statement. To make it easier to convert your programs to the new syntax, first use this `import` statement (introduced in Python 2.6):

```
from __future__ import print_function
```

Here is the interface to this function:

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

args

One or more positional arguments whose values are to be printed.

sep

By default, consecutive values are separated by one space. You may specify a different separator string using this keyword argument.

end

By default, a newline ("`\n`") is written after the last value in `args`. You may use this keyword argument to specify a different line terminator, or no terminator at all.

file

Output normally goes to the standard output stream (`sys.stdout`). To divert the output to another writeable file, use this keyword argument.

Here's an example. Suppose you are writing three strings named `clan`, `moiety`, and `distro` to a writeable file named `spreader`, and you want to separate the fields with tab ("`\t`") characters, and use ASCII²⁵ CR, Carriage Return ("`\r`"), as the line terminator. Your call to the `print()` function would go something like this:

```
print(clan, moiety, distro, file=spreader, end='\r', sep='\t')
```

23. Compound statements

The statements in this section alter the normal sequential execution of a program. They can cause a statement to be executed only under certain circumstances, or execute it repeatedly.

23.1. Python's block structure

One unusual feature of Python is the way that the indentation of your source program organizes it into blocks within blocks within blocks. This is contrary to the way languages like C and Perl organize code blocks by enclosing them in delimiters such as braces `{ ... }`.

Various Python branching statements like `if` and `for` control the execution of blocks of lines.

- At the very top level of your program, all statements must be unindented—they must start in column one.
- Various Python branching statements like `if` and `for` control the execution of one or more subsidiary blocks of lines.
- A block is defined as a group of adjacent lines that are indented the same amount, but indented further than the controlling line. The amount of indentation of a block is not critical.
- You can use either spaces or *tab* characters for indentation. However, mixing the two is perverse and can make your program hard to maintain. Tab stops are assumed to be every eight columns.

Blocks within blocks are simply indented further. Here is an example of some nested blocks:

²⁵ <http://en.wikipedia.org/wiki/ASCII>

```

if i < 0:
    print "i is negative"
else:
    print "i is nonnegative"
    if i < 10:
        print "i has one digit"
    else:
        print "i has multiple digits"

```

If you prefer a more horizontal style, you can always place statements after the colon (:) of a compound statement, and you can place multiple statements on a line by separating them with semicolons (;). Example:

```

>>> if 2 > 1: print "Math still works"; print "Yay!"
... else: print "Huh?"
...
Math still works
Yay!

```

You can't mix the block style with the horizontal style: the consequence of an `if` or `else` must either be on the same line or in a block, never both.

```

>>> if 1: print "True"
...     print "Unexpected indent error here."
      File "<stdin>", line 2
        print "Unexpected indent error here."
        ^
IndentationError: unexpected indent
>>>

```

23.2. The `break` statement: Exit a `for` or `while` loop

The purpose of this statement is to jump out of a `for` or `while` loop before the loop would terminate otherwise. Control is transferred to the statement after the last line of the loop. The statement looks like this:

```
break
```

Here's an example.

```

>>> for i in [1, 71, 13, 2, 81, 15]:
...     print i,
...     if (i%2) == 0:
...         break
...
1 71 13 2

```

Normally this loop would be executed six times, once for each value in the list, but the `break` statement gets executed when `i` is set to an even value.

23.3. The continue statement: Jump to the next cycle of a for or while

Use a `continue` statement inside a `for` or `while` loop when you want to jump directly back to the top of the loop and go around again.

- If used inside a `while` loop, the loop's condition expression is evaluated again. If the condition is `False`, the loop is terminated; if the condition is `True`, the loop is executed again.
- Inside a `for` loop, a `continue` statement goes back to the top of the loop. If there are any values remaining in the iterable that controls the loop, the loop variable is set to the next value in the iterable, and the loop body is entered.

If the `continue` is executed during the last pass through the loop, control goes to the statement after the end of the loop.

Examples:

```
>>> i = 0
>>> while i < 10:
...     print i,
...     i += 1
...     if (i%3) != 0:
...         continue
...     print "num",
...
0 1 2 num 3 4 5 num 6 7 8 num 9
>>> for i in range(10):
...     print i,
...     if (i%4) != 0:
...         continue
...     print "whee",
...
0 whee 1 2 3 4 whee 5 6 7 8 whee 9
```

23.4. The for statement: Iteration over a sequence

Use a `for` statement to execute a block of statements repeatedly. Here is the general form. (For the definition of a block, see Section 23.1, "Python's block structure" (p. 99).)

```
for V in S:
    B
```

- *V* is a variable called the *induction variable*.
- *S* is any iterable; see Section 19.2, "What is an iterable?" (p. 59).

This iterable is called the *controlling iterable* of the loop.

- *B* is a block of statements.

The block is executed once for each value in *S*. During each execution of the block, *V* is set to the corresponding value of *S* in turn. Example:

```
>>> for color in ['black', 'blue', 'transparent']:
...     print color
...
black
```

```
blue
transparent
```

In general, you can use any number of induction variables. In this case, the members of the controlling iterable must themselves be iterables, which are unpacked into the induction variables in the same way as sequence unpacking as described in Section 22.1, “The assignment statement: *name = expression*” (p. 91). Here is an example.

```
>>> fourDays = ( ('First', 1, 'orangutan librarian'),
...              ('Second', 5, 'loaves of dwarf bread'),
...              ('Third', 3, 'dried frog pills'),
...              ('Fourth', 2, 'sentient luggages') )
>>> for day, number, item in fourDays:
...     print ( "On the {1} day of Hogswatch, my true love gave "
...             "to me".format(day) )
...     print "{0} {1}".format(number, item)
...
On the First day of Hogswatch, my true love gave to me
1 orangutan librarian
On the Second day of Hogswatch, my true love gave to me
5 loaves of dwarf bread
On the Third day of Hogswatch, my true love gave to me
3 dried frog pills
On the Fourth day of Hogswatch, my true love gave to me
2 sentient luggages
```

You can change the induction variable inside the loop, but during the next pass through the loop, it will be set to the next element of the controlling iterable normally. Modifying the controlling iterable itself won't change anything; Python makes a copy of it before starting the loop.

```
>>> for i in range(4):
...     print "Before:", i,
...     i += 1000
...     print "After:", i
...
Before: 0 After: 1000
Before: 1 After: 1001
Before: 2 After: 1002
Before: 3 After: 1003
>>> L = [7, 6, 1912]
>>> for n in L:
...     L = [44, 55]
...     print n
...
7
6
1912
```

23.5. The `if` statement: Conditional execution

The purpose of the `if` construct is to execute some statements only when certain conditions are true.

Here is the most general form of an `if` construct:

```

if  $E_0$ :
     $B_0$ 
elif  $E_1$ :
     $B_1$ 
elif ...:
    ...
else:
     $B_f$ 

```

In words, this construct means:

- If expression E_0 is true, execute block B_0 .
- If expression E_0 is false but E_1 is true, execute block B_1 .
- If there are more `elif` clauses, evaluate each one's expression until that expression has a true value, and then execute the corresponding block.
- If all the expressions in `if` and `elif` clauses are false, execute block B_f .

An `if` construct may have zero or more `elif` clauses. The `else` clause is also optional.

Examples:

```

>>> for i in range(5):
...     if i == 0:
...         print "i is zero",
...     elif i == 1:
...         print "it's one",
...     elif i == 2:
...         print "it's two",
...     else:
...         print "many",
...     print i
...
i is zero 0
it's one 1
it's two 2
many 3
many 4
>>> if 2 > 3: print "Huh?"
...
>>> if 2 > 3: print "Huh?"
... else: print "Oh, good."
...
Oh, good.
>>> if 2 > 3: print "Huh?"
... elif 2 == 2: print "Oh."
...
Oh.

```

23.6. The `raise` statement: Cause an exception

Python's exception mechanism is the universal framework for dealing with errors—situations where your program can't really proceed normally. For an overview, see Section 25, “Exceptions: Error signaling and handling” (p. 114).

There are three forms of the `raise` statement:

```
raise
raise E1
raise E1, E2
```

The first form is equivalent to “`raise None, None`” and the second form is equivalent to “`raise E1, None`”. Each form raises an exception of a given type and with a given value. The type and value depend on how many expressions you provide:

<i>E1</i>	<i>E2</i>	Exception type	Exception value
None	None	Re-raise the current exception, if any. This might be done, for example, inside an <code>except</code> , <code>else</code> , or <code>finally</code> block; see Section 23.8, “The <code>try</code> statement: Anticipate exceptions” (p. 105).	
class	None	<i>E1</i>	<i>E1</i> ()
class	instance of <i>E1</i>	<i>E1</i>	<i>E2</i>
class	tuple	<i>E1</i>	<i>E1</i> (* <i>E2</i>)
class	none of the above	<i>E1</i>	<i>E1</i> (<i>E2</i>)
instance	None	<code>type(<i>E1</i>)</code>	<i>E1</i>

The current recommended practice is to use a `raise` statement of this form:

```
raise E(...)
```

where *E* is some class derived from the built-in `Exception` class: you can use one of the built-in exceptions, or you can create your own exception classes.

For classes derived from `Exception`, the constructor takes one argument, an error message—that is, a string explaining why the exception was raised. The resulting instance makes that message available as an attribute named `.message`. Example:

```
>>> try:
...     raise ValueError('The day is too frabjous.')
... except ValueError as x:
...     pass
...
>>> type(x)
<type 'exceptions.ValueError'>
>>> x.message
'The day is too frabjous.'
```

To create your own exceptions, write a class that inherits from `Exception` and passes its argument to the parent constructor, as in this example.

```
>>> class VocationError(Exception):
...     def __init__(self, mismatch):
...         Exception.__init__(self, mismatch)
...
>>> try:
...     print "And now, the Vocational Guidance Counsellor Sketch."
...     raise VocationError("Does not have proper hat")
...     print "This print statement will not be reached."
... except VocationError as problem:
```

```
...     print "Vocation problem: {0}".format(problem)
...
And now, the Vocational Guidance Counsellor Sketch.
Vocation problem: Does not have proper hat
```

23.7. The return statement: Exit a function or method

Within any function or method, you can exit the function with a `return` statement. There are two forms:

```
return expression
return
```

In the first form, execution resumes at the point where the function or method was called, and the value of the *expression* is substituted into the calling statement.

The second form is the equivalent of “`return None`”. (See Section 18, “None: The special placeholder value” (p. 56).)

23.8. The try statement: Anticipate exceptions

The purpose of a “`try`” construct is to specify what actions to take in the event of errors. For an introduction to Python's exception mechanism, see Section 25, “Exceptions: Error signaling and handling” (p. 114).

When an exception is raised, two items are associated with it:

- An exception type, and
- an exception value.

Here is the most general form of a `try` construct. Symbols like B_0 and B_1 represent indented blocks of statements. Each `except` clause names some exception class E_i (or a tuple of exception classes), and optionally a variable v_i that will be set to the exception value.

```
try:
     $B_0$ 
except  $E_1$  [as  $v_1$ ]:
     $B_1$ 
except  $E_2$  [as  $v_2$ ]:
     $B_2$ 
except ...:
    ...
else:
     $B_e$ 
finally:
     $B_f$ 
```

The `else:` and `finally:` blocks are optional. There must be at least one `except` block, but there may be any number.

Here is a simplified description of the execution of a `try` block in general:

1. If B_0 executes without raising any exceptions, the `else` block B_e is executed, then the `finally` block B_f .

2. If the execution of block B_0 raises some exception with type E_0 , that type is compared against each `except` clause until one of them matches the raised exception or there are no more `except` clauses.

The matching condition is slightly complicated: for some clause “`except E_i as v_i :`”, expression E_i is either an exception class or a tuple of exception classes.

- If E_i is a single class, it is considered a match if E_0 is either the same class as E_i or a subclass of E_i .
- If E_i is a tuple of exception classes, the raised exception is compared to each to see if it is the same class or a subclass, as in the single-class case.

If multiple `except` clauses match, the first matching clause is said to *handle* the exception. The corresponding variable v_i (if present) is bound to the raised exception instance, and control passes to the corresponding block B_i .

3. If there is a `finally` clause, it is executed, whether the exception was caught or not. If the exception was not caught, it is re-raised after the end of the `finally` clause.

Examples:

```
>>> try:
...     raise ValueError("Strike three!")
... except IOError as x:
...     print "I/O error caught:", x
... except ValueError as x:
...     print "Value error caught:", x
... except SyntaxError as x:
...     print "Syntax error caught:", x
... else:
...     print "This is the else clause"
... finally:
...     print "This is the finally clause"
...
Value error caught: Strike three!
This is the finally clause
>>> try:
...     raise ValueError("Uncaught!")
... except IOError as x:
...     print "I/O error:", x
... else:
...     print "This is the else clause"
... finally:
...     print "This is the finally clause"
...
This is the finally clause
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
ValueError: Uncaught!
>>> try:
...     print "No exceptions raised"
... except ValueError as x:
...     print "ValueError:", x
... else:
...     print "This is the else clause"
... finally:
```

```

...     print "This is the finally clause"
...
No exceptions raised
This is the else clause
This is the finally clause

```

For those of you who are interested in the gory details, the fun begins when a second or even a third exception is raised inside an `except`, `else`, or `finally` clause. The results are well-defined, and here is a pseudocode description of the edge cases. In this procedure, we'll use two internal variables named `pending` and `detail`.

1. Set `pending` to `None`.
2. Attempt to execute block B_θ . If this block raises an exception E_θ with detail d_θ , set `pending` to E_θ and set `detail` to d_θ .
3. If `pending` is `None`, go to Step 8 (p. 107).
4. Find the first block `except E_i , v_i :` such that `issubclass(E_θ , E_i)`.
If there is no such match, go to Step 10 (p. 107).
5. Set v_i to `detail`.
6. Attempt to execute block B_i .
If this block raises some new exception E_n with detail d_n , set `pending` to E_n and set `detail` to d_n .
However, if block B_i executes without exception, set `pending` to `None`. In this case, the original exception is said to have been caught or handled.
7. Go to Step 10 (p. 107).
8. If there is no `else:` clause, go to Step 10 (p. 107).
9. Attempt to execute the `else:` block B_e .
If this block raises some new exception E_n with detail d_n , set `pending` to E_n and set `detail` to d_n .
10. If there is no `finally:` clause, proceed to Step 12 (p. 107).
11. Attempt to execute the `finally:` block E_f .
If this block raises some new exception E_n with detail d_n , set `pending` to E_n and set `detail` to d_n .
12. If `pending` is not `None`, re-raise the exception as in this statement:

```
raise pending, detail
```

If `pending` is `None`, fall through to the statement following the `try:` block.

23.9. The `with` statement and context managers

The purpose of this statement is to protect a block of code with a *context manager* that insures that certain initialization and cleanup steps get performed, regardless of whether that block raises an exception.

A context manager is a class that has `.__enter__()` and `.__exit__()` methods.

1. The `.__enter__()` method performs any necessary initialization, and returns a value.
2. The `.__exit__()` method is always executed to perform necessary cleanup actions.


```
>>> for x in updown(4):
...     print x,
...
0 1 2 3 4 3 2 1 0
```

24. def (): Defining your own functions

The `def` construct is used to define functions and methods. Here is the general form:

```
def n(p0[=e0][, p1[=e1]]...[, *pv][, **pd]):
    B
```

The name *n* of the function is followed by a pair of parentheses containing descriptions of the arguments to the function. The block *B* is called the *body* of the function, and is executed when the function is called.

A function may have no arguments at all. If there are arguments to be passed to the function when it is called, they must be declared in this order:

- A *positional argument* is a name that is not followed by an equal sign (=) and default value.
- A *keyword argument* is followed by an equal sign and an expression that gives its *default value*.
If a function has both positional arguments and keyword arguments, all positional arguments must precede all keyword arguments.
- If there is a **p*_v parameter, when the function is called that name is bound to a (possibly empty) tuple of all positional arguments passed to the function that do not correspond to other positional or keyword arguments in the `def`.
- If there is a ***p*_d parameter, when the function is called that name is bound to a dictionary of all keyword arguments passed to the function that do not appear in the function's `def`.

When you call a function, the argument values you pass to it must obey these rules:

- There are two kinds of arguments: positional (also called non-default arguments) and keyword (also called default arguments). A positional argument is simply an expression, whose value is passed to the argument.

A keyword argument has this form:

```
name=expression
```

- All positional arguments in the function call (if any) must precede all keyword arguments (if any).

```
>>> def wrong(f=1, g):
...     print f, g
...
File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```

- You must supply at least as many positional arguments as the function expects.

```
>>> def wantThree(a, b, c):
...     print a,b,c
...
>>> wantThree('nudge', 'nudge', 'nudge')
```

```
nudge nudge nudge
>>> wantThree('nudge')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: wantThree() takes exactly 3 arguments (1 given)
```

- If you supply more positional arguments than the function expects, the extra arguments are matched against keyword arguments in the order of their declaration in the `def`. Any additional keyword arguments are set to their default values.

```
>>> def f(a, b, c=1, d='elk'):
...     print a,b,c,d
...
>>> f(99, 111)
99 111 1 elk
>>> f(99, 111, 222, 333)
99 111 222 333
>>> f(8, 9, 10, 11, 12, 13)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: f() takes at most 4 arguments (6 given)
```

- You may supply arguments for keyword parameters in any order by using the form `k=v`, where `k` is the keyword used in the declaration of that parameter and `v` is your desired argument.

```
>>> def blackKeys(fish='Eric', dawn='Stafford', attila='Abdul'):
...     print fish, dawn, attila
...
>>> blackKeys()
Eric Stafford Abdul
>>> blackKeys(attila='Gamera', fish='Abdul')
Abdul Stafford Gamera
```

- If you declare a parameter of the form `*name`, the caller can provide any number of additional keyword arguments, and the `name` will be bound to a tuple containing those additional arguments.

```
>>> def posish(i, j, k, *extras):
...     print i,j,k,extras
...
>>> posish(38, 40, 42)
38 40 42 ()
>>> posish(44, 46, 48, 51, 57, 88)
44 46 48 (51, 57, 88)
```

- Similarly, you may declare a final parameter of the form `**name`. If the caller provides any keyword arguments whose names do not match declared keyword arguments, that `name` will be bound to a dictionary containing the additional keyword arguments as key-value pairs.

```
>>> def extraKeys(a, b=1, *c, **d):
...     print a, b, c, d
...
>>> extraKeys(1,2)
1 2 () {}
```

```
>>> extraKeys(3,4,6,12, hovercraft='eels', record='scratched')
3 4 (6, 12) {'record': 'scratched', 'hovercraft': 'eels'}
```

24.1. A function's local namespace

Any name that appears in a function's argument list, or any name that is set to a value anywhere in the function, is said to be *local* to the function. If a local name is the same as a name from outside the function (a so-called *global* name), references to that name inside the function will refer to the local name, and the global name will be unaffected. Here is an example:

```
>>> x = 'lobster'
>>> y = 'Thermidor'
>>> def f(x):
...     y = 'crevettes'
...     print x, y
...
>>> f('spam')
spam crevettes
>>> print x, y
lobster Thermidor
```

Keyword parameters have a special characteristic: their names are local to the function, but they are also used to match keyword arguments when the function is called.

24.2. Iterators: Values that can produce a sequence of values

Closely related to Python's concept of sequences is the concept of an *iterator*:

For a given sequence S , an iterator I is essentially a set of instructions for producing the elements of S as a sequence of zero or more values.

To produce an iterator over some sequence S , use this function:

```
iter(S)
```

- The result of this function is an "iterator object" that can be used in a `for` statement.

```
>>> continents = ('AF', 'AS', 'EU', 'AU', 'AN', 'SA', 'NA')
>>> worldWalker = iter(continents)
>>> type(worldWalker)
<type 'tupleiterator'>
>>> for landMass in worldWalker:
...     print "Visit {0}.".format(landMass,)
...
Visit AF. Visit AS. Visit EU. Visit AU. Visit AN. Visit SA. Visit NA.
```

- All iterators have a `.next()` method that you can call to get the next element in the sequence. This method takes no arguments. It returns the next element in the sequence, if any. When there are no more elements, it raises a `StopIteration` exception.

```
>>> trafficSignal = [ 'green', 'yellow', 'red' ]
>>> signalCycle = iter(trafficSignal)
>>> type(signalCycle)
```

```

<type 'listiterator'>
>>> signalCycle.next()
'green'
>>> signalCycle.next()
'yellow'
>>> signalCycle.next()
'red'
>>> signalCycle.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration

```

Once an iterator is exhausted, it will continue to raise `StopIteration` indefinitely.

- You can also use an iterator as the right-hand operand of the “in” operator.

```

>>> signalCycle = iter(trafficSignal)
>>> 'red' in signalCycle
True

```

24.3. Generators: Functions that can produce a sequence of values

Unlike conventional functions that return only a single result, a *generator* is a function that produces a sequence of zero or more results.

Generators are a special case of iterators (see Section 24.2, “Iterators: Values that can produce a sequence of values” (p. 111)), so they can be used as the controlling iterable in `for` statements and the other places where iterators are allowed.

In a conventional function, the body of the function is executed until it either executes a `return` statement, or until it runs out of body statements (which is the equivalent of a “`return None`” statement).

By contrast, when a generator function is called, its body is executed until it either has another value to produce, or until there are no more values.

- When a function wishes to return the next generated value, it executes a statement of this form:

```
yield e
```

where the `e` is any Python expression.

The difference between `yield` and `return` is that when a `return` is executed, the function is considered finished with its execution, and all its current state disappears.

By contrast, when a function executes a `yield` statement, execution of the function is expected to resume just after that statement, at the point when the caller of the function needs the next generated value.

- A generator signals that there are no more values by executing this statement:

```
raise StopIteration
```

For an example of a generator, see Section 23.10, “The `yield` statement: Generate one result from a generator” (p. 108).

If you are writing a container class (that is, a class whose instances are containers for a set of values), and you want to define an iterator (see Section 26.3.17, “`__iter__()`: Create an iterator” (p. 134)), that

method can be a generator. Here is a small example. The constructor for class `Bunch` takes a sequence of values and stores them in instance attribute `.__stuffList`. The iterator method `.__iter__()` generates the elements of the sequence in order, except it wraps each of them in parentheses:

```
>>> class Bunch(object):
...     def __init__(self, stuffList):
...         self.__stuffList = stuffList
...     def __iter__(self):
...         for thing in self.__stuffList:
...             yield "({})".format(thing)
...         raise StopIteration
...
>>> mess = Bunch(('lobster Thermidor', 'crevettes', 'Mornay'))
>>> for item in mess:
...     print item,
...
(lobster Thermidor) (crevettes) (Mornay)
>>> messWalker = iter(mess)
>>> for thing in messWalker: print thing,
...
(lobster Thermidor) (crevettes) (Mornay)
```

24.4. Decorators

The purpose of a Python decorator is to replace a function or method with a modified version *at the time it is defined*. For example, the original way to declare a static method was like this:

```
def someMethod(x, y):
    ...
    someMethod = staticmethod(someMethod)
```

Using Python's decorator syntax, you can get the same effect like this:

```
@staticmethod
def someMethod(x, y):
    ...
```

In general, a function or method may be preceded by any number of decorator expressions, and you may also provide arguments to the decorators.

- If a function `f` is preceded by a decorator expression of the form “`@d`”, it is the equivalent of this code:

```
def f(...):
    ...
f = d(f)
```

- You may provide a parenthesized argument list after the name of your decorator. A decorator expression `d(...)` is the equivalent of this code:

```
def f(...):
    ...
f = d(...)(f)
```

First, the decorator is called with the argument list you provided. It must return a callable object. That callable is then called with one argument, the decorated function. The name of the decorated function is then bound to the returned value.

- If you provide multiple decorators, they are applied inside out, in sequence from the last to the first.

Here is an example of a function wrapped with two decorators, of which the second has additional arguments:

```
@f1
@f2('Pewty')
def f0(...):
    ...
```

This is the equivalent code without using decorators:

```
def f0(...):
    ...
f0 = f1 ( f2('Pewty') ( f0 ) )
```

First function `f2` is called with one argument, the string `'Pewty'`. The return value, which must be callable, is then called with `f0` as its argument. The return value from that call is then passed to `f1`. Name `f0` is then bound to the return value from the call to `f1`.

25. Exceptions: Error signaling and handling

Python's exception system provides a way to signal error conditions and other disruptions in normal processing, and also a way for programs to recover from these conditions.

- Section 25.1, "Definitions of exception terms" (p. 114).
- Section 25.2, "Life cycle of an exception" (p. 115).
- Section 25.3, "Built-in exceptions" (p. 116).

25.1. Definitions of exception terms

Some definitions:

- To *raise* an exception means to signal that the program cannot proceed normally due to an error or other condition. (In other programming languages, such as Java, the equivalent term is to *throw* an exception.)

Two values accompany the raising of an exception: the *type* and the *value*. For example, if a program attempts to open an existing disk file but there is no such file, the type is `IOError`, and the value is an instance of the `IOError` class that contains additional information about this error.

For more information about raising exceptions, see Section 23.6, "The `raise` statement: Cause an exception" (p. 103).

- A program may choose to *handle* an exception. That is, a program may say that if a certain exception or category of exceptions occurs in a specific block of code, Python must execute another code block called a *handler*.
- A *traceback* is a message from Python showing where an exception occurred.

If you type a statement in conversational mode that causes an exception, you will see a short traceback like this:

```
>>> x = 59 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

The above example showed that the offending statement was read from the standard input stream (<stdin>).

When looking at a traceback, always look at the last line first. It tells you the general type of exception (in the example, a `ZeroDivisionError`), followed by additional details (“integer division or modulo by zero”).

If an exception occurs inside one or more function calls, the traceback will give a complete list of the functions involved, from outermost to innermost. Again, the last line shows the exception type and details.

```
>>> def f(): g()
...
>>> def g(): h()
...
>>> def h(): return 1/0
...
>>> f()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in f
  File "<stdin>", line 1, in g
  File "<stdin>", line 1, in h
ZeroDivisionError: integer division or modulo by zero
```

25.2. Life cycle of an exception

If you anticipate that executing a particular statement may cause an exception and you don't want your program to terminate and display a traceback, you can use a `try` construct to specify handlers to be executed if an exception occurs. For details, see Section 23.8, “The `try` statement: Anticipate exceptions” (p. 105).

If an exception occurs inside a function and it is not handled at that level by a `try` construct, Python will work back through the pending function calls until it either finds a handler for that exception or runs out of pending function calls.

If there are multiple handlers for the exception in calling functions, the innermost will be used. If there are no handlers for the exception in calling functions, you will get a stack traceback and the program will terminate.

```
>>> def f():
...     try:
...         g()
...     except ValueError, detail:
...         print "Caught a ValueError:", detail.message
...
>>> def g(): h()
...
>>> def h():
...     raise ValueError('This is a test.')
```

```
...
>>> f()
Caught a ValueError: This is a test.
```

In the example above, function `f()` calls function `g()`, which in turn calls function `h()`. Function `h()` raises a `ValueError` exception, but there is no `try:` block around it. Python looks to see if there is a `ValueError` handler in `g()`, but there is not. Finally a handler for `ValueError` is found inside function `f()`, so control resumes inside that handler. Note that no stack traceback is displayed, because the `ValueError` exception was handled successfully.

25.3. Built-in exceptions

Python defines a complete hierarchy of built-in exception classes. When you write a handler, you can specify any class in this hierarchy, and that handler will apply to that class *and any derived classes*. This allows you to write generic handlers that catch whole groups of exception types.

In describing this hierarchy, we will use indentation to show the subclass/parent class relationships. Generally a `raise` statement will name one of the “leaf” classes, that is, a class that does not have any subclasses. Ancestor classes that are not usually raised are marked with an asterisk (*) in the section below.

- **BaseException***: This is the ancestor of all exception classes. The constructor for this class takes one argument, a string containing the error message. That argument is available as the `.message` attribute.
- **SystemExit**: Raise this exception to terminate execution of your program; it is a special case in that it does not produce a stack traceback. In module `sys`, the `.exit()` method raises this exception.
It is possible to write a handler for this exception. To defeat such a handler and force immediate termination, import module `os` and use method `os._exit()`.
- **KeyboardInterrupt**: Raised when the user signals an interruption with the keyboard (*del* under Windows, or *Control-C* in Linux or Mac environments).
This class inherits from `BaseException` rather than from `Exception` so that catch-all handlers for class `Exception` will not prevent program termination.
- **Exception***: This is the preferred base class for all built-in and user-defined exceptions. If you want to write a handler for all these exception types, use `Exception` as the handler's type.

```
>>> try:
...     x = 1 / 0
... except Exception, detail:
...     print "Fail:", detail.message
...
Fail: integer division or modulo by zero
>>> try:
...     x = noSuchVariable
... except Exception, detail:
...     print "Fail:", detail.message
...
Fail: name 'noSuchVariable' is not defined
```

Warning

A catch-all handler like this can mask any number of errors. Do not use such a handler unless your program must absolutely stay running.

- **StopIteration**: This is the exception that a generator must raise in order to signal that no more generated values are available. See Section 24.3, “Generators: Functions that can produce a sequence of values” (p. 112).
- **StandardError***: This is the base class for all built-in exceptions that are considered errors.
 - **ArithmeticError***: This is the base class for errors involving arithmetic computations.
 - **FloatingPointError**: This is raised for arithmetic errors involving the `float` type.
 - **OverflowError**: This is raised when the result of an operation cannot be represented.
 - **ZeroDivisionError**: An attempt to divide by zero.
 - **AssertionError**: An `assert` statement has failed. See Section 22.2, “The `assert` statement: Verify preconditions” (p. 94).
 - **AttributeError**: Failure to access an attribute.
 - **EnvironmentError***: Errors caused by functions outside of Python, such as the operating system or peripheral devices.
 - **IOError**: Errors related to file input or output.
 - **OSError**: Errors signaled from the operating system.
 - **ImportError**: Failure to import a module or to import items from a module.
 - **LookupError***: Superclass for errors caused by attempts to retrieve values from inside a container class.
 - **IndexError**: Attempt to retrieve a sequence member $S[I]$, where I is not a valid index in sequence S .
 - **KeyError**: Attempt to retrieve a dictionary member $D[K]$, where K is not a valid key in D .
 - **MemoryError**: No more processor memory is available.
 - **NameError**: Attempt to retrieve a name that is not defined.
 - **UnboundLocalError**: Attempt to retrieve the value of a local name when no value has yet been assigned to it.
 - **RuntimeError**: An error that doesn't fit the other categories.
 - **NotImplementedError**: This is the preferred way for the virtual methods of a base class to signal that they have not been replaced by a concrete method in a derived class.
 - **SyntaxError**: Attempt to execute invalid Python source code.
 - **TypeError**: Attempt to perform an operation on a value that does not support that operation, such as trying to use exponentiation (`**`) on a string.
 - **ValueError**: Caused by an operation that is performed on values of the correct type, but the actual values are not valid. Example: taking a negative number to a fractional power.

26. Classes: Defining your own types

This section assumes you already understand the basics of object-oriented programming in Python, and that you know the meaning of concepts such as class, instance, method, and attribute. For a general tutorial on these concepts, see the introduction to object-oriented Python programming²⁶ in the Tech Computer Center's *Python tutorial*²⁷.

Here is the general form of the class declaration for some class C with one or more parent classes P_1, P_2, \dots :

```
class C(P1, P2, ...):  
    attribute definitions  
    ...
```

To declare a class that does not inherit from any parent classes:

```
class C:  
    attribute definitions  
    ...
```

The *attribute definitions* may include any number of `def` blocks that declare methods of the class, and any number of class variable declarations.

Functionally, a class is really just a namespace. This namespace is just a place to store the pieces of the class mechanisms: its methods and class variables.

- When Python reads a “class” declaration, it creates a new, empty namespace.
- When Python reads a “def” within a class, the name of that method is added to the class's namespace.
- If you define a class variable (that is, if you assign a value to a name inside a class but outside of any methods of the class), the class variable's name and value are added to the namespace.

A brief conversational session may serve to illustrate these concepts. We'll make use of the built-in function `dir()` to show the contents of the class's namespace; see Section 21.5, “`dir()`: Display a namespace's names” (p. 81).

```
>>> class Taunter: 1  
...     tauntCount = 0 2  
...     def taunt(self): 3  
...         print "Go away, or I shall taunt you a second time!"  
...  
>>> dir(Taunter) 4  
['__doc__', '__module__', 'taunt', 'tauntCount']  
>>> type(Taunter.__doc__)  
<type 'NoneType'>  
>>> Taunter.__module__  
'__main__'  
>>> Taunter.tauntCount 5  
0  
>>> Taunter.taunt 6  
<unbound method Taunter.taunt>
```

- 1** When Python reads this line, it adds the name `Taunter` to the current local namespace, bound to a new, empty namespace of type `class`.

²⁶ <http://www.nmt.edu/tcc/help/pubs/lang/pytut/obj-intro.html>

²⁷ <http://www.nmt.edu/tcc/help/pubs/lang/pytut/>

- 2 Because this assignment takes place inside class `Taunter` but not inside a `def`, name `tauntCount` becomes a class variable, bound to an `int` value of zero.
- 3 The next two lines define a method named `taunt()` within the class.
- 4 After we've finished entering the class definition, we use `dir(Taunter)` to see what names are in the class's namespace. Variables `__doc__` and `__module__` are added automatically. Because there was no documentation string in the class, `__doc__` is bound to `None`. The `__module__` variable has the value `'__main__'` because the class was entered in conversational mode.
- 5 To retrieve the value of a class variable `V` in class `C`, use the syntax `"C.V"`.
- 6 Name `taunt` in the class namespace is bound to an object of type "unbound method." An unbound method is a method (function) that is inside a class, but it is not associated with an instance of the class.

An instance of a class is also a namespace. When the instance is created, all the names from the class's namespace are copied into the instance namespace. From that point on, any changes made to the instance's namespace do not affect the class namespace:

```

>>> frenchy = Taunter()           1
>>> dir(frenchy)
['__doc__', '__module__', 'taunt', 'tauntCount']
>>> frenchy.where = 'crenelations' 2
>>> dir(frenchy)                   3
['__doc__', '__module__', 'where', 'taunt', 'tauntCount']
>>> frenchy.where
'crenelations'
>>> dir(Taunter)
['__doc__', '__module__', 'taunt', 'tauntCount']
>>> frenchy.tauntCount              4
0
>>> frenchy.tauntCount += 1        5
>>> frenchy.tauntCount
1
>>> Taunter.tauntCount
0
>>> type(frenchy.taunt)            6
<type 'instancemethod'>
>>> frenchy.taunt()                7
Go away, or I shall taunt you a second time!
>>> Taunter.taunt(frenchy)        8
Go away, or I shall taunt you a second time!

```

- 1 This class does not have a constructor (`__init__`) method, so when an instance is created, the instance is a namespace with the same names as the class, and the same values.
- 2 This line adds a new name `where` to the instance's namespace. It is bound to the string value `'crenelations'`.
- 3 Note that the instance namespace now contains the name `where`, but the class's namespace is unchanged.
- 4 To retrieve an attribute `A` of an instance `I`, use the syntax `"I.A"`. Initially, the instance variable has the same value as the class variable of the same name.
- 5 Here, we add one to the instance variable `tauntCount`. The instance variable has the new value, but the class variable `tauntCount` is unchanged.
- 6 Within the instance namespace, name `taunt` is now a *bound method*: it is associated with the instance `frenchy`.

The next two lines show two equivalent methods of calling the `taunt` method.

- 7 Most method calls are *bound method calls*. To call a bound method *B* of an instance *I*, use the syntax "*I.B(...)*".

When a method *B* is bound to an instance *I*, the instance namespace *I* becomes the "self" argument passed in to the method.

- 8 This line has the same effect as the previous line, but it is an *unbound method call*.

The expression "`Taunter.taunt`" retrieves the unbound method from the class definition. When you call an unbound method, you must supply the "self" argument explicitly as the first argument.

Unbound method calls are not terribly common, but you will need to know about them when you write the constructor for a derived class: you must call the parent class constructor as an unbound call. Generally, if class *D* has parent class *C*, the derived class might look something like this:

```
class D(C):
    def __init__(self, ...):
        C.__init__(self, ...)
        ...
```

Namespaces are very much like dictionaries. Where a dictionary has unique keys, a namespace has unique names. As a matter of fact, classes and instances have a special built-in attribute called "`__dict__`" which, for most purposes, is the namespace as a dictionary. Continuing the examples above:

```
>>> Taunter.__dict__
{'taunt': <function taunt at 0xb7ed002c>, '__module__': '__main__', 'tauntCount': 0, '__doc__': None}
>>> newFrenchy=Taunter()
>>> newFrenchy.__dict__
{}
>>> frenchy.__dict__
{'tauntCount': 1, 'where': 'crenelations'}
```

The class's dictionary has the four names we expect: the built-ins `__module__` and `__doc__`, the class variable `tauntCount`, and the method `taunt`.

But notice that the `__dict__` attribute of the newly created instance `newFrenchy` does not have the four names copied from the class. In fact, it is empty. And the `__dict__` of instance `frenchy` contains only the names that have changed since its instantiation.

What actually happens when you refer to an attribute is that Python looks first in the instance's `__dict__`; if the name is not found there, it looks in the `__dict__` of the class. For derived classes, Python will also search the `__dict__` attributes of all the ancestor classes.

So, in our example, a reference to `frenchy.tauntCount` would find the value of 1 in the instance. A reference to `newFrenchy.tauntCount` would fail to find that name in `newFrenchy.__dict__`, but would succeed in finding the class variable value 0 in `Taunter.__dict__['tauntCount']`.

Let's now look at the life cycles of classes in more detail. Due to improvements made in the language since it was first introduced, Python has two kinds of classes, old-style and new-style. We encourage you to use new-style classes; old-style classes will no longer be supported in the next major release, Python 3000.

- All new-style classes must declare at least one parent class that is either the top-level class `object` or some other class that derives ultimately from `object`. Such a class is said to be *derived from*, or *inherits from*, the `object` class.

To declare a new-style class C that inherits from `object`:

```
class C(object):
    ...class methods and variables...
```

- An old-style class is one that doesn't declare a parent class at all, or a class that inherits from an existing old-style class. The life cycle of an old-style class is described in Section 26.1, "Old-style classes" (p. 121).

In most respects, the two classes perform identically.

- We'll start by explaining old-style classes in Section 26.1, "Old-style classes" (p. 121).
- To benefit from the many functional improvements of new-style classes, and especially if you expect to migrate your code to the major changes of Python 3.0, see Section 26.2, "Life cycle of a new-style class" (p. 124).

26.1. Old-style classes

Old-style classes are those declared without a parent class, or classes that inherit from an existing old-style class.

Here is an outline of the birth, life, and death of an old-style class and its instances.

- Section 26.1.1, "Defining an old-style class" (p. 121).
- Section 26.1.2, "Instantiation of an old-style class: The constructor, `__init__()`" (p. 121).
- Section 26.1.3, "Attribute references in old-style classes" (p. 122).
- Section 26.1.4, "Method calls in an old-style class" (p. 123).
- Section 26.1.5, "Instance deletion: the destructor, `__del__()`" (p. 123).

26.1.1. Defining an old-style class

To define an old-style class C with no parent class, use this general form:

```
class C:
    ...class methods and variables...
```

To create a class that inherits from one or more parent classes P_1, P_2, \dots :

```
class C(P1, P2, ...):
    ...class methods and variables...
```

As Python reads the definition of your class, it first creates a new, empty namespace called the *class namespace*. You can access the class namespace directly as an attribute named `__dict__`, a dictionary whose keys are the names in that namespace.

As Python reads each method or class variable in your class declaration, it adds the name of that method or variable to the class namespace.

26.1.2. Instantiation of an old-style class: The constructor, `__init__()`

The creation of a new instance of a class happens when a running Python program encounters a call to that class, that is, the class name with a pair of parentheses after it, with zero or more arguments inside the parentheses.

Here is the general form:

```
C(a1, a2, ...)
```

The instance creation (also called *instantiation*) is handled by the `__init__()` or constructor method.

1. First Python creates the instance with an empty `.__dict__` attribute that will contain the instance's values.
2. Python then calls the constructor. The argument list for this call always has the special first argument `self` (the instance), followed by whatever arguments were used in the initial call. The constructor call is equivalent to this:

```
C.__init__(self, a1, a2, ...)
```

3. The constructor method then executes. Typically the constructor will set up new instance attributes by assignments of this form:

```
self.name = expression
```

When the constructor finishes executing, the instance is returned to the constructor's caller.

26.1.3. Attribute references in old-style classes

The names inside the instance are called *attributes*. (The methods are technically attributes—attributes that are of type function.) There are three operations on attributes: *get*, *set*, and *delete*.

- To *get* an attribute means to retrieve its value. Python searches the instance's `.__dict__` for the attribute's name; if that name is found, its value is returned. If the instance's `.__dict__` does not have a binding for the given name, Python searches the class's `.__dict__`. If no value is found there, Python searches the namespaces of the ancestor classes (if any).

```
>>> class C:
...     def __init__(self, x):
...         self.thingy = x
...
>>> c=C(42)
>>> c.thingy
42
>>> c.__dict__['thingy']
42
```

When you call a method *M* of an instance *I* in an expression of the form “*I.M*(...)”, this is considered just another attribute “get” operation: the get operation *I.M* retrieves the method, and then that method is called using the arguments inside the “(...)”.

- To *set* an attribute means to give it a value. If there is an existing attribute with the same name, its old value is discarded, and the attribute name is bound to the new value. The new value is stored in the instance's `.__dict__`.

```
>>> c.thingy
42
>>> c.thingy = 58
>>> c.thingy
58
>>> c.__dict__['thingy']
58
```

- You can *delete* an attribute from an instance using a `del` statement (see Section 22.3, “The `del` statement: Delete a name or part of a value” (p. 95)).

```
>>> c.thingy
58
>>> del c.thingy
>>> c.thingy
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: C instance has no attribute 'thingy'
```

In addition to ordinary attributes and methods, your class can accept references to names that do *not* exist in the instance or class namespace. You can define special methods that will be called when some statement tries to get, set, or delete an attribute that isn't found in the instance's `.__dict__`. See Section 26.3.14, “`__getattr__()`: Handle a reference to an unknown attribute” (p. 134), Section 26.3.21, “`__setattr__()`: Intercept all attribute changes” (p. 135), and Section 26.3.9, “`__delattr__()`: Delete an attribute” (p. 132).

If all else fails—if an attribute is not found in the instance's namespace and the class does not provide a special method that handles the attribute reference—Python will raise an `AttributeError` exception.

26.1.4. Method calls in an old-style class

There are two different ways to call a method *M* of some class *C*:

- Most calls are *bound method* calls of this form, where *I* is an instance of some class *C*:

```
I.methodName(a1, a2, ...)
```

The instance *I* replaces `self` as the first argument when setting up the arguments to be passed to the method.

- The following form, called an *unbound method call*, is exactly equivalent to the above:

```
C.methodName(i, a1, a2, ...)
```

Here is a demonstration of the equivalence of bound and unbound method calls.

```
>>> class C:
...     def __init__(self, x):
...         self.x = x
...     def show(self, y):
...         print "*** ({0},{1}) ***".format(self.x, y)
...
>>> c=C(42)
>>> c.show(58)
*** (42,58) ***
>>> C.show(c,58)
*** (42,58) ***
```

26.1.5. Instance deletion: the destructor, `.__del__()`

When there are no values that refer to an instance any more, the storage occupied by the instance is recycled. However, if there are certain cleanup operations that must be done (such as closing external files), these operations can be placed into a *destructor* method that will be called before recycling the instance. Here is the general form of a destructor:

```
def __del__(self):
    ...cleanup statements...
```

26.2. Life cycle of a new-style class

Most of the features of new-style classes are the same as for old-style classes. This section will discuss only the differences. We won't cover a few of the more obscure advanced features here; for information on such topics as descriptors and metaclasses, see the "Data model" section of the *Python Reference Manual*²⁸.

The declaration of a new-style class looks the same as for an old-style class, with one constraint: the class must inherit from the universal base class named `object`, or from one or more other new-style classes.

- Section 26.2.1, "`__new__()`: New instance creation" (p. 124).
- Section 26.2.2, "Attribute access control in new-style classes" (p. 125).
- Section 26.2.3, "Properties in new-style classes: Fine-grained attribute access control" (p. 125).
- Section 26.2.4, "Conserving memory with `__slots__`" (p. 125).

26.2.1. `__new__()`: New instance creation

New-style classes have a new special method name, `__new__()`, that is called on instantiation *before* the constructor. It handles the creation of a new instance.

- The `__new__()` method is called when an instance is created.
- Method `__new__()` is always a static method (see Section 26.4, "Static methods" (p. 136)), even if you do not specifically make it a static method.
- A constructor call for some class `C` has this general form:

```
C(*p, **k)
```

That is, it can have any number of positional arguments and any number of keyword arguments.

The equivalent call to the `__new__()` method will look like this:

```
def __new__(cls, *p, **k):
    ...
```

The first argument `cls` must be the class being created.

- The `__new__()` method must call the parent class's `__new__()` method to create the instance.

For example, if your class inherits directly from `object`, you must call:

```
object.__new__(cls)
```

The value returned by that call is the new instance.

- In most cases, the `__new__()` method will return a new instance of `cls`, and that class's `__init__()` will then be called with that instance as its `self` argument, and the positional and keyword arguments `p` and `k` will be passed to that constructor as well.

```
>>> class Test(object):
...     def __new__(cls, *p, **k):
```

²⁸ <http://docs.python.org/reference/datamodel.html>

```

...     inst = object.__new__(cls)
...     return inst
...     def __init__(self, *p, **k):
...         print "p={0} k={1}".format(p, k)
...
>>> t=Test('egg', 'kale', sauce='Bearnaise')
p=('egg', 'kale') k={'sauce': 'Bearnaise'}

```

However, if the `.__new__()` method does *not* return an instance of class `cls`, the constructor method `.__init__()` will *not* be called. This allows the class more control over how new instances are created and initialized. You can return an instance of an entirely different class if you like.

26.2.2. Attribute access control in new-style classes

New-style classes give you more ways to control what happens when an instance's attribute is accessed.

Here is the general procedure for access to attribute `a` of instance `i`, where `C` is the class of `i`.

1. If the instance has a `__getattr__()` special method (see Section 26.3.15, “`__getattr__()`: Intercept all attribute references” (p. 134)), execute that method, which must either return the attribute value or raise `AttributeError`.
2. If the instance has a `__slots__` attribute (see Section 26.2.4, “Conserving memory with `__slots__`” (p. 125)), return the value of the slot with name `a`. If `a` does not match any of the slot names, or if the named slot has never been set to a value, raise `AttributeError`.
3. If `a` is a key in `i.__dict__`, return the corresponding value.
4. Search for attribute `a` in class `C`. If that fails, search all the parent classes of `C` all the way back to `object`.
5. If all searches in the preceding step failed and the instance has a `.__getatr__()` special method, call that method. See Section 26.3.14, “`.__getatr__()`: Handle a reference to an unknown attribute” (p. 134); please note the differences from Section 26.3.15, “`__getattr__()`: Intercept all attribute references” (p. 134).
6. If all the above steps fail to produce a value, raise `AttributeError`.

26.2.3. Properties in new-style classes: Fine-grained attribute access control

The outline of attribute access in Section 26.2.2, “Attribute access control in new-style classes” (p. 125) is slightly oversimplified in one respect. Any of the attribute search steps in this procedure may produce a *property* rather than the actual attribute value.

A property is a special object that is produced by the `property()` function. For a discussion of the three types of attribute access (get, set, and delete), the protocols for the accessor functions, and examples, see Section 21.15, “`property()`: Create an access-controlled attribute” (p. 86).

26.2.4. Conserving memory with `__slots__`

Normally, you can add new attributes to an instance's namespace with any name you want. The instance's `.__dict__` attribute is effectively a dictionary, and you can add any number of names to it.

However, in a new-style class, you may specify a given, limited set of attribute names that are allowed in instances of the class. There are two reasons why you might want to do this:

- If your program is going to create large numbers of instances of a class, to the point where you may run out of memory, you can save some storage within each instance by sacrificing the ability to add arbitrary attribute names.
- If you limit the set of permissible attribute names, Python will detect any reference to a name not in the permissible set, and raise an `AttributeError` exception. This may help you catch certain programming errors.

To limit the set of attribute names in a new-style class, assign to a class variable named `__slots__` a tuple containing the allowable names, like this:

```
__slots__ = (n1, n2, ...)
```

Here's a small example. Suppose you want instances of class `Point` to contain nothing more than two attributes named `.x` and `.y`:

```
>>> class Point(object):
...     __slots__ = ('x', 'y')
...     def __init__(self, abscissa, ordinate):
...         self.x, self.y = abscissa, ordinate
...
>>> x2=Point(3, 7)
>>> x2.x
3
>>> x2.y
7
>>> x2.temperature = 98.6
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Point' object has no attribute 'temperature'
```

When you declare a `__slots__` attribute in a new-style class, instances will *not* have a `__dict__` attribute.

26.3. Special method names

Within a class, a number of reserved method names have special meaning. Here is a list of the ones covered in this document.

<code>__abs__</code>	Section 26.3.4, "Special methods to emulate built-in functions" (p. 130)
<code>__add__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__and__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__call__</code>	Section 26.3.5, " <code>__call__()</code> : What to do when someone calls an instance" (p. 131)
<code>__cmp__</code>	Section 26.3.6, " <code>__cmp__()</code> : Generalized comparison" (p. 131)
<code>__complex__</code>	Section 26.3.4, "Special methods to emulate built-in functions" (p. 130)
<code>__contains__</code>	Section 26.3.7, " <code>__contains__()</code> : The "in" and "not in" operators" (p. 132)
<code>__del__</code>	Section 26.3.8, " <code>__del__()</code> : Destructor" (p. 132)
<code>__delattr__</code>	Section 26.3.9, " <code>__delattr__()</code> : Delete an attribute" (p. 132)
<code>__delitem__</code>	Section 26.3.10, " <code>__delitem__()</code> : Delete one item of a sequence" (p. 132)
<code>__divmod__</code>	Section 26.3.4, "Special methods to emulate built-in functions" (p. 130)

<code>__div__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__divmod__</code>	Section 26.3.4, "Special methods to emulate built-in functions" (p. 130)
<code>__enter__</code>	Section 26.3.11, " <code>__enter__</code> : Context manager initialization" (p. 133)
<code>__exit__</code>	Section 26.3.12, " <code>__exit__</code> : Context manager cleanup" (p. 133)
<code>__eq__</code>	Section 26.3.1, "Rich comparison methods" (p. 129)
<code>__floordiv__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__float__</code>	Section 26.3.4, "Special methods to emulate built-in functions" (p. 130)
<code>__ge__</code>	Section 26.3.1, "Rich comparison methods" (p. 129)
<code>__getattr__</code>	Section 26.3.14, " <code>__getattr__()</code> : Handle a reference to an unknown attribute" (p. 134)
<code>__getattribute__</code>	Section 26.3.15, " <code>__getattribute__()</code> : Intercept all attribute references" (p. 134)
<code>__getitem__</code>	Section 26.3.16, " <code>__getitem__()</code> : Get one item from a sequence or mapping" (p. 134)
<code>__gt__</code>	Section 26.3.1, "Rich comparison methods" (p. 129)
<code>__hex__</code>	Section 26.3.4, "Special methods to emulate built-in functions" (p. 130)
<code>__iadd__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__iand__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__idiv__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__ifloordiv__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__ilshift__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__imod__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__imul__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__init__</code>	Section 26, "Classes: Defining your own types" (p. 118)
<code>__int__</code>	Section 26.3.4, "Special methods to emulate built-in functions" (p. 130)
<code>__invert__</code>	Section 26.3.3, "Unary operator special methods" (p. 130)
<code>__ior__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__ipow__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__irshift__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__isub__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__iter__</code>	Section 26.3.17, " <code>__iter__()</code> : Create an iterator" (p. 134)
<code>__ixor__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__le__</code>	Section 26.3.1, "Rich comparison methods" (p. 129)
<code>__len__</code>	Section 26.3.4, "Special methods to emulate built-in functions" (p. 130)
<code>__long__</code>	Section 26.3.4, "Special methods to emulate built-in functions" (p. 130)
<code>__lshift__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__lt__</code>	Section 26.3.1, "Rich comparison methods" (p. 129)
<code>__mod__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)
<code>__mul__</code>	Section 26.3.2, "Special methods for binary operators" (p. 129)

<code>__ne__</code>	Section 26.3.1, “Rich comparison methods” (p. 129)
<code>__neg__</code>	Section 26.3.3, “Unary operator special methods” (p. 130)
<code>__new__</code>	Section 26.2.1, “ <code>__new__()</code> : New instance creation” (p. 124)
<code>__nonzero__</code>	Section 26.3.18, “ <code>__nonzero__()</code> : True/false evaluation” (p. 135)
<code>__oct__</code>	Section 26.3.4, “Special methods to emulate built-in functions” (p. 130)
<code>__or__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__pos__</code>	Section 26.3.3, “Unary operator special methods” (p. 130)
<code>__pow__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__radd__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__rand__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__rdiv__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__repr__</code>	Section 26.3.19, “ <code>__repr__()</code> : String representation” (p. 135)
<code>__reversed__</code>	Section 26.3.20, “ <code>__reversed__()</code> : Implement the <code>reversed()</code> function” (p. 135)
<code>__rfloordiv__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__rlshift__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__rmod__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__rmul__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__ror__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__rpow__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__rrshift__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__rshift__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__rsub__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__rxor__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__setattr__</code>	Section 26.3.21, “ <code>__setattr__()</code> : Intercept all attribute changes” (p. 135)
<code>__setitem__</code>	Section 26.3.22, “ <code>__setitem__()</code> : Assign a value to one item of a sequence” (p. 135)
<code>__str__</code>	Section 26.3.4, “Special methods to emulate built-in functions” (p. 130)
<code>__sub__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)
<code>__unicode__</code>	Section 26.3.4, “Special methods to emulate built-in functions” (p. 130)
<code>__xor__</code>	Section 26.3.2, “Special methods for binary operators” (p. 129)

The most important special method name is the class constructor, `__init__()`.

- For a general introduction to class constructors, see Section 26, “Classes: Defining your own types” (p. 118).
- For old-style class constructors, see Section 26.1.2, “Instantiation of an old-style class: The constructor, `__init__()`” (p. 121).
- For new-style class constructors, see Section 26.2, “Life cycle of a new-style class” (p. 124).

Many special methods fall into broad categories:

- Section 26.3.1, “Rich comparison methods” (p. 129): for methods that implement comparisons such as “<=” and “==”.
- Section 26.3.2, “Special methods for binary operators” (p. 129): for operators that operate on two operands, such as “%”.
- Section 26.3.3, “Unary operator special methods” (p. 130): for operators that operate on a single operand, such as negation, “-”.
- Section 26.3.4, “Special methods to emulate built-in functions” (p. 130): for classes that handle calls to built-in functions such as “str()”.

26.3.1. Rich comparison methods

These special methods allow your class to specify what happens when comparison operators such as “<=” are used, and the left-hand operand is an instance of your class. (In most cases the right-hand operand is also an instance of the same class, but this is not actually required.)

In each case, the calling sequence for the method must look like this:

```
def __method__(self, other):
    ...
```

The `self` argument is the left-hand operand and the `other` argument is the operand on the right hand of the operator.

Each method must return a numeric value:

- A negative number indicates that `self` precedes `other`.
- Zero indicates that `self` and `other` are considered equal.
- A positive number indicates that `other` precedes `self`.
- If the method does not implement the operation, it may return the special value `NotImplemented`.

Operator	Method name
==	<code>__eq__</code>
>=	<code>__ge__</code>
>	<code>__gt__</code>
<=	<code>__le__</code>
<	<code>__lt__</code>
!=	<code>__ne__</code>

26.3.2. Special methods for binary operators

Your class can define special methods with these names to tell Python how to handle binary operators such as “*” or “%”. In each case, the calling sequence will look something like this:

```
def __method__(self, other):
    ...
```

The `self` argument is the left-hand operand, and the `other` argument is the right-hand operand. Your method will return the result of the operation.

For each operator, you may supply up to three methods:

- The method in the first column performs the normal operation.

- The method in the second column is used when the left-hand operand does not support the given operation and the operands have different types. In these methods, `self` is the *right-hand* operand and `other` is the left-hand operand.
- The third column implements the “augmented assignment” operators such as “+=”. For example, for method `__iadd__(self, other)`, the method must perform the equivalent of “`self += other`”.

Operator	Normal	Reversed	Augmented
+	<code>__add__</code>	<code>__radd__</code>	<code>__iadd__</code>
&	<code>__and__</code>	<code>__rand__</code>	<code>__iand__</code>
/	<code>__div__</code>	<code>__rdiv__</code>	<code>__idiv__</code>
//	<code>__floordiv__</code>	<code>__rfloordiv__</code>	<code>__ifloordiv__</code>
<<	<code>__lshift__</code>	<code>__rlshift__</code>	<code>__ilshift__</code>
%	<code>__mod__</code>	<code>__rmod__</code>	<code>__imod__</code>
*	<code>__mul__</code>	<code>__rmul__</code>	<code>__imul__</code>
	<code>__or__</code>	<code>__ror__</code>	<code>__ior__</code>
**	<code>__pow__</code>	<code>__rpow__</code>	<code>__ipow__</code>
>>	<code>__rshift__</code>	<code>__rrshift__</code>	<code>__irshift__</code>
-	<code>__sub__</code>	<code>__rsub__</code>	<code>__isub__</code>
^	<code>__xor__</code>	<code>__rxor__</code>	<code>__ixor__</code>

26.3.3. Unary operator special methods

You can define these special methods in your class to specify what happens when a unary operator such as “-” (negate) is applied to an instance of the class.

In each case, the definition will look like this:

```
def __method__(self):
    ...
```

The method returns the result of the operation.

Operator	Method
~	<code>__invert__</code>
-	<code>__neg__</code>
+	<code>__pos__</code>

26.3.4. Special methods to emulate built-in functions

You can define special methods that will handle calls to some of Python's built-in functions. The number of arguments will be the same as for the built-in functions, except that `self` is always the first argument.

For example, a special method to handle calls to function `divmod(x, y)` will look like this:

```
def __divmod__(self, other):
    ...
```

In this method, the value of the first argument will be passed to `self` and the second argument to `other`.

Function	Method
abs	<code>__abs__</code>
complex	<code>__complex__</code>
divmod	<code>__divmod__</code>
hex	<code>__hex__</code>
int	<code>__int__</code>
len	<code>__len__</code>
long	<code>__long__</code>
mod	<code>__mod__</code>
oct	<code>__oct__</code>
str	<code>__str__</code>
unicode	<code>__unicode__</code>

26.3.5. `__call__()`: What to do when someone calls an instance

If a class has a `__call__()` method, its instances can be called as if they were functions.

Any arguments passed in that function call become arguments to the `__call__()` method. Example:

```
>>> class CallMe(object):
...     def __call__(self, *p, **k):
...         print "CallMe instance called with:"
...         print "Positional arguments", p
...         print "Keyword arguments", k
...
>>> c=CallMe()
>>> c(1, 'rabbit', fetchez='la vache', hamster='elderberries')
CallMe instance called with:
Positional arguments (1, 'rabbit')
Keyword arguments {'fetchez': 'la vache', 'hamster': 'elderberries'}
```

26.3.6. `__cmp__()`: Generalized comparison

The purpose of this special method is to implement comparison operations between instances. It will be called in these situations:

- If the built-in `cmp()` function is called to compare an instance of a class to some other value, and the class has a `__cmp__()` method, that method is called to perform the comparison.
- When an instance appears on the left-hand side of a comparison (relational) operator, and that instance's class has a the corresponding rich-comparison method (such as `__eq__()` for the `"=="` operator; see Section 26.3.1, "Rich comparison methods" (p. 129)), the rich-comparison method will be called to perform the comparison. and, if so, that method is called.

The comparison operators are `"<"`, `"<="`, `"=="`, `"!="`, `">"`, and `">="`.

However, if the class does not have the correct rich-comparison method, but it does have a `.__cmp__()` method, that method will be called to evaluate the comparison.

The calling sequence is:

```
def __cmp__(self, other):  
    ...
```

The convention for return values is the same one described in Section 20.8, “`cmp()`: Compare two values” (p. 63): negative if `self` precedes `other`, positive if `other` precedes `self`, zero if they are considered equal.

26.3.7. `__contains__()`: The “in” and “not in” operators

This special method defines how instances of a class behave when they appear as the right-hand operand of Python’s “in” and “not in” operators.

Here is the calling sequence:

```
def __contains__(self, x):  
    ...
```

The method returns `True` if `x` is considered to be in `self`, `False` otherwise.

26.3.8. `__del__()`: Destructor

If a class has a `.__del__()` method, that method is called when an instance is deleted. For details, see Section 26.1.5, “Instance deletion: the destructor, `.__del__()`” (p. 123).

26.3.9. `__delattr__()`: Delete an attribute

If a class has a `.__delattr__()` method, that method is called when an attribute is deleted. Here is the calling sequence:

```
def __delattr__(self, name):  
    ...
```

The `name` argument is the name of the attribute to be deleted.

26.3.10. `__delitem__()`: Delete one item of a sequence

This method defines the behavior of a `del` statement of this form:

```
del s[i]
```

Such a statement can be used either on objects that act like sequences, where `i` specifies the position of the element to be deleted, or mapping objects (that is, dictionary-like objects), where `i` is the key of the key-value pair to be deleted.

The calling sequence is:

```
def __delitem__(self, i):  
    ...
```

26.3.11. `__enter__`: Context manager initialization

For a general explanation of context managers, see Section 23.9, “The `with` statement and context managers” (p. 107). A class that acts a content manager must provide this special method as well as the one described in Section 26.3.12, “`__exit__`: Context manager cleanup” (p. 133).

The expression that follows the word `with` in the `with` statement must evaluate to a context manager, whose `__enter__()` method is called with no arguments (other than `self`). The value it returns will be bound to the variable named in the `with` statement's `as` clause, if one was provided.

26.3.12. `__exit__`: Context manager cleanup

For a general explanation of context managers, see Section 23.9, “The `with` statement and context managers” (p. 107). A class that acts a content manager must provide this special method as well as the one described in Section 26.3.11, “`__enter__`: Context manager initialization” (p. 133).

When the body of a `with` statement completes its execution, the `__exit__()` method of the related context manager *M* is called with three arguments. If the block terminated without raising an exception, all three arguments will be `None`; otherwise see below.

```
M.__exit__(self, eType, eValue, eTrace)
```

eType

The type of the exception.

eValue

The exception instance raised.

eTrace

A traceback instance. For more information about stack traces, see the documentation for the `traceback` module²⁹.

Your `__exit__()` method's return value determines what happens next if the block raised an exception. If it returns `True`, Python ignores the exception and proceeds with execution at a point just after the `with` block. If you don't want your context manager to suppress the exception, don't re-raise it explicitly, just return `False` and Python will then re-raise the exception.

26.3.13. `__format__`: Implement the `format()` function

Use this special method to determine how an instance of your class acts when passed to the function described in Section 20.16, “`format()`: Format a value” (p. 66). The calling sequence is:

```
def __format__(self, fmt):  
    ...
```

The interpretation of the `fmt` argument is entirely up to you. The return value should be a string representation of the instance.

If the call to the `format()` function does not provide a second argument, the `fmt` value passed to this method will be an empty string.

²⁹ <http://docs.python.org/library/traceback.html>

26.3.14. `__getattr__()`: Handle a reference to an unknown attribute

This method, if present, handles statements that get an attribute value of an instance, but there is no such entry in the instance's namespace. The calling sequence is:

```
def __getattr__(self, name):  
    ...
```

The argument is the name of the desired attribute. The method must either return the attribute's value or raise an `AttributeError` exception.

Compare Section 26.3.15, "`__getattribute__()`: Intercept all attribute references" (p. 134), which is called even if the instance namespace *does* have an attribute with the desired name.

26.3.15. `__getattribute__()`: Intercept all attribute references

This method is called whenever an attribute is referenced, whether the instance or class namespace has an attribute with the given name or not. It works only with new-style classes.

For an overview and examples, see Section 26.2.2, "Attribute access control in new-style classes" (p. 125).

26.3.16. `__getitem__()`: Get one item from a sequence or mapping

If a class defines it, this special method is called whenever a value is retrieved from a sequence or mapping (dictionary-like object) using the syntax "`v[i]`", where `v` is the sequence or mapping and `i` is a position in a sequence, or a key in a mapping.

Here is the calling sequence:

```
def __getitem__(self, i):  
    ...
```

The method either returns the corresponding item or raises an appropriate exception: `IndexError` for sequences or `KeyError` for mappings.

26.3.17. `__iter__()`: Create an iterator

If a class defines an `__iter__()` method, that method is called:

- Whenever the built-in `iter()` function is applied to an instance of the class.
- In any situation where the instance is iterated over, such as when it appears as the controlling iterable of a `for` statement.

The calling sequence is:

```
def __iter__(self):  
    ...
```

The return value must be an iterator. An iterator is any object that has a `.next()` method that returns the next value in the sequence, or raises `StopIteration` when the sequence is exhausted. For more information, see Section 24.2, "Iterators: Values that can produce a sequence of values" (p. 111).

26.3.18. `__nonzero__()`: True/false evaluation

If a class defines it, this special method is called whenever an instance is converted to a Boolean value, either implicitly (for example, when it is the test in an “if” statement) or explicitly via the built-in `bool()` function. Here is the calling sequence:

```
def __nonzero__(self):  
    ...
```

Return a Boolean value, either `True` or `False`.

26.3.19. `__repr__()`: String representation

If a class defines it, this special method is called to find the “representation” of an object. There are two ways to get a representation:

- Via the function described in Section 21.17, “`repr()`: Representation” (p. 88).
- Via the “back-quote operator”, enclosing a Python expression in open single quotes.

The calling sequence is:

```
def __repr__(self):  
    ...
```

The method returns the representation of `self` as a string.

26.3.20. `__reversed__()`: Implement the `reversed()` function

If provided, this method allows the `reversed()` function to be applied to an instance of the class. It returns an iterator that iterates over the contained elements in reverse order.

26.3.21. `__setattr__()`: Intercept all attribute changes

If a class defines it, this method is called whenever a new value is stored into an attribute. Calling sequence:

```
def __setattr__(self, name, value):  
    ...
```

The method sets the attribute given by the `name` argument to the `value` argument, or raises `AttributeError` if that operation is not permitted.

26.3.22. `__setitem__()`: Assign a value to one item of a sequence

If a class defines it, this method is called whenever a new value is stored into a sequence or mapping (dictionary-like object), such as in statements of this form:

```
V[i] = expr
```

Here is the calling sequence:

```
def __setitem__(self, i, value):  
    ...
```

For sequence-type objects, the `i` argument specifies the position in the sequence to be modified. For mappings, the `i` argument is the key under which the `value` is to be stored.

26.4. Static methods

A *static method* of a Python class is a method that does not obey the usual convention in which `self`, an instance of the class, is the first argument to the method.

To declare a static method, declare the method normally, and wrap it with the built-in `staticmethod` function, using either of the techniques described in Section 21.20, “`staticmethod()`: Create a static method” (p. 90).

Once you have declared a method to be static, the arguments you pass it are exactly the arguments it receives. Example:

```
>>> class Hobbs:
...     @staticmethod
...     def represent():
...         print "Hobbs represent!"
...
>>> Hobbs.represent()
Hobbs represent!
```

26.5. Class methods

A class method is similar to a static method (see Section 26.4, “Static methods” (p. 136)), except that its first argument is always the class that contains the method.

To declare a class method, use a declaration of this general form:

```
@classmethod
def methodName(cls, ...):
    ...
```

When this method is called, the first argument (`cls`) will be the class containing *methodName*.

There are two ways to call a class method: using its class *C*, or an instance *i*. These two general forms are:

```
C.methodName(...)
i.methodName(...)
```

In the first case, the class *C* is passed as the `cls` argument of the method. In the second case, the class of instance *i* is passed as the `cls` argument.

```
>>> class Jal(object):
...     @classmethod
...     def whatClass(cls, n):
...         print "cls={0} n={1}".format(cls, n)
...     def __init__(self, color):
...         self.color = color
...
>>> Jal.whatClass(5)
cls=<class '__main__.Jal'> n=5
>>> eunice=Jal('green')
```

```
>>> eunice.whatClass(17)
cls=<class '__main__.Jal'> n=17
```

27. pdb: The Python interactive debugger

The Python debugger allows you to monitor and control execution of a Python program, to examine the values of variables during execution, and to examine the state of a program after abnormal termination (post-mortem analysis).

- Section 27.1, “Starting up pdb” (p. 137).
- Section 27.2, “Functions exported by pdb” (p. 137).
- Section 27.3, “Commands available in pdb” (p. 138).

27.1. Starting up pdb

To execute any Python statement under the control of `pdb`:

```
>>> import pdb
>>> import yourModule
>>> pdb.run('yourModule.test()') # Or any other statement
```

where `yourModule.py` contains the source code you want to debug.

To debug a Python script named `myscript.py`:

```
python /usr/lib/python2.5/pdb.py myscript.py
```

To perform post-mortem analysis:

```
>>> import pdb
>>> import yourModule
>>> yourModule.test()
[crash traceback appears here]
>>> pdb.pm()
(pdb)
```

Then you can type debugger commands at the `(pdb)` prompt.

27.2. Functions exported by pdb

The `pdb` module exports these functions:

`pdb.run(stmt[, globals[, locals]])`

Executes any Python statement. You must provide the statement *as a string*. The debugger prompts you before beginning execution. You can provide your own global name space by providing a *globals* argument; this must be a dictionary mapping global names to values. Similarly, you can provide a local namespace by passing a dictionary *locals*.

`pdb.runeval(expr[, globals[, locals]])`

This is similar to the `pdb run` command, but it evaluates an expression, rather than a statement. The *expr* is any Python expression in string form.

pdb.runcall(*func*[,*arg*]....)

Calls a function under `pdb` control. The *func* must be either a function or a method. Arguments after the first argument are passed to your function.

27.3. Commands available in `pdb`

The debugger prompts with a line like this:

```
(pdb)
```

At this prompt, you can type any of the `pdb` commands discussed below. You can abbreviate any command by omitting the characters in square brackets. For example, the `where` command can be abbreviated as simply `w`.

expr

Evaluate an expression *expr* and print its value.

!*stmt*

Execute a Python statement *stmt*. The “!” may be omitted if the statement does not resemble a `pdb` command.

(empty line)

If you press Enter at the `(pdb)` prompt, the previous command is repeated. The `list` command is an exception: an empty line entered after a `list` command shows you the next 11 lines after the ones previously listed.

a[*rgs*]

Display the argument names and values to the currently executing function.

b[*reak*] [[*filename*:]*lineno*[,*condition*]]

The `break` command sets a breakpoint at some location in your program. If execution reaches a breakpoint, execution will be suspended and you will get back to the `(pdb)` prompt.

This form of the command sets a breakpoint at a specific line in a source file. Specify the line number within your source file as *lineno*; add the *filename*: if you are working with multiple source files, or if your source file hasn't been loaded yet.

You can also specify a conditional breakpoint, that is, one that interrupts execution only if a given *condition* evaluates as true. For example, the command `break 92, i>5` would break at line 92 only when `i` is greater than 5.

When you set a breakpoint, `pdb` prints a “breakpoint number.” You will need to know this number to clear the breakpoint.

b[*reak*] [*function*[,*condition*]]

This form of the `break` command sets a breakpoint on the first executable statement of the given *function*.

c[*ont*][*inue*]]

Resume execution until the next breakpoint (if any).

c[*lear*] [*lineno*]

If used without an argument, clears all breakpoints. To clear one breakpoint, give its breakpoint number (see `break` above).

h[*elp*] [*cmd*]

Without an argument, prints a list of valid commands. Use the *cmd* argument to get help on command *cmd*.

l[ist] [begin[,end]]

Displays your Python source code. With no arguments, it shows 11 lines centered around the current point of execution. The line about to be executed is marked with an arrow (->), and the letter B appears at the beginning of lines with breakpoints set.

To look at a given range of source lines, use the *begin* argument to list 11 lines around that line number, or provide the ending line number as an *end* argument. For example, the command `list 50,65` would list lines 50-65.

n[ext]

Like `step`, but does not stop upon entry to a called function.

q[uit]

Exit `pdb`.

r[eturn]

Resume execution until the current function returns.

s[tep]

Single step: execute the current line. If any functions are called in the current line, `pdb` will break upon entering the function.

tbreak

Same options and behavior as `break`, but the breakpoint is temporary, that is, it is removed after the first time it is hit.

w[here]

Shows your current location in the program as a stack traceback, with an arrow (->) pointing to the current stack frame.

28. Commonly used modules

The sections below discuss only a tiny fraction of the official and unofficial module library of Python. For a full set, consult the *Python Library Reference*³⁰.

If you want to use any of these modules, you must import them. See Section 22.6, “The `import` statement: Use a module” (p. 97).

28.1. math: Common mathematical operations

This module provides the usual basic transcendental mathematical functions. All trig functions use angles in radians. (For a similar set of functions using the complex number system, see the Python library documentation for the `cmath` module³¹.)

The `math` module has two constant attributes:

<code>pi</code>	The constant 3.14159...
<code>e</code>	The base of natural logarithms, 2.71828...

Functions in this module include:

<code>acos(x)</code>	Angle (in radians) whose cosine is <i>x</i> , that is, arccosine of <i>x</i> .
----------------------	--

³⁰ <http://docs.python.org/library/>

³¹ <http://docs.python.org/library/cmath.html>

<code>acosh(x)</code>	Inverse hyperbolic cosine of x
<code>asin(x)</code>	Arcsine of x .
<code>asinh(x)</code>	Inverse hyperbolic sine of x
<code>atan(x)</code>	Arctangent of x .
<code>atanh(x)</code>	Inverse hyperbolic tangent of x
<code>atan2(y, x)</code>	Angle whose slope is y/x , even if x is zero.
<code>ceil(x)</code>	True ceiling function, defined as the smallest integer that is greater than or equal to x . For example, <code>ceil(3.9)</code> yields 4.0, while <code>ceil(-3.9)</code> yields -3.0.
<code>cos(x)</code>	Cosine of x , where x is expressed in radians.
<code>cosh(x)</code>	Hyperbolic cosine of x .
<code>degrees(x)</code>	For x in radians, returns that angle in degrees.
<code>erf(x)</code>	Error function.
<code>erfc(x)</code>	Error function complement.
<code>exp(x)</code>	e to the x power.
<code>fabs(x)</code>	Returns the absolute value of x as a <code>float</code> value.
<code>factorial(n)</code>	Returns the factorial of n , which must be a nonnegative integer.
<code>floor(x)</code>	True floor function, defined as the largest integer that is less than or equal to x . For example, <code>floor(3.9)</code> is 3.0, and <code>floor(-3.9)</code> is -4.0.
<code>fmod(x, y)</code>	Returns $(x - \text{int}(x/y)*y)$.
<code>frexp(x)</code>	For a float value x , returns a tuple (m, e) where m is the mantissa and e is the exponent. For $x=0.0$, it returns $(0.0, 0)$; otherwise, <code>abs(m)</code> is a float in the half-open interval $[0.5, 1)$ and e is an integer, such that $x == m*2**e$.
<code>gamma(x)</code>	Gamma function.
<code>hypot(x, y)</code>	The square root of (x^2+y^2) .
<code>ldexp(x, i)</code>	Returns $x * (2**i)$. This is the inverse of <code>frexp()</code> .
<code>lgamma(x)</code>	Natural log of <code>abs(gamma(x))</code> .
<code>log(x[, b])</code>	With one argument, returns the natural log of x . With the second argument, returns the log of x to the base b .
<code>log10(x)</code>	Common log (base 10) of x .
<code>modf(x)</code>	Returns a tuple (f, i) where f is the fractional part of x , i is the integral part (as a float), and both have the same sign as x .
<code>radians(x)</code>	For x in degrees, returns that angle in radians.
<code>sin(x)</code>	Sine of x .
<code>sinh(x)</code>	Hyperbolic sine of x .
<code>sqrt(x)</code>	Square root of x .
<code>tan(x)</code>	Tangent of x .
<code>tanh(x)</code>	Hyperbolic tangent of x .

28.2. string: Utility functions for strings

Variables and functions for working with character strings.

ascii_letters

A string containing all the letters from `ascii_uppercase` and `ascii_lowercase`.

ascii_lowercase

The characters that are considered lowercase letters in the ASCII³² character set, namely:

```
"abcdefghijklmnopqrstuvwxyz"
```

ascii_uppercase

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

The value of this constant does not depend on the locale setting; see Section 19.4, “What is the locale?” (p. 60).

digits

The decimal digits: `"0123456789"`.

hexdigits

The hexadecimal digits: `"0123456789abcdefABCDEF"`.

letters

A string containing all the characters that are considered letters in the current locale.

lowercase

A string containing all the characters that are considered lowercase letters in the current locale.

maketrans(s, t)

Builds a translation table to be used as the first argument to the `.translate()` string method. The arguments `s` and `t` are two strings of the same length; the result is a translation table that will convert each character of `s` to the corresponding character of `t`.

```
>>> import string
>>> bingo=string.maketrans("lLrR", "rRLl")
>>> "Cornwall Llanfair".translate(bingo)
'Colnwarr Rranfail'
>>> "Cornwall Llanfair".translate(bingo, 'ai')
'Colnwrr Rrnfl'
```

octdigits

The octal digits: `"01234567"`.

printable

A string containing all the printable characters.

punctuation

All printable characters that are not letters or digits in the current locale. If ASCII³³ is the locale's current encoding, these characters are included:

```
!"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~
```

uppercase

A string containing all the characters that are considered uppercase letters in the current locale. May not be the same as `ascii_uppercase`

³² <http://en.wikipedia.org/wiki/ASCII>

³³ <http://en.wikipedia.org/wiki/ASCII>

whitespace

A string containing all the characters considered to be white space in the current locale. For ASCII this will be:

```
"\t\n\x0b\x0c\r "
```

See Section 9.2, “Definition of “whitespace”” (p. 15).

28.3. random: Random number generation

This module provides for the generation of pseudorandom numbers.

Any one who considers arithmetical methods of producing random digits is, of course, in a state of sin. For, as has been pointed out several times, there is no such thing as a random number—there are only methods to produce random numbers, and a strict arithmetic procedure of course is not such a method.

— John von Neumann³⁴

choice(S)

Returns a randomly selected element from an iterable *S*.

```
>>> for i in range(9): print choice ( ['s', 'h', 'd', 'c'] ),  
...  
s c c h c s s h d
```

normalvariate(m, s)

Generate a normally distributed pseudorandom number with mean *m* and standard deviation *s*.

randint(x, y)

Returns a random integer in the closed interval [*x*,*y*]; that is, any result *r* will satisfy $x \leq r \leq y$.

```
>>> for i in range(20): print randint(1,6),  
...  
3 4 6 3 2 1 1 2 1 2 1 2 3 3 3 5 6 4 4 2
```

random()

Returns a random float in the half-open interval [0.0, 1.0); that is, for any result *r*, $0.0 \leq r < 1.0$.

```
>>> for count in range(48):  
...     print "{0:.3f}".format(random()),  
...     if (count % 12) == 11: print  
...  
0.012 0.750 0.899 0.339 0.371 0.561 0.358 0.931 0.822 0.990 0.682 0.847  
0.245 0.541 0.992 0.151 0.394 0.335 0.702 0.885 0.986 0.350 0.417 0.748  
0.918 0.103 0.109 0.328 0.423 0.180 0.203 0.689 0.600 0.794 0.201 0.008  
0.564 0.920 0.906 0.469 0.510 0.818 0.142 0.589 0.590 0.290 0.650 0.889
```

randrange([start,]stop[,step])

Return a random element from the sequence `range(start, stop, step)`.

³⁴ http://en.wikiquote.org/wiki/John_von_Neumann

```

>>> from random import *
>>> for i in range(35): print randrange(4),
...
0 2 2 2 1 1 2 3 1 3 3 2 2 2 3 0 2 0 0 1 2 0 2 1 1 1 2 2 2 1 1 3 1 1 2
>>> for i in range(35): print randrange(1,5),
...
3 3 2 1 1 1 4 4 3 2 1 1 3 2 1 2 4 4 1 4 2 4 4 1 1 1 1 1 4 4 1 1 2 2 1
>>> range(2,18,5)
[2, 7, 12, 17]
>>> for i in range(28): print randrange(2,18,5),
...
12 2 7 2 17 17 7 7 12 17 17 2 7 17 12 7 7 12 17 17 7 12 7 7 7 7 7

```

shuffle(L)

Randomly permute the elements of a sequence *L*.

Here's an example. First we build a (small) deck of cards, using a list comprehension to build a list of all possible combinations of three ranks (ace, king, queen) and four suits (spades, hearts, diamonds, and clubs). Then we shuffle the deck twice and inspect the results.

```

>>> ranks = 'AKQ'
>>> suits = 'shdc'
>>> deck = [ r+s
...         for s in suits
...         for r in ranks ]
>>> deck
['As', 'Ks', 'Qs', 'Ah', 'Kh', 'Qh', 'Ad', 'Kd', 'Qd', 'Ac', 'Kc', 'Qc']
>>> shuffle(deck)
>>> deck
['Qh', 'Ks', 'Kh', 'As', 'Kc', 'Kd', 'Qd', 'Qc', 'Ah', 'Ad', 'Qs', 'Ac']
>>> shuffle(deck)
>>> deck
['As', 'Qs', 'Ks', 'Kc', 'Ad', 'Kh', 'Qh', 'Ac', 'Ah', 'Qc', 'Qd', 'Kd']

```

uniform(x,y)

Returns a random float in the half-open interval $[x,y)$; that is, each result *r* will satisfy $x \leq r < y$.

An assortment of other pseudorandom distributions is available. See the *Python Library Reference*³⁵ for details.

28.4. time: Clock and calendar functions

This module provides minimal time and date functions. For a newer and more complete module, see the `datetime` module³⁶.

- *Epoch time* is the time in seconds since some arbitrary starting point. For example, Unix measures time in seconds since January 1, 1970.
- *UTC* is Coordinated Universal Time, the time on the zero meridian (which goes through London).
- *DST* refers to Daylight Savings Time.

³⁵ <http://docs.python.org/library/random.html>

³⁶ <http://docs.python.org/library/datetime>

A *time tuple* is a 9-tuple T with these elements, all integers:

$T[0]$	Four-digit year.	$T[5]$	Second, in $[0,59]$.
$T[1]$	Month, 1 for January, 12 for December.	$T[6]$	Day of week, 0 for Monday, 6 for Sunday.
$T[2]$	Day of month, in $[1,31]$.	$T[7]$	Ordinal day of the year, in $[1,366]$.
$T[3]$	Hour, in $[0,23]$.	$T[8]$	DST flag: 1 if the time is DST, 0 if it is not DST, and -1 if unknown.
$T[4]$	Minute, in $[0,59]$.		

Contents of the `time` module:

altzone

The local DST offset, in seconds west of UTC (negative for east of UTC).

asctime([T])

For a time-tuple T , returns a string of exactly 24 characters with this format:

```
"Thu Jun 12 15:25:31 1997"
```

The default time is now.

clock()

The accumulated CPU time of the current process in seconds, as a float.

ctime([E])

Converts an epoch time E to a string with the same format as `asctime()`. The default time is now.

daylight

Nonzero if there is a DST value defined locally.

gmtime([E])

Returns the time-tuple corresponding to UTC at epoch time E ; the DST flag will be zero. The default time is now.

localtime([E])

Returns the time-tuple corresponding to local time at epoch time E . The default time is now.

mktime(T)

Converts a time-tuple T to epoch time as a float, where T is the *local* time.

sleep(s)

Suspend execution of the current process for s seconds, where s can be a float or integer.

strftime(f[, t])

Time formatting function; formats a time-tuple t according to format string f . The default time t is now. As with the old string format operator, format codes start with `%`, and other text appears unchanged in the result. See the table of codes below.

time()

The current epoch time, as a float.

timezone

The local non-DST time zone as an offset in seconds west of UTC (negative for east of UTC). This value applies when daylight savings time is not in effect.

tzname

A 2-tuple (*s*, *d*) where *s* is the name of the non-DST time zone locally and *d* is the name of the local DST time zone. For example, in Socorro, NM, you get ('MST' , 'MDT').

Format codes for the `strftime` function include:

%a	Abbreviated weekday name, e.g., 'Tue'.
%A	Full weekday name, e.g., 'Tuesday'.
%b	Abbreviated month name, e.g., 'Jul'.
%B	Full month name, e.g., 'July'.
%d	Day of the month, two digits with left zero fill; e.g. '03'.
%H	Hour on the 24-hour clock, two digits with zero fill.
%I	Hour on the 12-hour clock, two digits with zero fill.
%j	Day of the year as a decimal number, three digits with zero fill, e.g. '366'.
%m	Decimal month, two digits with zero fill.
%M	Minute, two digits with zero fill.
%p	Either 'AM' or 'PM'. Midnight is considered AM and noon PM.
%S	Second, two digits with zero fill.
%w	Numeric weekday: 0 for Sunday, 6 for Saturday.
%y	Two-digit year. Not recommended! ³⁷
%Y	Four-digit year.
%Z	If there is a time zone, a string representing that zone; e.g., 'PST'.
%%	Outputs the character %.

28.5. re: Regular expression pattern-matching

The `re` module provides functions for matching strings against regular expressions. See the O'Reilly book *Mastering Regular Expressions* by Friedl and Oram for the whys and hows of regular expressions. We discuss only the most common functions here. Refer to the *Python Library Reference*³⁸ for the full feature set.

- Section 28.5.1, "Characters in regular expressions" (p. 145).
- Section 28.5.2, "Functions in the `re` module" (p. 147).
- Section 28.5.3, "Compiled regular expression objects" (p. 148).
- Section 28.5.4, "Methods on a `MatchObject`" (p. 148).

28.5.1. Characters in regular expressions

Note: The raw string notation `r' . . . '` is most useful for regular expressions; see raw strings (p. 15), above.

These characters have special meanings in regular expressions:

.	Matches any character except a newline.
---	---

³⁷ <http://en.wikipedia.org/wiki/Y2k>

³⁸ <http://docs.python.org/library/re.html>

<code>^</code>	Matches the start of the string.
<code>\$</code>	Matches the end of the string.
<code>r*</code>	Matches zero or more repetitions of regular expression <i>r</i> .
<code>r+</code>	Matches one or more repetitions of <i>r</i> .
<code>r?</code>	Matches zero or one <i>r</i> .
<code>r*?</code>	Non-greedy form of <i>r*</i> ; matches as few characters as possible. The normal <code>*</code> operator is greedy: it matches as much text as possible.
<code>r+?</code>	Non-greedy form of <i>r+</i> .
<code>r??</code>	Non-greedy form of <i>r?</i> .
<code>r{m,n}</code>	Matches from <i>m</i> to <i>n</i> repetitions of <i>r</i> . For example, <code>r{x{3,5}}</code> matches between three and five copies of letter 'x'; <code>r(bl){4}</code> matches the string 'b1b1b1b1'.
<code>r{m,n}?</code>	Non-greedy version of the previous form.
<code>[...]</code>	Matches one character from a set of characters. You can put all the allowable characters inside the brackets, or use <i>a-b</i> to mean all characters from <i>a</i> to <i>b</i> inclusive. For example, regular expression <code>r'[abc]</code> will match either 'a', 'b', or 'c'. Pattern <code>r'[0-9a-zA-Z]</code> will match any single letter or digit.
<code>[^...]</code>	Matches any character <i>not</i> in the given set.
<code>rs</code>	Matches expression <i>r</i> followed by expression <i>s</i> .
<code>r s</code>	Matches either <i>r</i> or <i>s</i> .
<code>(r)</code>	Matches <i>r</i> and forms it into a group that can be retrieved separately after a match; see <code>MatchObject</code> , below. Groups are numbered starting from 1.
<code>(?:r)</code>	Matches <i>r</i> but does not form a group for later retrieval.
<code>(?P<n>r)</code>	Matches <i>r</i> and forms it into a named group, with name <i>n</i> , for later retrieval.
<code>(?P=n)</code>	Matches whatever string matched an earlier <code>(?P<n>r)</code> group.
<code>(?#...)</code>	Comment: the "... " portion is ignored and may contain a comment.
<code>(?=...)</code>	The "... " portion must be matched, but is not consumed by the match. This is sometimes called a lookahead match. For example, <code>r'a(=bcd)</code> matches the string 'abcd' but not the string 'abcxyz'. Compared to using <code>r'abcd</code> as the regular expression, the difference is that in this case the matched portion would be 'a' and not 'abcd'.
<code>(?!...)</code>	This is similar to the <code>(?=...)</code> : it specifies a regular expression that must <i>not</i> match, but does not consume any characters. For example, <code>r'a(?!bcd)</code> would match 'axyz', and return 'a' as the matched portion; but it would not match 'abcdef'. You could call it a negative lookahead match.

The special sequences in the table below are recognized. However, many of them function in ways that depend on the locale; see Section 19.4, "What is the locale?" (p. 60). For example, the `r'\s` sequence matches characters that are considered whitespace in the current locale.

<code>\n</code>	Matches the same text as a group that matched earlier, where <i>n</i> is the number of that group. For example, <code>r'([a-zA-Z]+):\1</code> matches the string "foo:foo".
<code>\A</code>	Matches only at the start of the string.
<code>\b</code>	Matches the empty string but only at the start or end of a word (where a word is set off by whitespace or a non-alphanumeric character). For example, <code>r'foo\b</code> would match "foo" but not "foot".

<code>\B</code>	Matches the empty string when <i>not</i> at the start or end of a word.
<code>\d</code>	Matches any digit.
<code>\D</code>	Matches any non-digit.
<code>\s</code>	Matches any whitespace character.
<code>\S</code>	Matches any non-whitespace character.
<code>\w</code>	Matches any alphanumeric character plus the underbar ' _ '.
<code>\W</code>	Matches any non-alphanumeric character.
<code>\Z</code>	Matches only at the end of the string.
<code>\\</code>	Matches a backslash (<code>\</code>) character.

28.5.2. Functions in the re module

There are two ways to match regular expressions with the `re` module. Assuming you import the module with `import re`, you can test whether a regular expression `r` matches a string `s` with the construct:

```
re.match(r, s)
```

However, if you will be matching the same regular expression many times, the performance will be better if you compile the regular expression like this:

```
re.compile(r)
```

The `re.compile()` function returns a compiled regular expression object. You can then check a string `s` for matching by using the `.match(s)` method on that object.

Here are the functions in module `re`:

compile(*r* [, *f*])

Compile regular expression `r`. This function returns a compiled regular expression object; see Section 28.5.3, “Compiled regular expression objects” (p. 148). To get case-insensitive matching, use `re.I` as the `f` argument. There are other flags that may be passed to the `f` argument; see the *Python Library Reference*³⁹.

match(*r*, *s* [, *f*])

If `r` matches the start of string `s`, return a `MatchObject` (see below), otherwise return `None`.

search(*r*, *s* [, *f*])

Like the `match()` method, but matches `r` anywhere in `s`, not just at the beginning.

split(*r*, *s* [, *maxsplit*=*m*])

Splits string `s` into pieces where pattern `r` occurs. If `r` does not contain groups, returns a list of the parts of `s` that match `r`, in order. If `r` contains groups, returns a list containing all the characters from `s`, with parts matching `r` in separate elements from the non-matching parts. If the `m` argument is given, it specifies the maximum number of pieces that will be split, and the leftovers will be returned as an extra string at the end of the list.

sub(*r*, *R*, *s* [, *count*=*c*])

Replace the leftmost non-overlapping parts of `s` that match `r` using `R`; returns `s` if there is no match. The `R` argument can be a string or a function that takes one `MatchObject` argument and returns the string to be substituted. If the `c` argument is supplied (defaulting to 0), no more than `c` replacements are done, where a value of 0 means do them all.

³⁹ <http://docs.python.org/library/re.html>

28.5.3. Compiled regular expression objects

Compiled regular expression objects returned by `re.compile()` have these methods:

.match(*s* [, *p_s*] [, *p_e*])

If the start of string *s* matches, return a `MatchObject`; if there is no match, return `None`. If *p_s* is given, it specifies the index within *s* where matching is to start; this defaults to 0. If *p_e* is given, it specifies the maximum length of *s* that can be used in matching.

.pattern

The string from which this object was compiled.

.search(*s* [, *p_s*] [, *p_e*])

Like `match()`, but matches anywhere in *s*.

.split(*s* [, *maxsplit*=*m*])

Like `re.split()`.

.sub(*R*, *s* [, *count*=*c*])

Like `re.sub()`.

28.5.4. Methods on a MatchObject

A `MatchObject` is the object returned by `.match()` or other methods. Such an object has these methods and attributes:

.end([*n*])

Returns the location where a match ended. If no argument is given, returns the index of the first character past the match. If *n* is given, returns the index of the first character past where the *n*th group matched.

.endpos

The effective *p_e* value passed to `.match()` or `.search()`.

.group([*n*])

Retrieves the text that matched. If there are no arguments, or if *n* is zero, it returns the entire string that matched.

To retrieve just the text that matched the *n*th group, pass in an integer *n*, where the groups are numbered starting at 1. For example, for a `MatchObject` *m*, `m.group(2)` would return the text that matched the second group, or `None` if there were no second group.

If you have named the groups in your regular expression using a construct of the form `(?P<name>...)`, the *n* argument can be the *name* as a string. For example, if you have a group `(?P<year>[\d]{4})` (which matches four digits), you can retrieve that field using `m.group('year')`.

.groups([*default*])

Return a tuple (*s*₁, *s*₂, ...) containing all the matched strings, where *s*_{*i*} is the string that matched the *i*th group.

For groups that did not match, the corresponding value in the tuple will be `None`, or an optional default value that you specify in the call to this method.

.groupdict([*default*])

Return a dictionary whose keys are the named groups in the regular expression. Each corresponding value will be the text that matched the group. If a group did not match, the corresponding value will be `None`, or an alternate default value that you supply when you call the method.

.lastgroup

Holds the name of the last named group (using the `(?P<n>r)` construct) that matched. It will be `None` if no named groups matched, or if the last group that matched was a numbered group and not a named group.

.lastindex

Holds the index of the last group that matched, or `None` if no groups matched.

.pos

The effective p_s value passed to `.match()` or `.search()`.

.re

The regular expression object used to produce this `MatchObject`.

.span([n])

Returns a 2-tuple `(m.start(n), m.end(n))`.

.start([n])

Returns the location where a match started. If no argument is given, returns the index within the string where the entire match started. If an argument n is given, returns the index of the start of the match for the n th group.

.string

The s argument passed to `.match()` or `.search()`.

28.6. sys: Universal system interface

The services in this module give you access to command line arguments, standard input and output streams, and other system-related facilities.

argv

`sys.argv[0]` is the name of your Python script, or `'-c'` if in interactive mode. The remaining elements, `sys.argv[1:]`, are the command line arguments, if any.

builtin_module_names

A list of the names of the modules compiled into your installation of Python.

exit(n)

Terminate execution with status n .

modules

A dictionary of the modules already loaded.

path

The search path for modules, a list of strings in search order.

Note: You can modify this list. For example, if you want Python to search directory `/u/dora/python/lib` for modules to import before searching any other directory, these two lines will do it:

```
import sys
sys.path.insert(0, "/u/dora/python/lib")
```

platform

A string identifying the software architecture.

stderr

The standard error stream as a file object.

stdin

The standard input stream as a file object.

stdout

The standard output stream as a file object.

28.7. os: The operating system interface

The variables and methods in the `os` module allow you to interact with files and directories. In most cases the names and functionalities are the same as the equivalent C language functions, so refer to Kernighan and Ritchie, *The C Programming Language*, second edition, or the equivalent for more details.

chdir(*p*)

Change the current working directory to that given by string *p*

chmod(*p, m*)

Change the permissions for pathname *p* to *m*. See module `stat`, below, for symbolic constants to be used in making up *m* values.

chown(*p, u, g*)

Change the owner of pathname *p* to user id *u* and group id *g*.

environ

A dictionary whose keys are the names of all currently defined environmental variables, and whose values are the values of those variables.

error

The exception raised for errors in this module.

execv(*p, A*)

Replace the current process with a new process executing the file at pathname *p*, where *A* is a list of the strings to be passed to the new process as command line arguments.

execve(*p, A, E*)

Like `execv()`, but you supply a dictionary *E* that defines the environmental variables for the new process.

_exit(*n*)

Exit the current process and return status code *n*. This method should be used only by the child process after a `fork()`; normally you should use `sys.exit()`.

fork()

Fork a child process. In the child process, this function returns `0`; in the parent, it returns the child's process ID.

getcwd()

Returns the current working directory name as a string.

getegid()

Returns the effective group ID.

geteuid()

Returns the effective user ID.

getgid()

Returns the current process's group ID. To decode user IDs, see the `grp` standard module⁴⁰.

⁴⁰ <http://docs.python.org/library/grp.html>

getpid()

Returns the current process's process ID.

getppid()

Returns the parent process's PID (process ID).

getuid()

Returns the current process's user ID. To decode user IDs, see the `pwd` standard module⁴¹.

kill(*p*, *s*)

Send signal *s* to the process whose process ID is *p*.

link(*s*, *d*)

Create a hard link to *s* and call the link *d*.

listdir(*p*)

Return a list of the names of the files in the directory whose pathname is *p*. This list will never contain the special entries `'.'` and `'..'` for the current and parent directories. The entries may not be in any particular order.

lstat(*p*)

Like `stat()`, but if *p* is a link, you will get the status tuple for the link itself, rather than the file it points at.

makedirs(*p*[, *mode*])

Works like `mkdir()`, but will also create any intermediate directories between existing directories and the desired new directory.

mkdir(*p*[, *m*])

Create a directory at pathname *p*. You may optionally specify permissions *m*; see module `stat` below for the interpretation of permission values.

mkfifo(*p*, *m*)

Create a named pipe with name *p* and open mode *m*. The server side of the pipe will open it for reading, and the client side for writing. This function does not actually open the fifo, it just creates the rendezvous point.

nice(*i*)

Renice (change the priority) of the current process by adding *i* to its current priority.

readlink(*p*)

If *p* is the pathname to a soft (symbolic) link, this function returns the pathname to which that link points.

remove(*p*)

Removes the file with pathname *p*, as in the Unix `rm` command. Raises `OSError` if it fails.

removedirs(*p*)

Similar to `remove()`, but also removes any other parent directory in the path that has no other children.

rename(*p*_o, *p*_n)

Rename path *p*_o to *p*_n.

rmdir(*p*)

Remove the directory at path *p*.

⁴¹ <http://docs.python.org/library/pwd.html>

stat(*p*)

Return a status tuple describing the file or directory at pathname *p*. See module `stat`, below, for the interpretation of a status tuple. If *p* is a link, you will get the status tuple of the file to which *p* is linked.

symlink(*s, d*)

Create a symbolic link to path *s*, and call the link *d*.

system(*c*)

Execute the command in string *c* as a sub-shell. Returns the exit status of the process.

times()

Returns a tuple of statistics about the current process's elapsed time. This tuple has the form (*u, s, u', s', r*) where *u* is user time, *s* is system time, *u'* and *s'* are user and system time including all child processes, and *r* is elapsed real time. All values are in seconds as floats.

tmpfile()

Returns a new, open temporary file, with update mode "w+b". This file will not appear in any directory, and will disappear when it is no longer in use.

umask(*m*)

Sets the "umask" that determines the default permissions for newly created files. Returns the previous value. Each bit set in the umask corresponds to a permission that is *not* granted by default.

uname()

Returns a tuple of strings describing the operating system's version: (*s, n, r, v, m*) where *s* is the name of the operating system, *n* is the name of the processor (node) where you are running, *r* is the operating system's version number, *v* is the major version, and *m* describes the type of processor.

urandom(*n*)

Return a string of *n* random bytes. These bytes should be sufficiently random for use in cryptographic applications.

utime(*p, t*)

The *t* argument must be a tuple (*a, m*) where *a* and *m* are epoch times. For pathname *p*, set the last access time to *a* and the last modification to *m*.

wait()

Wait for the termination of a child process. Returns a tuple (*p, e*) where *p* is the child's process ID and *e* is its exit status.

waitpid(*p, o*)

Like `wait()`, but it waits for the process whose ID is *p*. The option value *o* specifies what to do if the child is still running. If *o* is 0, you wait for the child to terminate. Use a value of `os.WNOHANG` if you don't want to wait.

WNOHANG

See `waitpid()` above.

28.8. stat: Interpretation of file status

The `stat` module contains a number of variables used in encoding and decoding various items returned by certain methods in the `os` module, such as `stat()` and `chmod()`.

First, there are constants for indexing the components of a "status tuple" such as that returned by `os.stat()`:

ST_ETIME	The epoch time of last access (see the <code>time</code> module for interpretation of times).
ST_CTIME	The epoch time of the file's last status change.
ST_DEV	The device number.
ST_GID	The group ID.
ST_INO	The i-node number.
ST_MODE	The file's permissions.
ST_MTIME	The epoch time of last modification.
ST_NLINK	The number of hard links.
ST_SIZE	The current size in bytes.
ST_UID	The user ID.

The following functions are defined in the `stat` module for testing a mode value m , where m is the `ST_MODE` element of the status tuple. Each function is a predicate:

<code>S_ISBLK(m)</code>	Is this a block device?
<code>S_ISCHR(m)</code>	Is this a character device?
<code>S_ISDIR(m)</code>	Is this a directory?
<code>S_ISFIFO(m)</code>	Is this a FIFO?
<code>S_ISLNK(m)</code>	Is this a soft (symbolic) link?
<code>S_ISREG(m)</code>	Is this an ordinary file?
<code>S_ISSOCK(m)</code>	Is this a socket?

These constants are defined for use as mask values in testing and assembling permission values such as those returned by `os.stat()` in Section 28.7, “`os`: The operating system interface” (p. 150).

<code>S_IRGRP</code>	Group read permission.
<code>S_IROTH</code>	World read permission.
<code>S_IRUSR</code>	Owner read permission.
<code>S_ISGID</code>	SGID (set group ID) bit.
<code>S_ISUID</code>	SUID (set user ID) bit.
<code>S_IWGRP</code>	Group write permission.
<code>S_IWOTH</code>	World write permission.
<code>S_IWUSR</code>	Owner write permission.
<code>S_IXGRP</code>	Group execute permission.
<code>S_IXOTH</code>	World execute permission.
<code>S_IXUSR</code>	Owner execute permission.

28.9. `os.path`: File and directory interface

These functions allow you to deal with path names and directory trees. To use a given *method* in this module, import the `os` module and then use `os.path.method()`.

For example, to get the base name of a path p , use `os.path.basename(p)`.

abspath(*p*)

Return the absolute path name that is equivalent to path *p*.

basename(*p*)

Return the base name portion of a path name string *p*. See `split()`, below.

commonprefix(*L*)

For a list *L* containing pathname strings, return the longest string that is a prefix of each element in *L*.

exists(*p*)

Predicate for testing whether pathname *p* exists.

expanduser(*p*)

If *p* is a pathname starting with a tilde character (~), return the equivalent full pathname; otherwise return *p*.

isabs(*p*)

Predicate for testing whether *p* is an absolute pathname (e.g., starts with a slash on Unix systems).

isfile(*p*)

Predicate for testing whether *p* refers to a regular file, as opposed to a directory, link, or device.

islink(*p*)

Predicate for testing whether *p* is a soft (symbolic) link.

ismount(*p*)

Predicate for testing whether *p* is a mount point, that is, whether *p* is on a different device than its parent directory.

join(*p*,*q*)

If *q* is an absolute path, returns *q*. Otherwise, if *p* is empty or ends in a slash, returns *p+q*, but otherwise it returns *p+'/'+q*.

normcase(*p*)

Return pathname *p* with its case normalized. On Unix systems, this does nothing, but on Macs it lowercases *p*.

samefile(*p*,*q*)

Predicate for testing whether *p* and *q* are the same file (that is, the same inode on the same device). This method may raise an exception if `os.stat()` fails for either argument.

split(*p*)

Return a 2-tuple (*H*, *T*) where *T* is the tail end of the pathname (not containing a slash) and *H* is everything up to the tail. If *p* ends with a slash, returns (*p*, ''). If *p* contains no slashes, returns ('', *p*). The returned *H* string will have its trailing slash removed unless *H* is the root directory.

splittext(*p*)

Returns a 2-tuple (*R*, *E*) where *E* is the “extension” part of the pathname and *R* is the “root” part. If *p* contains at least one period, *E* will contain the last period and everything after that, and *R* will be everything up to but not including the last period. If *p* contains no periods, returns (*p*, '').

walk(*p*,*V*,*a*)

Walks an entire directory structure starting at pathname *p*. See below for more information.

The `os.path.walk(p,V,a)` function does the following for every directory at or below *p* (including *p* if *p* is a directory), this method calls the “visitor function” *V* with arguments

<i>V</i> (<i>a</i> , <i>d</i> , <i>N</i>)

<i>a</i>	The same <i>a</i> passed to <code>os.path.walk()</code> . You can use <i>a</i> to provide information to the <code>V()</code> function, or to accumulate information throughout the traversal of the directory structure.
<i>d</i>	A string containing the name of the directory being visited.
<i>N</i>	A list of all the names within directory <i>d</i> . You can remove elements from this list in place if there are some elements of <i>d</i> that you don't want <code>walk()</code> to visit.

