Python 2.7 Quick Reference

Contents

- Front matter
- Invocation Options
- Environment variables
- Lexical entities: keywords, identifiers, string literals, boolean constants, numbers, sequences, dictionaries, sets, operators
- Basic types and their operations: None, bool, Numeric types, sequence types, list, dictionary, string, file, set, named tuples, date/time
- Advanced types
- Statements: assignment, conditional expressions, control flow, exceptions, name space, function def, class def
- Iterators; Generators; Descriptors; Decorators
- Built-in Functions
- Built-in Exceptions
- Standard methods & operators redefinition in user-created Classes
- Special informative state attributes for some types
- Important modules: sys, os, posix, posixpath, shutil, time, string, re, math, compressions
- List of modules in the base distribution
- Workspace exploration and idiom hints
- Python mode for Emacs

Front matter

Version 2.7 (What's new?)
Check updates at http://rgruet.free.fr/#QuickRef.
Please report errors, inaccuracies and suggestions to Richard Gruet (pqr at rgruet.net).

Creative Commons License.

Last updated on April 16, 2013.

Apr 16, 2013
Some corrections, see bottom, by Stefan McKinnon Hoj-Edwards.
Oct, 2011
upgraded by Stefan McKinnon Hoj-Edwards for Python 2.7
Feb 10, 2009
upgraded by Richard Gruet and Josh Stone for Python 2.6
Dec 14, 2006
upgraded by Richard Gruet for Python 2.5
Feb 17, 2005.
upgraded by Richard Gruet for Python 2.4
Oct 3, 2003
upgraded by Richard Gruet for Python 2.3
May 11, 2003, rev 4
upgraded by Richard Gruet for Python 2.2 (restyled by Andrei)
Aug 7, 2001
upgraded by Simon Brunning for Python 2.1
May 16, 2001
upgraded by Richard Gruet and Simon Brunning for Python 2.0
Jun 18, 2000
upgraded by Richard Gruet for Python 1.5.2
Oct 20, 1995
created by Chris Hoffmann for Python 1.3

Color coding:
Features added in 2.7 since 2.6
Features added in 2.6 since 2.5
Features added in 2.5 since 2.4
A link

Originally based on:
- Python Bestiary, author: Ken Manheimer
- Python manuals, authors: Guido van Rossum and Fred Drake
- python-mode.el, author: Tim Peters
- and the readers of comp.lang.python

Useful links:
Invocation Options

```python
python [w] [-BEdhimOQSstuVwXxj] [-c command | scriptFile ] [-] [args]
```

(please note that pythonw does not open a terminal/console; python does)

**Invocation Options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>-B</td>
<td>Prevents module imports from creating .pyc or .pyo files (see also env variable PYTHONDONTWRITEBYTECODE=x and attribute sys.dont_write_bytecode).</td>
</tr>
<tr>
<td>-d</td>
<td>Output parser debugging information (also PYTHONDEBUG=x).</td>
</tr>
<tr>
<td>-E</td>
<td>Ignore environment variables (also PYTHONPATH).</td>
</tr>
<tr>
<td>-h</td>
<td>Print a help message and exit (formerly -?).</td>
</tr>
<tr>
<td>-i</td>
<td>Inspect interactively after running script (also PYTHONINSPECT=x) and force prompts, even if stdin appears not to be a terminal.</td>
</tr>
<tr>
<td>-m</td>
<td>Search for module on sys.path and runs the module as a script. (Implementation improved in 2.5: module runpy)</td>
</tr>
<tr>
<td>-O</td>
<td>Optimize generated by code (also PYTHONOPTIMIZE=x). Asserts are suppressed.</td>
</tr>
<tr>
<td>-OO</td>
<td>Remove doc-strings in addition to the -O optimizations.</td>
</tr>
<tr>
<td>-Q arg</td>
<td>Division options: -Qold (default), -Qwarn, -Qwarnall, -Qnew.</td>
</tr>
<tr>
<td>-s</td>
<td>Disables the user-specific module path (also PYTHONNOUSERSITE=x).</td>
</tr>
<tr>
<td>-t</td>
<td>Print a help message and exit (formerly -?).</td>
</tr>
<tr>
<td>-u</td>
<td>Unbuffered binary stdout and stderr (also PYTHONUNBUFFERED=x).</td>
</tr>
<tr>
<td>-U</td>
<td>Force Python to interpret all string literals as Unicode literals.</td>
</tr>
<tr>
<td>-V</td>
<td>Print the Python version number and exit.</td>
</tr>
<tr>
<td>-x</td>
<td>Skip first line of source, allowing use of non-unix Forms of #! cmd.</td>
</tr>
<tr>
<td>-X</td>
<td>Disable class-based built-ins (for backward compatibility management of exceptions).</td>
</tr>
<tr>
<td>-3</td>
<td>Emit a DeprecationWarning for Python 3.5 incompatibilities that 2to3 cannot trivially fix.</td>
</tr>
<tr>
<td>-c</td>
<td>Specify the command to execute (see next section). This terminates the option list (following options are passed as command arguments to the command).</td>
</tr>
<tr>
<td>scriptFile</td>
<td>The name of a python file (.py) to execute. Read from stdin.</td>
</tr>
<tr>
<td>args</td>
<td>Passed to script or command [in sys.argv[1:]].</td>
</tr>
</tbody>
</table>


- Typical python module header:

```
#!usr/bin/env python
# -*- coding: latin1 -*-
```

Since 2.3 the **encoding** of a Python source file must be declared as one of the two first lines (or defaults to **7 bits Ascii**) [PEP-0263], with the format:

```
# -*- coding: encoding -*-
```

Std **encodings** are defined here, e.g. ISO-8859-1 (aka latin1), iso-8859-15 (latin9), UTF-8... Not all encodings supported, in particular UTF-16 is not supported.
It's now a syntax error if a module contains string literals with 8-bit characters but doesn't have an encoding declaration (was a warning before).

Since 2.5, from __future__ import feature statements must be declared at beginning of source file.

Site customization: File sitecustomize.py is automatically loaded by Python if it exists in the Python path (ideally located in ${PYTHONHOME}/lib/site-packages/).

Tip: when launching a Python script on Windows,

<pythonHome>/python myScript.py args ... can be reduced to:
myScript.py args ... if <pythonHome> is in the PATH envt variable, and further reduced to:
myScript args ... provided that .py; .pyw; .pyc; .pyo is added to the PATHENV envt variable.

Environment variables

<table>
<thead>
<tr>
<th>Environment variables</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>PYTHONHOME</td>
<td>Alternate prefix directory (or prefix:exec_prefix). The default module search path uses prefix/lib Augments the default search path for module files. The format is the same as the shell’s $PATH: one or more directory pathnames separated by ‘:’ or ‘;’ without spaces around (semi-) colons! On Windows Python first searches for Registry key HKEY_LOCAL_MACHINE\Software\Python\PythonCore\x.y\PythonPath (default value). You can create a key named after your application with a default string value giving the root directory path of your appl. Alternatively, you can create a text file with a .pth extension, containing the path(s), one per line, and put the file somewhere in the Python search path (ideally in the site-packages/ directory). It’s better to create a .pth for each application, to make easy to uninstall them.</td>
</tr>
<tr>
<td>PYTHONSTARTUP</td>
<td>If this is the name of a readable file, the Python commands in that file are executed before the first prompt is displayed in interactive mode (no default).</td>
</tr>
<tr>
<td>PYTHONDEBUG</td>
<td>If non-empty, same as -d option</td>
</tr>
<tr>
<td>PYTHONINSPECT</td>
<td>If non-empty, same as -i option</td>
</tr>
<tr>
<td>PYTHONOPTIMIZE</td>
<td>If non-empty, same as -O option</td>
</tr>
<tr>
<td>PYTHONUNBUFFERED</td>
<td>If non-empty, same as -u option</td>
</tr>
<tr>
<td>PYTHONVERBOSE</td>
<td>If non-empty, same as -v option</td>
</tr>
<tr>
<td>PYTHONCASEOK</td>
<td>If non-empty, ignore case in file/module names (imports)</td>
</tr>
<tr>
<td>PYTHONDONTWRITEBYTECODE</td>
<td>If non-empty, same as -B option</td>
</tr>
<tr>
<td>PYTHONIOENCODING</td>
<td>Alternate encodingname of encodingname: errorhandler for stdin, stdout, and stderr, with the same choices accepted by str.encode().</td>
</tr>
</tbody>
</table>
| PYTHONUSERBASE        | Provides a private site-packages directory for user-specific modules. [PEP-0370] - On Unix and Mac OS X, defaults to ~/.local/, and modules are found in a version-specific subdirectory like lib/python2.x/site-packages. - On Windows, defaults to %APPDATA%\Python and %Python26\site-packages, 
| PYTHONNOUSERSITE      | If non-empty, same as -s option |
| PYTHONWARNINGS        | Allows controlling warnings, same as -W option |

Notable lexical entities

Keywords

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

* (List of keywords available in std module: keyword)

Illegitimate Tokens (only valid in strings): $ 7 (plus 8 before 2.4)

A statement must all be on a single line. To break a statement over multiple lines, use "\"", as with the C preprocessor.

Exception: can always break when inside any {}, [], or () pair, or in triple-quoted strings.

More than one statement can appear on a line if they are separated with semicolons (";").

Comments start with "#" and continue to end of line.

Identifiers

(letter | _) (letter | digit | _)*)

* Python identifiers keywords, attributes, etc. are case-sensitive.

  Special forms: __ident (not imported from module import *); __ident__ (system defined name); __ident (class-private name mangling).

String literals

Two flavors: str (standard 8 bits locale-dependent strings, like ascii, iso 8859-1, utf-8, ...) and unicode (16 or 32 bits/char in utf-
16 mode or 32 bits/char in uct-32 mode); one common ancestor |basestring.

Literal
"a string enclosed by double quotes"
'another string delimited by single quotes and with a ' inside'
"a string containing embedded newlines and quote () marks, can be delimited with triple quotes."

**may also use \- double quotes as delimiters**

b"An 8-bit string" - A |bytes instance, a forward-compatible form for an 8-bit string
B"Another 8-bit string"

u'unicode string'
U"Another unicode string"
r's raw string where \ are kept (literalized): handy for regular expressions and windows paths!
R"another raw string" -- raw strings cannot end with a \
u'raw unicode raw string'
U"another raw unicode"

- Use \ at end of line to continue a string on next line.
- Adjacent strings are concatenated, e.g. 'Monty' 'Python' is the same as 'Monty Python'.
- u'hello' + ' world' -> u'hello world' (coerced to unicode)

String Literal Escapes

<table>
<thead>
<tr>
<th>Escape</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>newline</td>
<td>Backslash ()</td>
</tr>
<tr>
<td>\w</td>
<td>Vertical Tab (VT)</td>
</tr>
<tr>
<td>\v</td>
<td>Single quote (')</td>
</tr>
<tr>
<td>\f</td>
<td>Formfeed (FF)</td>
</tr>
<tr>
<td>\r</td>
<td>char with octal value ooo</td>
</tr>
<tr>
<td>\l</td>
<td>char with hex value \r</td>
</tr>
<tr>
<td>\n</td>
<td>Linefeed (LF)</td>
</tr>
<tr>
<td>\t</td>
<td>Bell (BEL)</td>
</tr>
<tr>
<td>\b</td>
<td>Carriage Return (CR)</td>
</tr>
<tr>
<td>\xhh</td>
<td>char with hex value hh</td>
</tr>
<tr>
<td>\b</td>
<td>Backspace (BS)</td>
</tr>
<tr>
<td>\r</td>
<td>Horizontal Tab (TAB)</td>
</tr>
<tr>
<td>\uxxx</td>
<td>Character with 16-bit hex value \r (unicode only)</td>
</tr>
<tr>
<td>\Uxxxxx</td>
<td>Character with 32-bit hex value \r (unicode only)</td>
</tr>
<tr>
<td>\N(name)</td>
<td>Character named in the Unicode database (unicode only), e.g. u'\N{Greek Small Letter Pi}' == u'\u03c0'</td>
</tr>
</tbody>
</table>

\AnyOtherChar

- NUL byte (\000) is not an end-of-string marker; NULs may be embedded in strings.
- Strings (and tuples) are immutable: they cannot be modified.

Boolean constants

- True
- False

Since 2.3, they are of new type bool.

Numbers

- **Decimal integer**: 1234, 123456789056378940L (or 1)
- **Binary integer**: 0b10, 0b10, 0b010101010101010101010101010101010101010101L (begins with a 0b or 0B)
- **Octal integer**: 0177, 00177, 0177777777777777L (begins with a 0, 0o, or 0O)
- **Hex integer**: 0x7f, 0xfffffff00000000L (begins with 0x or 0X)
- **Long integer** (unlimited precision): 1234567890123456L (ends with L or l) or **long**(1234)
- **Float** (double precision): 3.14e-10, .001, 10., 1k3
- **Complex**: 1J, 2+3J, 4+5J (ends with J or j, + separates (float) real and imaginary parts)

Integers and long integers are unified starting from release 2.2 (the L suffix is no longer required)

Sequences

Strings and tuples are immutable, lists are mutable.

- **Strings** (type str and |unicode) of length 0, 1, 2 (see above)
  - "", 's, "hello\n"
- **Tuples** (type tuple) of length 0, 1, 2, etc:
  - ()(1), (1,2) # parentheses are optional if len > 0
- **Lists** (type list) of length 0, 1, 2, etc:
  - [1],[1,2]

- Indexing is 0-based. Negative indices (usually) mean count backwards from end of sequence.
- **Sequence slicing** (start: atm_index, but: less_than_index, step: step): Start defaults to 0, end to len(sequence), step to 1
a = (0, 1, 2, 3, 4, 5, 6, 7)
a[3] == 3
a[-1] == 7
a[2::] == (2, 3)
a[1:] == (1, 2, 3, 4, 5, 6, 7)
a[:3] == (0, 1, 2)
a[1:] == (0, 1, 2, 3, 4, 5, 6, 7)  # makes a copy of the sequence.
a[:1] == (0, 2, 4, 6)  # Only even numbers.
a[::-1] = (7, 6, 5, 4, 3, 2, 1, 0)  # Reverse order.

Dictionaries (Mappings)

Dictionaries (type dict) of length 0, 1, 2, etc: {key: value} {1: 'first'} {1: 'first', 'two': 2, key:value}

Keys must be of a hashable type; Values can be any type.
Dictionaries are unordered, ie. iterating over a dictionary provides key/value pairs in arbitrary order. OrderedDict in the collections module works as regular dictionaries but iterates over keys and values in a guaranteed order depending on when a key was first inserted.

Sets

A set can either be mutable or immutable. Curly brackets {} are used to surround the contents of the resulting mutable set; set literals are distinguished from dictionaries by not containing colons and values. An empty {} continues to represent an empty dictionary; use set() for an empty set.

Operators and their evaluation order

<table>
<thead>
<tr>
<th>Highest</th>
<th>Operator</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>, [[]] {...} <code>...</code></td>
<td>Tuple, list &amp; dict. creation; string conv.</td>
<td></td>
</tr>
<tr>
<td>s$[i] s[i:][j] sattr[f...]</td>
<td>indexing &amp; slicing; attributes, function calls</td>
<td></td>
</tr>
<tr>
<td>+, x, -x, ~x</td>
<td>Unary operators</td>
<td></td>
</tr>
<tr>
<td>x**y</td>
<td>Power</td>
<td></td>
</tr>
<tr>
<td>x*y, x/y, x%y</td>
<td>mult, division, modulo</td>
<td></td>
</tr>
<tr>
<td>x*y, x/y</td>
<td>addition, substraction</td>
<td></td>
</tr>
<tr>
<td>x&lt;y, x&gt;y</td>
<td>Bit shifting</td>
<td></td>
</tr>
<tr>
<td>x&amp;y</td>
<td>Bitwise 'and'; also intersection of sets</td>
<td></td>
</tr>
<tr>
<td>x^y</td>
<td>Bitwise exclusive or</td>
<td></td>
</tr>
<tr>
<td>x&lt;y</td>
<td>Bitwise 'or'; also union of sets</td>
<td></td>
</tr>
<tr>
<td>x in s, x not in s</td>
<td>membership</td>
<td></td>
</tr>
<tr>
<td>not x</td>
<td>boolean negation</td>
<td></td>
</tr>
<tr>
<td>x and y, x or y</td>
<td>boolean and</td>
<td></td>
</tr>
<tr>
<td>lambda args: expr</td>
<td>anonymous function</td>
<td></td>
</tr>
</tbody>
</table>

* Alternate names are defined in module operator (e.g. __add__ and add for +)
* Most operators are overridable

Basic types and their operations

Comparisons (defined between any types)

<table>
<thead>
<tr>
<th>Comparison</th>
<th>Meaning</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>strictly less than</td>
<td>(1)</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
<td></td>
</tr>
<tr>
<td>&gt;</td>
<td>strictly greater than</td>
<td></td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
<td></td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
<td></td>
</tr>
<tr>
<td>!= or &lt;&gt;</td>
<td>not equal to</td>
<td></td>
</tr>
<tr>
<td>is</td>
<td>object identity</td>
<td>(2)</td>
</tr>
<tr>
<td>is not</td>
<td>negated object identity</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Notes:
* Comparison behavior can be overridden for a given class by defining special method __cmp__.
* (1) X < Y < Z < W has expected meaning, unlike C
* (2) Compare object identities (i.e. id(object)), not object values.

None

* None is used as default return value on functions. Built-in single object with type NoneType. Might become a keyword in the future.
* Input that evaluates to None does not print when running Python interactively.
* None is now a constant; trying to bind a value to the name "None" is now a syntax error.
Boolean operators

**Boolean values and operators**

<table>
<thead>
<tr>
<th>Value or Operator</th>
<th>Evaluates to</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>built-in bool(expr)</td>
<td>True if expr is true, False otherwise.</td>
<td>see True, False</td>
</tr>
<tr>
<td>None, numeric zeros, empty sequences and mappings</td>
<td>considered False</td>
<td></td>
</tr>
<tr>
<td>all other values</td>
<td>considered True</td>
<td></td>
</tr>
<tr>
<td>not x</td>
<td>True if x is False, else False</td>
<td></td>
</tr>
<tr>
<td>x or y</td>
<td>if x is False then y, else x</td>
<td>(1)</td>
</tr>
<tr>
<td>x and y</td>
<td>if x is False then x, else y</td>
<td>(1)</td>
</tr>
</tbody>
</table>

**Notes:**
- Truth testing behavior can be overridden for a given class by defining special method `__nonzero__`.
- (1) Evaluate second arg only if necessary to determine outcome.

**Numeric types**

**Floats, integers, long integers, Decimals.**

- Floats (type `float`) are implemented with C doubles.
- Integers (type `int`) are implemented with C longs (signed 32 bits, maximum value is `sys.maxint`)
- Long integers (type `long`) have unlimited size (only limit is system resources).
- Integers and long integers are unified starting from release 2.2 (the `L` suffix is no longer required).
- `int()` returns a long integer instead of raising `OverflowError`. Overflowing operations such as `2**32` no longer trigger `FutureWarning` and return a long integer.
- Since 2.4, new type `Decimal` introduced (see module: `decimal`) to compensate for some limitations of the floating point type, in particular with fractions. Unlike floats, decimal numbers can be represented exactly; exactness is preserved in calculations; precision is user settable via the `Context` type [PEP 327].

**Operators on all numeric types**

**Operators on all numeric types**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs(x)</td>
<td>the absolute value of x</td>
<td></td>
</tr>
<tr>
<td>int(x)</td>
<td>x converted to integer</td>
<td>(2)</td>
</tr>
<tr>
<td>long(x)</td>
<td>x converted to long integer</td>
<td>(2)</td>
</tr>
<tr>
<td>float(x)</td>
<td>x converted to floating point</td>
<td></td>
</tr>
<tr>
<td>~x</td>
<td>x negated</td>
<td></td>
</tr>
<tr>
<td>+x</td>
<td>x unchanged</td>
<td></td>
</tr>
<tr>
<td>x + y</td>
<td>the sum of x and y</td>
<td></td>
</tr>
<tr>
<td>x - y</td>
<td>difference of x and y</td>
<td></td>
</tr>
<tr>
<td>x * y</td>
<td>product of x and y</td>
<td></td>
</tr>
<tr>
<td>x / y</td>
<td>true division of x by y: <code>1/2 -&gt; 0.5</code></td>
<td>(1)</td>
</tr>
<tr>
<td>x // y</td>
<td>floor division operator: <code>1/2 -&gt; 0</code></td>
<td>(1)</td>
</tr>
<tr>
<td>x % y</td>
<td>x modulo y</td>
<td></td>
</tr>
<tr>
<td>divmod(x, y)</td>
<td>the tuple <code>(x//y, x%y)</code></td>
<td></td>
</tr>
<tr>
<td>x ** y</td>
<td>x to the power y (the same as pow(x,y))</td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- (1) `/` is still a floor division (`1/2 == 0`) unless validated by a from `__future__ import division`.
- (2) `int and long` has `bit_length()` method that returns the number of bits necessary to represent its argument in binary.
- classes may override methods `__truediv__` and `__floordiv__` to redefine these operators.

**Bit operators on integers and long integers**

**Bit operators**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>~x</td>
<td>the bits of x inverted</td>
</tr>
<tr>
<td>x ^ y</td>
<td>bitwise exclusive or of x and y</td>
</tr>
<tr>
<td>x &amp; y</td>
<td>bitwise and of x and y</td>
</tr>
<tr>
<td>x</td>
<td>bitwise or of x and y</td>
</tr>
<tr>
<td>x &lt;&lt; n</td>
<td>x shifted left by n bits</td>
</tr>
<tr>
<td>x &gt;&gt; n</td>
<td>x shifted right by n bits</td>
</tr>
</tbody>
</table>

**Complex Numbers**

- Type `complex`, represented as a pair of machine-level double precision floating point numbers.
- The real and imaginary value of a complex number z can be retrieved through the attributes `z.real` and `z.imag`.

**Numeric exceptions**

- `TypeError` raised on application of arithmetic operation to non-number
- `OverflowError` raised when numeric bounds exceeded
- `ZeroDivisionError` raised when zero second argument of div or modulo op
## Operations on all sequence types (lists, tuples, strings)

**Operations on all sequence types**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x in s</code></td>
<td>True if an item of <code>s</code> is equal to <code>x</code>, else False</td>
<td>(3)</td>
</tr>
<tr>
<td><code>x in not s</code></td>
<td>False if an item of <code>s</code> is equal to <code>x</code>, else True</td>
<td>(3)</td>
</tr>
<tr>
<td><code>s1 + s2</code></td>
<td>the concatenation of <code>s1</code> and <code>s2</code></td>
<td></td>
</tr>
<tr>
<td><code>x * n, n*s</code></td>
<td><code>n</code> copies of <code>x</code> concatenated</td>
<td>(1)</td>
</tr>
<tr>
<td><code>s[i]</code></td>
<td><code>i</code>th item of <code>s</code>, origin o</td>
<td></td>
</tr>
<tr>
<td><code>s[i:j]</code></td>
<td>Slice of <code>s</code> from <code>i</code>(included) to <code>j</code> (excluded).</td>
<td>(1), (2)</td>
</tr>
<tr>
<td><code>s.count(x)</code></td>
<td>returns number of <code>x</code>s for which <code>s[i] == x</code></td>
<td></td>
</tr>
<tr>
<td><code>s.index(x[, start[, stop]])</code></td>
<td>returns smallest <code>i</code> such that <code>s[i] == x. start and stop</code> limit search to only part of the sequence.</td>
<td>(4)</td>
</tr>
<tr>
<td><code>len(s)</code></td>
<td>Length of <code>s</code></td>
<td></td>
</tr>
<tr>
<td><code>min(s)</code></td>
<td>Smallest item of <code>s</code></td>
<td></td>
</tr>
<tr>
<td><code>max(s)</code></td>
<td>Largest item of <code>s</code></td>
<td></td>
</tr>
<tr>
<td><code>reversed(s)</code></td>
<td></td>
<td>(2.4)</td>
</tr>
<tr>
<td><code>sorted(Iterable[, cmp])</code></td>
<td>[2.4] Returns an iterator on <code>s</code> in reverse order. <code>s</code> must be a sequence, not an iterator (use <code>reversed(list(s))</code> in this case. [PEP 312]</td>
<td></td>
</tr>
<tr>
<td><code>cmp</code>, <code>key=getKey()</code></td>
<td>[2.4] works like the new in-place list.sort(), but sorts a new list created from the iterable.</td>
<td></td>
</tr>
<tr>
<td><code>start</code>, <code>stop</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>s.sort(key=cmpFunc)</code></td>
<td>[2.4]</td>
<td></td>
</tr>
<tr>
<td><code>reverse=bool</code></td>
<td></td>
<td>(2), (3)</td>
</tr>
</tbody>
</table>

**Notes:**
- (1) If `i` or `j` is negative, the index is relative to the end of the string, i.e. `len(s)+i` or `len(s)+j` is substituted. But note that `-o` is still 0.
- (2) The slice of `s` from `i` to `j` is defined as the sequence of items with index `k` such that `i <= k < j`.
- (3) For strings: `x in s` is True if `x` is a substring of `s`.
- (4) Raises a `ValueError` exception when `x` is not found in `s` (i.e. out of range).

## Operations on mutable sequences (type `list`)

**Operations on mutable sequences**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>s[i] = x</code></td>
<td>item of <code>s</code> is replaced by <code>x</code></td>
<td></td>
</tr>
<tr>
<td><code>s[i:][step] = t</code></td>
<td>slice of <code>s</code> from <code>i</code> to <code>j</code> is replaced by <code>t</code></td>
<td></td>
</tr>
<tr>
<td><code>del s[e:j][step]</code></td>
<td>same as <code>s[e:j] = []</code></td>
<td>(6)</td>
</tr>
<tr>
<td><code>s.append(x)</code></td>
<td>same as <code>s[len(s):] = [x]</code></td>
<td>(6)</td>
</tr>
<tr>
<td><code>s.extend(x)</code></td>
<td>same as <code>s[len(s):len(s)] = x</code></td>
<td>(6)</td>
</tr>
<tr>
<td><code>s.count(x)</code></td>
<td>returns number of <code>x</code>s for which <code>s[i] == x</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>s.index(x[, start[, stop]])</code></td>
<td>returns smallest <code>i</code> such that <code>s[i] == x. start and stop</code> limit search to only part of the list.</td>
<td>(1)</td>
</tr>
<tr>
<td><code>s.insert(i, x)</code></td>
<td>same as <code>s[e:] = [x]</code> if <code>i == 0</code>, <code>i == -1</code> inserts <code>before</code> the last element.</td>
<td></td>
</tr>
<tr>
<td><code>s.remove(x)</code></td>
<td>same as <code>del s[index(x)]</code></td>
<td>(1)</td>
</tr>
<tr>
<td><code>s.pop(i)</code></td>
<td>same as <code>s = s[i]; del s[i]; return x</code></td>
<td>(4)</td>
</tr>
<tr>
<td><code>s.reverse()</code></td>
<td>reverses the items of <code>s</code> in place</td>
<td>(3)</td>
</tr>
<tr>
<td><code>s.sort(key=cmpFunc)</code></td>
<td>[2.4]</td>
<td>(2), (3)</td>
</tr>
<tr>
<td><code>reverse=bool</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notes:**
- (1) Raises a `ValueError` exception when `x` is not found in `s` (i.e. out of range).
- (2) The `sort()` method takes an optional argument `cmp` specifying a comparison function taking 2 list items and returning `-1, 0, or 1` depending on whether the 1st argument is considered smaller than, equal to, or larger than the 2nd argument. Note that this slows the sorting process down considerably. Since 2.4, 2 optional keywords `args` are added: `key` is a function of one argument that used to extract a comparison key from each list element (faster than `cmp`). Also, see `attrgetter` and `itemgetter` in the `operator` module. `reverse`: If True, reverse the sense of the comparison used. Since Python 2.3, the sort is guaranteed “stable”: This means that two entries with equal keys will be returned in the same order as they were input. For example, you can sort a list of people by name, and then sort the list by age, resulting in a list sorted by age where people with the same age are in name-sorted order.
- (3) The `sort` and `reverse` methods modify the list in place for economy of space when sorting or reversing a large list. They don't return the sorted or reversed list to remind you of this side effect.
- (4) The `pop()` method is not supported by mutable sequence types other than lists. The optional argument `i defaults to -1, so that by default the last item is removed and returned.
- (5) Raises a `TypeError` when `x` is not a list object.
- (6) Append vs. `extend`: append takes any object and places as last element in list, while `extend only takes a iterable object and extends the list with each element in x.

## Operations on mappings / dictionaries (type `dict`)

**Operations on mappings**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
</table>

**Notes:**
- (1) Raises a `ValueError` exception when `x` is not found in `s` (i.e. out of range).
- (2) The `sort()` method takes an optional argument `cmp` specifying a comparison function taking 2 list items and returning `-1, 0, or 1` depending on whether the 1st argument is considered smaller than, equal to, or larger than the 2nd argument. Note that this slows the sorting process down considerably. Since 2.4, 2 optional keywords `args` are added: `key` is a function of one argument that used to extract a comparison key from each list element (faster than `cmp`). Also, see `attrgetter` and `itemgetter` in the `operator` module. `reverse`: If True, reverse the sense of the comparison used. Since Python 2.3, the sort is guaranteed “stable”: This means that two entries with equal keys will be returned in the same order as they were input. For example, you can sort a list of people by name, and then sort the list by age, resulting in a list sorted by age where people with the same age are in name-sorted order.
- (3) The `sort` and `reverse` methods modify the list in place for economy of space when sorting or reversing a large list. They don't return the sorted or reversed list to remind you of this side effect.
- (4) The `pop()` method is not supported by mutable sequence types other than lists. The optional argument `i defaults to -1, so that by default the last item is removed and returned.
- (5) Raises a `TypeError` when `x` is not a list object.
- (6) Append vs. `extend`: append takes any object and places as last element in list, while `extend only takes a iterable object and extends the list with each element in x.`
len(d)                     The number of items in d
dict()                    Creates an empty dictionary.
dict(**kwargs)            Creates a dictionary init with the keyword args kwargs.
dict(ite)                 Creates a dictionary init with (key, value) pairs provided by ite.
dict(d)                   Creates a dictionary which is a copy of dictionary d.
d.fromkeys(iterable, value=None)  Class method to create a dictionary with keys provided by iterator, and all values set to value.
d[k]                      The item of d with key k
[d[k] = x]                 Set d[k] to x
del d[k]                  Removes d[k] from d
d.clear()                 Removes all items from d
copy()                    A shallow copy of d
d.has_key(k)              True if d has key k, else False
d in d                    A copy of d's list of items)
d.keys()                  A copy of d's list of keys
1d.update(d2)             for k, v in d2.items(): d[k] = v

d values()                Returns a copy of d's list of values
2d.get(k, defaultVal)     The item of d with key k
3d.setdefault(k, defaultVal)                 d[k] if k in d, else defaultVal (and inserts it)
d.items()                  Returns an iterator over (key, value) pairs.
d.iteritems()              Returns an iterator over the mapping's keys.
d.itervalues()             Returns an iterator over the mapping's values.
d.pop(k, defaultVal)      Removes key k and returns the corresponding value. If key is not found, default is returned if given, otherwise KeyError is raised.
d.popitem()                Removes and returns an arbitrary (key, value) pair from d
d.viewitems()              Returns a view object of the (key, value) pairs
2d.viewkeys()              Returns a view object of the mapping's keys
2d.viewvalues()            Returns a view object of the mapping's values

Notes:
- TypeError is raised if key is not acceptable.
- (1) KeyError is raised if key k is not in the map.
- (2) Keys and values are listed in random order.
- (3) Never raises an exception if k is not in the map, instead it returns defaultVal. defaultVal is optional, when not provided and k is not in the map, None is returned.
- (4) Never raises an exception if k is not in the map, instead returns defaultVal, and adds k to map with value defaultVal. defaultVal is optional. When not provided and k is not in the map, None is returned and added to map.
- (5) A view object provides a dynamic view on the dictionary's entries, which means that when the dictionary changes, the view reflects these changes. A view object is also iterable.

**Operations on strings (types str & unicode)**

These string methods largely (but not completely) supersede the functions available in the string module.
The str and unicode types share a common base class basestring.

**Operations on strings**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>s.capitalize()</td>
<td>Returns a copy of s with its first character capitalized, and the rest of the characters lowercased.</td>
<td>(1)</td>
</tr>
<tr>
<td>s.center(width[, fillChar=’ ’])</td>
<td>Returns a copy of s centered in a string of length width, surrounded by the appropriate number of fillChar characters.</td>
<td>(1)</td>
</tr>
<tr>
<td>s.count(sub[, start[, end]])</td>
<td>Returns the number of occurrences of substring sub in string s.</td>
<td>(2)</td>
</tr>
<tr>
<td>s.decode([encoding[, errors]])</td>
<td>Returns a unicode string representing the decoded version of str s, using the given codec (encoding). Useful when reading from a file or a I/O function that handles only str. Inverse of encode.</td>
<td>(3)</td>
</tr>
<tr>
<td>s.encode([encoding[, errors]])</td>
<td>Returns a str representing an encoded version of s. Mostly used to encode a unicode string to a str in order to print it or write it to a file (since these I/O functions only accept str), e.g. u’légère’.encode(’utf8’). Also used to encode a str to a str, e.g. to zip (codec ’zip’ or uuencode (codec ’uu’) it. Inverse of decode.</td>
<td>(3)</td>
</tr>
<tr>
<td>s.endswith(suffix[, start[, end]])</td>
<td>Returns True if s ends with the specified suffix, otherwise return False. Since 2.5 suffix can also be a tuple of strings to try.</td>
<td>(2)</td>
</tr>
<tr>
<td>s.expandtabs([tabsize])</td>
<td>Returns a copy of s where all tab characters are expanded using spaces.</td>
<td>(4)</td>
</tr>
<tr>
<td>s.find(sub[, start[, end]])</td>
<td>Returns the lowest index in s where substring sub is found. Returns -1 if sub is not found.</td>
<td>(2)</td>
</tr>
<tr>
<td>s.format(*args, **kwargs)</td>
<td>Returns s after replacing numeric and named formatting references found in braces {}. (details)</td>
<td></td>
</tr>
<tr>
<td>s.index(sub[, start[, end]])</td>
<td>like find(), but raises ValueError when the substring is not found.</td>
<td>(2)</td>
</tr>
<tr>
<td>s.isalnum()</td>
<td>Returns True if all characters in s are alphanumeric, False otherwise.</td>
<td>(5)</td>
</tr>
<tr>
<td>s.isalpha()</td>
<td>Returns True if all characters in s are alphabetic, False otherwise.</td>
<td>(5)</td>
</tr>
<tr>
<td>s.isdigit()</td>
<td>Returns True if all characters in s are digit characters, False otherwise.</td>
<td>(5)</td>
</tr>
<tr>
<td>s.islower()</td>
<td>Returns True if all characters in s are lowercase, False otherwise.</td>
<td>(6)</td>
</tr>
<tr>
<td>s.isspace()</td>
<td>Returns True if all characters in s are whitespace characters, False otherwise.</td>
<td>(5)</td>
</tr>
<tr>
<td>s.isupper()</td>
<td>Returns True if string s is a titlecased string, False otherwise.</td>
<td>(7)</td>
</tr>
<tr>
<td>s.join(sep)</td>
<td>Returns a concatenation of the strings in the sequence seq, separated by string separator</td>
<td>(6)</td>
</tr>
</tbody>
</table>
\[s.ljust/rjust/center\]width[, \_fillChar=']\]

\[s.lower()\]

\[s.lstrip\[chars\]\]

\[s.partition\[sep\]\]

\[s.replace\[old, new[, maxCount=-1]\]\]

\[s.rfind\[sub[, start[, end]]\]\]

\[s.rindex\[sub[, start[, end]]\]\]

\[s.rpartition\[sep\]\]

\[s.rstrip\[chars\]\]

\[s.split\[\{separator[, maxsplit]\}\]\]

\[s.splitlines\[\{keepends\}\]\]

\[s.startswith\[prefix[, start[, end]]\]\]

\[s.strip\[chars\]\]

\[s.swapcase()\]

\[s.title()\]

\[s.translate\[\{table[, deletechars=']\}\]\]

\[s.upper()\]

\[s.zfill\[width\]\]

Notes:

- (1) Padding is done using spaces or the given character.
- (2) If optional argument \start is supplied, substring \s[start:] is processed. If optional arguments \start and \end are supplied, substring \s[start:end] is processed.
- (5) Default encoding is sys.getdefaultencoding(). Can be changed via sys.setdefaultencoding(). Optional argument errors may be given to set a different error handling scheme. The default for errors is 'strict', meaning that encoding errors raise a \ValueError. Other possible values are 'ignore' and 'replace'. See also module codecs.
- (4) If optional argument \tabsize is not given, a tab size of 8 characters is assumed.
- (5) Returns False if string \s does not contain at least one character.
- (6) Returns False if string \s does not contain at least one cased character.
- (7) A titlecased string is a string in which uppercase characters may only follow uncased characters and lowercase characters only cased ones.
- (8) \s is returned if \width is less than \len(s).
- (9) If the optional argument \maxCount is given, only the first \maxCount occurrences are replaced.
- (10) If \separator is not specified or None, any whitespace string is a separator. If \maxsplit is given, at most \maxsplit splits are done.
- (11) Line breaks are not included in the resulting list unless \keepends is given and true.
- (12) \table must be a string of length 256.

String formatting with the % operator

\texttt{formatString \% \texttt{args} \rightarrow evaluates to a string}

\texttt{formatString} mixes normal text with C \printf format fields:

\[
\%[flag][width][.precision]formatCode
\]

where \texttt{formatCode} is one of c, s, d, u, o, x, X, e, E, f, g, G, r, \% (see table below).

- The flag characters -, +, blank, \# and 0 are understood (see table below).

- Width and precision may be a * to specify that an integer argument gives the actual width or precision. Examples of \width and \precision:

\begin{center}
\begin{tabular}{|c|c|}
\hline
Format string & Result \\
\hline
\%3d & 2 \ 0 \ 2' \\
\%3d & 3, 2 \ 0 \ 2' \\
\%3d & 2 \ 0 \ 2' \\
\%3d & 2, 2 \ 0 \ 2' \\
\%3d & 2 \ 0 \ 2' \\
\%d & 2 \ 0 \ 2' \\
\%d & 2 \ 0 \ 2' \\
\%d & 2 \ 0 \ 2' \\
\%d & 2 \ 0 \ 2' \\
\%d & 2 \ 0 \ 2' \\
\x & 0 \ 0 \ \ 0 \ \ 0 \ \ 0 \ \ 0 \\
\hline
\end{tabular}
\end{center}
%s will convert any type argument to string (uses str() function)
• args may be a single arg or a tuple of args

```python
a = '%(lang)s has %(c)03d quote types.' % {'c':2, 'lang':'Python'}
```

Right-hand-side can also be a mapping:

```python
a = '%(lang)s has %(c)03d quote types.' % {'c':2, 'lang':'Python'}
```

(derive() function very handy to use on right-hand-side)

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>d</td>
<td>Signed integer decimal.</td>
</tr>
<tr>
<td>i</td>
<td>Signed integer decimal.</td>
</tr>
<tr>
<td>o</td>
<td>Unsigned octal.</td>
</tr>
<tr>
<td>u</td>
<td>Unsigned decimal.</td>
</tr>
<tr>
<td>x</td>
<td>Unsigned hexadecimal (lowercase).</td>
</tr>
<tr>
<td>X</td>
<td>Unsigned hexadecimal (uppercase).</td>
</tr>
<tr>
<td>e</td>
<td>Floating point exponential format (lowercase).</td>
</tr>
<tr>
<td>E</td>
<td>Floating point exponential format (uppercase).</td>
</tr>
<tr>
<td>f</td>
<td>Floating point decimal format.</td>
</tr>
<tr>
<td>F</td>
<td>Floating point decimal format.</td>
</tr>
<tr>
<td>g</td>
<td>Same as “e” if exponent is greater than -4 or less than precision, “f” otherwise.</td>
</tr>
<tr>
<td>c</td>
<td>Single character (accepts integer or single character string).</td>
</tr>
<tr>
<td>r</td>
<td>String (converts any python object using repr()).</td>
</tr>
<tr>
<td>s</td>
<td>String (converts any python object using str()).</td>
</tr>
<tr>
<td>%</td>
<td>No argument is converted, results in a “%” character in the result. (The complete specification is %%).</td>
</tr>
</tbody>
</table>

### Conversion flag characters

<table>
<thead>
<tr>
<th>Flag</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>The value conversion will use the &quot;alternate form&quot;.</td>
</tr>
<tr>
<td>0</td>
<td>The conversion will be zero padded.</td>
</tr>
<tr>
<td>-</td>
<td>The converted value is left adjusted (overrides &quot;).</td>
</tr>
<tr>
<td>+</td>
<td>A sign character (&quot;+&quot; or &quot;) will precede the conversion (overrides a &quot;space&quot; flag).</td>
</tr>
</tbody>
</table>

### String formatting

Since 2.4 [PEP 292] the string module provides a new mechanism to substitute variables into template strings. Variables to be substituted begin with a $. Actual values are provided in a dictionary via the substitute or safe_substitute methods (substitute throws KeyError if a key is missing while safe_substitute ignores it):

```python
t = string.Template('Hello $name, you won $$amount') # (note $$ to literalize $)
t.substitute({'name': 'Eric', 'amount': 100000}) -> u'Hello Eric, you won $100000'
```

### String formatting with format()

Since 2.6 [PEP 3101] string formatting can also be done with the format() method:

```
"string-to-format".format(args)
```

Format fields are specified in string-to-format, surrounded by {}, while actual values are args to format():

```python
{{field}}{{conversion}}{format_spec}
```

• Each field refers to an arg either by its position (>=0), or by its name if it's a keyword argument. If left out, automatic numbering is used, so the first {...} specifier will use the first argument, the next specifier will use the next argument, and so on. Autonumbering cannot be mixed with explicit numbering, but it can be mixed with named fields. The same arg can be referenced more than once.

• The conversion can be % or ! to call str() or repr() on the field before formatting.

• The format_spec takes the following form:

```python
[fill][align][sign][#][0][width][.][precision][type]
```

• The align flag controls the alignment when padding values (see table below), and can be preceded by a fill character. A fill cannot be used on its own.

• The sign flag controls the display of signs on numbers (see table below).

• The # flag adds a leading 0b, 0o, or 0x for binary, octal, and hex conversions.

• The 0 flag zero-pads numbers, equivalent to having a fill-align of 0=.

• The width is a number giving the minimum field width. Padding will be added according to align until this width is achieved.
- The , option indicates that commas should be included in the output as a thousands separator.
- For floating-point conversions, precision gives the number of places to display after the decimal point. For non-
  numeric conversion, precision gives the maximum field width.
- The type specifies how to present numeric types (see tables below).
- Braces can be doubled ({{ or }}) to insert a literal brace character.

Alignment flag characters
Flag Meaning
< Left-aligns the field and pads to the right (default for non-numbers)
> Right-aligns the field and pads to the left (default for numbers)
= Inserts padding between the sign and the field (numbers only)
^ Aligns the field to the center and pads both sides

Sign flag characters
Flag Meaning
+ Displays a sign for all numbers
- Displays a sign for negative numbers only (default)
(a space) Displays a sign for negative numbers and a space for positive numbers

Integer type flags
Flag Meaning
b Binary format (base 2)
c Character (interprets integer as a Unicode code point)
d Decimal format (base 10) (default)
o Octal format (base 8)
x Hexadecimal format (base 16) (lowercase)
X Hexadecimal format (base 16) (uppercase)

Floating-point type flags
Flag Meaning
e Exponential format (lowercase)
E Exponential format (uppercase)
F Fixed-point format
G General format - same as "e" if exponent is greater than -4 or less than precision, "f" otherwise. (default)
N Number format - Same as "g", except it uses locale settings for separators.
% Percentage - Multiplies by 100 and displays as "f", followed by a percent sign.

For examples, see Format examples in the Python documentation.

Operations on files (type file)
(Type file). Created with built-in functions open() [preferred] or its alias file(). May be created by other modules' functions as well.

Unicode file names are now supported for all functions accepting or returning file names (open, os.listdir, etc...).

Operators on file objects

File operations
Operation Result
f.close() Close file f.
f.fileno() Get fileno (fd) for file f.
f.flush() Flush file's internal buffer.
f.isatty() 1 if file f is connected to a tty-like dev, else 0.
f.next() Returns the next input line of file f, or raises StopIteration when EOF is hit. Files are their own iterators. next is implicitly called by constructs like for line in f: print line.
f.read([size]) Read at most size bytes from file f and return as a string object. If size omitted, read to EOF.
f.readline() Read one entire line from file f. The returned line has a trailing \n, except possibly at EOF. Return " on EOF.
f.readlines() Read until EOF with readline() and return a list of lines read.
f.xreadlines() Return a sequence-like object for reading a file line-by-line without reading the entire file into memory. From 2.2, use rather: for line in f (see below).

for line in f: do something...
for line in f: print line  # using readline

f.seek(offset, whence=0) Set file f's position, like "stdio's fseek()".
  whence == 0 then use absolute indexing.
  whence == 1 then offset relative to current pos.
  whence == 2 then offset relative to file end.

f.tell() Return file f's current position (byte offset).

f.truncate([size]) Truncate f's size. If size is present, f is truncated to (at most) that size, otherwise f is truncated at current position (which remains unchanged).

f.write(str) Write string to file f.
f.writelines(list) Write list of strings to file f. No EOL are added.

File Exceptions
EOFError End-of-file hit when reading (may be raised many times, e.g. if f is a tty).
IOError
Operation on sets (types `set` & `frozenset`)

`set` and `frozenset` (immutable set). Sets are unordered collections of unique (non duplicate) elements. Elements must be hashable. `frozensets` are hashable (thus can be elements of other sets) while `sets` are not. All sets are `iterable`.

A `set` may be created with `set(iterable)` or curly brackets `{}`, which also allows for list comprehensions, using curly brackets instead of square brackets.

Classes `Sets` and `ImmutableSet` in the module `sets` is now deprecated.

**Main Set operations**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>len(s)</code></td>
<td>[using built-in types] Builds a <code>set</code> or <code>frozenset</code> from the given <code>iterable</code> (default: <code>empty</code>), e.g. <code>set([1,2,3]), set(&quot;hello&quot;)</code>.</td>
</tr>
<tr>
<td><code>elt in s / not in s</code></td>
<td>Cardinality of set <code>s</code>. True if element <code>elt</code> belongs / does not belong to set <code>s</code>.</td>
</tr>
<tr>
<td><code>for elt in s: process elt...</code></td>
<td>Iterates on elements of set <code>s</code>.</td>
</tr>
<tr>
<td><code>s.update()</code></td>
<td>True if every element in <code>s</code> is in <code>iterable s2</code>.</td>
</tr>
<tr>
<td><code>s.add(elt)</code></td>
<td>True if every element in <code>s2</code> is in <code>iterable s1</code>.</td>
</tr>
<tr>
<td><code>s.remove(elt)</code></td>
<td>Adds element <code>elt</code> to set <code>s</code> (if it doesn't already exist).</td>
</tr>
<tr>
<td><code>s.pop()</code></td>
<td>Removes element <code>elt</code> from set <code>s</code>. Raises <code>KeyError</code> if element not found.</td>
</tr>
<tr>
<td><code>s.clear()</code></td>
<td>Removes and returns an arbitrary element from set <code>s</code>; raises <code>KeyError</code> if empty.</td>
</tr>
<tr>
<td><code>s.intersection(s2[, s3...]) or s1&amp;s2</code></td>
<td>Removes all elements from this set (not on immutable sets!).</td>
</tr>
<tr>
<td>`s.union(s2[, s3...]) or s1</td>
<td>s2`</td>
</tr>
<tr>
<td><code>s.intersection(s2[, s3...]) or s1-s2</code></td>
<td>Returns a new Set with elements from either <code>set</code> (in the method `s2, s3,... can be any iterable).</td>
</tr>
<tr>
<td><code>s.symmetric_difference(s2) or s1^s2</code></td>
<td>Returns a new Set with elements in <code>s1</code> but not in any of <code>s2, s3... (in the method </code>s2, s3,... can be any iterable).</td>
</tr>
<tr>
<td><code>s.copy()</code></td>
<td>Returns a new Set with elements in either <code>s1 or s2</code> but not both.</td>
</tr>
<tr>
<td><code>s.update(iterable1[ , iterable2...])</code></td>
<td>Returns a shallow copy of set <code>s</code>.</td>
</tr>
</tbody>
</table>

**Named Tuples**

Python 2.6 module `collections` introduces the `namedtuple` datatype. The factory function `namedtuple(typename, fieldnames)` creates subclasses of `tuple` whose fields are accessible by `name` as well as `index`:

```python
# Create a named tuple class 'person':
person = collections.namedtuple('person', 'name firstName age') # field names separated by space or comma
assert issubclass(person, tuple)
assert person._fields == ('name', 'firstName', 'age')

# Create an instance of person:
jdoe = person('Doe', 'John', 30)
assert str(jdoe) == "person(name='Doe', firstName='John', age=30)"
assert jdoe[0] == 'Doe' # access by index or name is equivalent
assert jdoe[1] == 'John'
assert jdoe[2] == 30

# Convert instance to dict:
assert jdoe._asdict() == {'age': 30, 'name': 'Doe', 'firstName': 'John'}

# Although tuples are normally immutable, one can change field values via _replace():
jdoe.replace(35, firstName='Jane')
assert str(jdoe) == "person(name='Doe', firstName='Jane', age=35)"
```

**Date/Time**

Python has **no intrinsic** Date and Time types, but provides 2 built-in modules:

- `time`: time access and conversions
- `datetime`: classes `date`, `time`, `datetime`, `timedelta`, `tzinfo`.
- `calendar`: with functions such as `isleap(year)`, `leapdays(y1, y2)` and `weekday(year, month, day)`. See also the third-party module: `pymDateTime`.

**Advanced Types**

- See manuals for more details -

  - `Module objects`
  - `Class objects`
  - `Class instance` objects
  - `Type objects` (see module: `types`)
  - `File objects` (see above)
  - `Slice objects`
Ellipsis object, used by extended slice notation (unique, named Ellipsis)

* Null object (unique, named None)

* XRange objects

** Callable types:**
- User-defined (written in Python):
  - User-defined Function objects
  - User-defined Method objects
- Built-in (written in C):
  - Built-in Function objects
  - Built-in Method objects

** Internal Types:**
- Code objects (byte-compile executable Python code: bytecode)
- Frame objects (execution frames)
- Traceback objects (stack trace of an exception)

---

### Statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>pass</td>
<td>Null statement</td>
</tr>
<tr>
<td>del name[, name]*</td>
<td>Unbind name(s) from object. Object will be indirectly (and automatically) deleted only if no longer referenced.</td>
</tr>
<tr>
<td>print([^fileobject,] [s1 [, s2 ]]* [,])</td>
<td>Writes to sys.stdout, or to fileobject if supplied. Puts spaces between arguments. Puts newline at end unless statement ends with comma [if nothing is printed when using a comma, try calling sys.stdout.flush()]. Print is not required when running interactively, simply typing an expression will print its value, unless the value is None.</td>
</tr>
<tr>
<td>exec(in globals [, , locals])</td>
<td>Executes x in namespaces provided. Defaults to current namespaces. x can be a string, open file-like object or a function object. locals can be any mapping type, not only a regular Python dict. See also built-in function execfile.</td>
</tr>
</tbody>
</table>

**callable(value,... [id=value] ,*args, [**kw])**

- Call function callable with parameters. Parameters can be passed by name or be omitted if function defines default values. E.g. if callable is defined as "def callable(p1=1, p2=2)"

  "callable()" <=> "callable(1, 2)"

  "callable(10)" <=> "callable(10, 2)"

  "callable(p2=99)" <=> "callable(1, 99)"

*args is a tuple of positional arguments. 
**kw is a dictionary of keyword arguments. 
See function definition.

---

### Assignment operators

<table>
<thead>
<tr>
<th>Assignment operators</th>
<th>Operator</th>
<th>Result</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>a = b</td>
<td>Basic assignment - assign object b to label a</td>
<td>(1)(2)</td>
<td></td>
</tr>
<tr>
<td>a += b</td>
<td>Roughly equivalent to a = a + b</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>a *= b</td>
<td>Roughly equivalent to a = a * b</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>a /= b</td>
<td>Roughly equivalent to a = a / b</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>a //= b</td>
<td>Roughly equivalent to a = a = b</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>a %= b</td>
<td>Roughly equivalent to a = a % b</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>a **= b</td>
<td>Roughly equivalent to a = a ** b</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>a &amp;= b</td>
<td>Roughly equivalent to a = a &amp; b</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>= b</td>
<td>Roughly equivalent to a = a</td>
<td>b</td>
</tr>
<tr>
<td>a ^= b</td>
<td>Roughly equivalent to a = a ^ b</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>a &gt;&gt;= b</td>
<td>Roughly equivalent to a = a &gt;&gt; b</td>
<td>(3)</td>
<td></td>
</tr>
<tr>
<td>a &lt;&lt;= b</td>
<td>Roughly equivalent to a = a &lt;&lt; b</td>
<td>(3)</td>
<td></td>
</tr>
</tbody>
</table>

Notes:
- (1) Can unpack tuples, lists, and strings:

  ```
  first, second = l[0:2]  # equivalent to: first=l[0]; second=l[1]
  [f, s] = range(2)       # equivalent to: f=0; s=1
  c1, c2, c3 = 'abc'     # equivalent to: c1='a'; c2='b'; c3='c'
  (a, b), c, (d, e, f) = ('ab', 'c', 'def')  # equivalent to: a='a'; b='b'; c='c'; d='d'; e='e'; f='f'
  ```

  Tip: x, y = y, x swaps x and y.

- (2) Multiple assignment possible:

  ```
  a = b = c = 0
  list1 = list2 = [1, 2, 3]  # list1 and list2 points to the same list (l1 is l2)
  ```

- (3) Not exactly equivalent - a is evaluated only once. Also, where possible, operation performed in-place - a is modified rather than replaced.

---

### Conditional Expressions
Conditional Expressions (not statements) have been added since 2.5 [PEP 308]:

```python
result = (whenTrue if condition else whenFalse)
```

is equivalent to:

```python
if condition:
    result = whenTrue
else:
    result = whenFalse
```

() are not mandatory but recommended.

## Control Flow statements

### Control flow statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>if condition:</strong></td>
<td>Usual if/else if/else statement. See also Conditional Expressions for one-line if-statements.</td>
</tr>
<tr>
<td><strong>suite</strong></td>
<td></td>
</tr>
<tr>
<td><strong>[else: suite]</strong></td>
<td></td>
</tr>
<tr>
<td><strong>while condition:</strong></td>
<td>Usual while statement. The <strong>else suite</strong> is executed after loop exits, unless the loop is exited with <strong>break.</strong></td>
</tr>
<tr>
<td><strong>suite</strong></td>
<td></td>
</tr>
<tr>
<td><strong>[else: suite]</strong></td>
<td></td>
</tr>
<tr>
<td><strong>for element in sequence:</strong></td>
<td>Iterates over <strong>sequence</strong>, assigning each element to <strong>element</strong>. Use built-in <strong>range</strong> or <strong>xrange</strong> function to iterate a number of times. The <strong>else suite</strong> is executed at end unless loop exited with <strong>break.</strong></td>
</tr>
<tr>
<td><strong>suite</strong></td>
<td></td>
</tr>
<tr>
<td><strong>[else: suite]</strong></td>
<td></td>
</tr>
<tr>
<td><strong>break</strong></td>
<td>Immediately exits <strong>for</strong> or <strong>while</strong> loop.</td>
</tr>
<tr>
<td><strong>continue</strong></td>
<td>Immediately does next iteration of <strong>for</strong> or <strong>while</strong> loop.</td>
</tr>
<tr>
<td><strong>return [result]</strong></td>
<td>Exits from function (or method) and returns <strong>result</strong> (use a <strong>tuple</strong> to return more than one value). If no result given, then returns <strong>None.</strong></td>
</tr>
<tr>
<td><strong>yield expression</strong></td>
<td>Only used within the body of a generator function, outside a try of a <strong>try..finally</strong>. &quot;Returns&quot; the evaluated expression.</td>
</tr>
</tbody>
</table>

## Exception statements

### Exception statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>assert exp[, message]</strong></td>
<td><code>exp</code> is evaluated. If false, raises exception <code>AssertionError</code> with <code>message</code>. Before 2.3, inhibited if <code>__debug__</code> is <code>0</code>. Statements in <code>block1</code> are executed. If an exception occurs, look in <code>except</code> clause(s) for matching <code>exception(s)</code>. If matches or bare <code>except</code>, execute <code>handler</code> of that clause. If no exception happens, <code>else-block</code> in <code>else clause</code> is executed after <code>block1</code>. If <code>exception</code> has a value, it is put in variable <code>value</code>. <code>exception</code> can also be a <code>tuple</code> of exceptions, e.g. <code>except (KeyError, NameError), e: print e</code>. 2.6 also supports the keyword <code>as</code> instead of a comma between the <code>exception</code> and the <code>value</code>, which will become a mandatory change in Python 3.0 [PEP3110]. Statements in <code>block1</code> are executed. If no exception, execute <code>final-block</code> (even if <code>block1</code> is exited with a <code>return</code>, <code>break</code> or <code>continue</code> statement). If exception did occur, execute <code>final-block</code> and then immediately re-raise exception. Typically used to ensure that a resource (file, lock...) allocated before the <code>try</code> is freed (in the <code>final-block</code>) whatever the outcome of <code>block1</code> execution. See also the with statement below. Unified <code>try/except/finally</code> Equivalent to a <code>try..except</code> nested inside a <code>try..finally</code> [PEP341]. See also the with statement below.</td>
</tr>
<tr>
<td><strong>try:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>block1</strong></td>
<td></td>
</tr>
<tr>
<td><strong>finally:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>final-block</strong></td>
<td></td>
</tr>
<tr>
<td><strong>try:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>block1</strong></td>
<td></td>
</tr>
<tr>
<td><strong>finally:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>with allocate-expression [as variable]]:</strong></td>
<td><em>Alternative to the <code>try...finally</code> structure [PEP343]. Attempted <code>allocate-expression</code> should evaluate to an object that supports the context management protocol, representing a resource. This object may return a value that can optionally be bound to <code>variable</code> (variable is not assigned the result of expression). The object can then run <code>setup code before with-block is executed and some </code>clean-up code is executed after the block is done, even if the block raised an exception. `Standard Python objects such as files and locks support the context management protocol:</em></td>
</tr>
<tr>
<td><strong>with-block</strong></td>
<td></td>
</tr>
<tr>
<td><strong>with allocate-expression as variable [, allocate-expression as variable2 as variable2]:</strong></td>
<td></td>
</tr>
<tr>
<td><strong>with-block</strong></td>
<td></td>
</tr>
</tbody>
</table>
print one

with threading.Lock(): # lock automatically released on block exit
do something...

- You can write your own context managers.
- Helper functions are available in module contextlib.

In 2.5 the statement must be enabled by: from __future__ import
with_statement. The statement is always enabled starting in Python 2.6.

raise exceptionInstance
raise exceptionClass [, value [, 
traceback]]

Raising an instance of a class derived from BaseException (preferred form of raise).
Raises exception of given class exceptionClass with optional value value. Arg traceback
specifies a traceback object to use when printing the exception’s backtrace.
A raise statement without arguments re-raises the last exception raised in the current
function.

* An exception is an instance of an exception class.
* Exception classes must be derived from the predefined class: Exception, e.g.:

class TextException(Exception): pass
try:
    if bad:
        raise TextException()
except Exception:
    print 'Oops' # This will be printed because TextException is a subclass of Exception

* When an error message is printed for an unhandled exception, the class name is printed, then a colon and a space, and
finally the instance converted to a string using the built-in function str().
* All built-in exception classes derives from StandardError, itself derived from Exception.
* [PEP 352]: Exceptions can now be new-style classes, and all built-in ones are. Built-in exception hierarchy slightly
reorganized with the introduction of base class BaseException. Raising strings as exceptions is now deprecated (warning).

Name Space Statements

Imported module files must be located in a directory listed in the Python path (sys.path). Since 2.3, they may reside in a zip
file [e.g. sys.path.insert(0, 'aZipFile.zip')].

Absolute/relative imports (since 2.5 (PEP328)):

* Feature must be enabled by: from __future__ import absolute_import; will probably be adopted in 2.7.
* Imports are normally relative: modules are searched first in the current directory/package, and then in builtn modules,
resulting in possible ambiguities (e.g. masking a builtin symbol).
* When the new feature is enabled:
  * import X will look up for module X in sys.path first (absolute import).
  * import .X (with a dot) will still search for X in the current package first, then in builtns (relative import).
  * import ..X will search for X in the package containing the current one, etc...

Packages (>1.5): a package is a name space which maps to a directory including module(s) and the special initialization
module __init__.py (possibly empty).

Packages/directories can be nested. You address a module’s symbol via [package.[package...].module.symbol.
[1.51: On Mac & Windows, the case of module file names must now match the case as used in the import statement]

Name space statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>import module1 [as name1] [, module2]*</td>
<td>Imports modules. Members of module must be referred to by qualifying with [package.]module name, e.g.:</td>
</tr>
<tr>
<td></td>
<td>import sys; print sys.argv</td>
</tr>
<tr>
<td></td>
<td>import package1.subpackage.module</td>
</tr>
<tr>
<td></td>
<td>package1.subpackage.module.foo()</td>
</tr>
<tr>
<td>from module import name1 [as othername1][, name2]*</td>
<td>Imports names from module module in current namespace.</td>
</tr>
<tr>
<td></td>
<td>from sys import argy; print argy</td>
</tr>
<tr>
<td></td>
<td>from package1.module import module; module.foo()</td>
</tr>
<tr>
<td></td>
<td>from package1.module.module.foo; module.foo()</td>
</tr>
<tr>
<td></td>
<td>name1 renamed as othername1, if supplied.</td>
</tr>
<tr>
<td>[2.4] You can now put parentheses around the list of names in a from module import names statement (PEP 328).</td>
<td></td>
</tr>
<tr>
<td>global name1 [, name2]</td>
<td>Names are from global scope (usually meaning from module) rather than local (usually</td>
</tr>
</tbody>
</table>
meaning only in function).
E.g. in function without global statements, assuming "x" is name that hasn't been used in
function or module so far:
- Try to read from "x" -> NameError
- Try to write to "x" -> creates "x" local to function
If "x" not defined in function, but is in module, then: - Try to read from "x", gets value from module
- Try to write to "x", creates "x" local to function
But note "x[0]=3" starts with search for "x", will use to global "x" if no local "x".

Function Definition

```python
def funcName (**paramList**):
    suite

paramList ::= [param, ... param]
param ::= value | id=value | *id | **id
```

* Args are passed by "call-by-object-reference". This means, that mutable objects can be modified (ie. in parameters),
while immutable are passed by value (ie. in parameters).
* Use return to return (None) from the function, or return value to return value. Use a tuple to return more than one
value, e.g. return 1,2,3
* Keyword arguments arg=value specify a default value (evaluated at function def. time). They can only appear last in the
param list, e.g. foo(x, y=1, z='t').
* Pseudo-arg *args captures a tuple of all remaining non-keyword args passed to the function, e.g. if def foo(x, *args):
... is called foo(1, 2, 3), then args will contain (2,3).
* Pseudo-arg **kwargs captures a dictionary of all extra keyword arguments, e.g. if def foo(x, **kwargs): ... is called
foo(1, y=2, z=3), then kwargs will contain {'y':2, 'z':3}. If def foo(x, *args, **kwargs): ... is called foo(1, 2,
3, y=4, z=5), then args will contain (2, 3), and kwargs will contain {'y':4, 'z':5}
* args and kwargs are conventional names, but other names may be used as well.
* *args and **kwargs can be "forwarded" (individually or together) to another function, e.g.
def f1(x, *args, **kwargs):
    f2(*args, **kwargs)
    Since 2.6, **kwargs can be any mapping, not only a dict.
* See also Anonymous functions (lambdas).

Class Definition

```python
class className ((super_class1[, super_class2]*)):
    suite
```

Creates a class object and assigns it name className.
suite may contain local "defs" of class methods and assignments to class attributes.

Examples:

class MyClass (class1, class2): ...

Creates a class object inheriting from both class1 and class2. Assigns new class object to name MyClass.

class MyClass: ...

Creates a base class object (inheriting from nothing). Assigns new class object to name MyClass. Since 2.5 the equivalent
syntax class MyClass(): ... is allowed.

class MyClass (object): ...

Creates a new-style class (inheriting from object makes a class a new-style class -available since Python 2.2-). Assigns new
class object to name MyClass.

* First arg to class instance methods (operations) is always the target instance object, called 'self' by convention.
* Special static method __new__(cls,...) called when instance is created. 1st arg is a class, others are args to __init__(),
more details here
* Special method __init__() is called when instance is created.
* Special method __del__() called when no more reference to object.
* Create instance by 'calling' class object, possibly with arg (thus instance=apply(aClassObject, args...) creates an
instance!)

Example:

class C (c_parent):
    def __init__(self, name):
        self.name = name
    def print_name(self):
        print "I'm", self.name
    def call_parent(self):
c_parent.print_name(self)

instance = c('tom')
print instance.name
'tom'
instance.print_name()
"I'm tom"

Call parent's super class by accessing parent's method directly and passing `self` explicitly (see `call_parent` in example above). Many other special methods available for implementing arithmetic operators, sequence, mapping indexing, etc...

### Types / classes unification

**Base types** `int, float, str, list, tuple, dict` and `file` now (2.2) behave like classes derived from base class `object`, and may be subclassed:

```python
x = int(2)  # built-in cast function, now a constructor for base type
y = 3  # int(3) (literals are instances of new base types)
print type(x), type(y) # int, int

assert isinstance(x, int) # replaces isinstance(x, types.IntType)

assert issubclass(int, object) # base types derive from base class 'object'.
s = "hello"  # str("hello")
assert isinstance(s, str)

f = 2.3  # float(2.3)
class MyInt(int): pass  # may subclass base types
x,y = MyInt(1), MyInt("2")
print x, y, x+y # => 1,2,3

class MyList(list): pass

l = MyList("hello")
print l # ['h', 'e', 'l', 'l', 'o']
```

**New-style classes** extends `object`. **Old-style classes** don't.

### Documentation Strings

Modules, classes and functions may be documented by placing a string literal by itself as the first statement in the suite. The documentation can be retrieved by getting the `__doc__` attribute from the module, class or function.

**Example:**

```python
class C:
    "A description of C"
    def __init__(self):
        "A description of the constructor"
        # etc.

c.__doc__ = "A description of C".
c.__init__.__doc__ = "A description of the constructor"
```

### Iterators

- **An iterator** enumerates elements of a *collection*. It is an object with a single method `next()` returning the next element or raising `StopIteration`.
- You get an iterator on `obj` via the new built-in function `iter(obj)`, which calls `obj.__class__.__iter__()`.
- A collection may be its own iterator by implementing both `__iter__()` and `next()`.
- Built-in collections (lists, tuples, strings, dict) implement `__iter__()`; dictionaries (maps) enumerate their keys; files enumerate their lines.
- You can build a `list` or `tuple` from an iterator, e.g. `list(an_iterator)`
- Python implicitly uses iterators wherever it has to `loop`:
  - `for elt in collection`
  - `if elt in collection`
  - `when assigning tuples: x,y,z= collection`

### Generators

- **A generator** is a function that retains its state between 2 calls and produces a `new` value at each `invocation`. The values are returned (one at a time) using the keyword `yield`, while `return` or raise `StopIteration()` are used to notify the end of values.
- A typical use is the production of IDs, names, or serial numbers. fancier applications like nanotreads are also possible.
- To `use` a generator: call the `generator function` to get a generator object, then call `generator.next()` to get the next value until `StopIteration` is raised.
- 2.4 introduces `generator expressions` [PEP 289] similar to list comprehensions, except that they create a generator
that will return elements one by one, which is suitable for long sequences:
linkGenerator = (link for link in get_all_links() if not link.followed)
for link in linkGenerator:
    ... process link...

Generator expressions must appear between parentheses.
× [PEP342] Generators before 2.5 could only produce output. Now values can be passed to generators via their method send(value), yield in now an expression returning a value, so val = (yield i) will yield i to the caller, and will
two other new generator methods allow for additional control:
× throw(type, value=None, traceback=None) is used to raise an exception inside the generator (appears as raised
by the yield expression).
× close() raises a new GeneratorExit exception inside the generator to terminate the iteration.
× Since 2.6 Generator objects have a gi_code attribute that refers to the original code object backing the generator.

Example:

def genID(initialValue=0):
    v = initialValue
    while v < initialValue + 1000:
        yield "ID_%05d" % v
        v += 1
    return # or: raise StopIteration()

generator = genID() # Create a generator
for i in range(10): # Generates 10 values
    print generator.next()

Descriptors / Attribute access

× **Descriptors** are objects implementing at least the first three of these 3 methods representing the descriptor protocol:
× _get__(self, obj, type=None) --> value
× _set__(self, obj, value)
× _delete__(self, obj)

Python now transparently uses descriptors to describe and access the attributes and methods of new-style classes (i.e.
derived from object.)

× Built-in descriptors now allow to define:
× **Static methods** use staticmethod(f) to make method f(x) static (unbound), or (recommended) use decorator
  @staticmethod.
× **Class methods**: like a static but takes the Class as 1st argument => Use f = classmethod(f) to make method
  f(TheClass, x) a class method, or (recommended) use decorator @classmethod.
× **Properties**: A property is an instance of the new built-in type property, which implements the descriptor
  protocol for attributes => Use propertyName = property(fget=None, fset=None, fdel=None, doc=None) to define
  a property inside or outside a class. Then access it as propertyName or obj.propertyName.

Since 2.6, the new decorators @prop.getter, @prop.setter, and @prop.deleter add functions to an existing
property:

class C(object):
    #property # (since Python 2.4)
def x(self):
    return self._x

@x.setter
def x(self, value):
    self._x = value

@x.deleter
def x(self):
    del self._x

× **Slots**. New style classes can define a class attribute __slots__ to constrain the list of assignable attribute names,
  to avoid typos (which is normally not detected by Python and leads to the creation of new attributes), e.g.
  __slots__ = ('x', 'y')

  Note: According to recent discussions, the real purpose of slots seems still unclear (optimization?), and their use
  should probably be discouraged.

Decorators for functions, methods & classes

× [PEP318] A decorator D is noted @D on the line preceding the function/method it decorates:

```
@D
def f(): ...
```

and is equivalent to:

```
def f(): ...
    f = D(f)
```

thus, a decorator can be any function returning another function usually applied as a function transformation.
× Several decorators can be applied in cascade:
A decorator is just a function taking the function to be decorated and returns the same function or some new callable thing.

Decorator functions can take arguments:

```python
@A
@B
@C(args)
```

becomes:

```python
def f(): ...
    _deco = C(args)
    f = A(B(_deco(f)))
```

The decorators `@staticmethod` and `@classmethod` replace more elegantly the equivalent declarations:

```python
f = staticmethod(f)
f = classmethod(f)
```

[PEP 3129] Decorators may also be applied to classes:

```python
@D
class C(): ...
```

is equivalent to:

```python
class C(): ...
c = D(c)
```

Some selected decorators

- `@staticmethod` makes a method static (unbound) from an instance.
- `@classmethod` A class method receives the class as implicit first argument, just like an instance method receives the instance.
- `@prop.getter`, `@prop.setter` and `@prop.deleter` Use a function for getting, setting or deleting the property `prop`

Misc

```python
lambda [param_list]: returnedExpr
```

Creates an anonymous function. `returnedExpr` must be an expression, not a statement (e.g., not "if xx:...", "print xxx", etc.) and thus can't contain newlines. Used mostly for `filter()`, `map()`, `reduce()` functions, and GUI callbacks.

List comprehensions

```python
result = [expression for item1 in sequence1 [if condition1]
          [for item2 in sequence2 ... for itemN in sequenceN]
]
```

is equivalent to:

```python
result = []
for item1 in sequence1:
    for item2 in sequence2:
        ...
        for itemN in sequenceN:
            if condition1 and further conditions:
                result.append(expression)
```

List comprehensions for dictionaries and sets

```python
>>> {x: x*x for x in range(6)}
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25} # Dictionary
```

Equivalent to:

```python
>>> dict([(x, x*x) for x in range(6)])
```

Sets:

```python
>>> {'a'*x for x in range(6)}
set('', 'a', 'aa', 'aaa', 'aaaa', 'aaaaa')
```

See also Generator expressions.

Built-In Functions
**Built-In Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>import</strong>(name,</td>
<td>Imports module within the given context (see library reference for more details)</td>
</tr>
<tr>
<td>globals[,locals,[</td>
<td></td>
</tr>
<tr>
<td>from list]])</td>
<td></td>
</tr>
<tr>
<td>abs(x)</td>
<td>Returns the absolute value of the number x.</td>
</tr>
<tr>
<td>all( iterable )</td>
<td>Returns True if bool(x) is True for all values x in the iterable.</td>
</tr>
<tr>
<td>any( iterable )</td>
<td>Returns True if bool(x) is True for any value x in the iterable.</td>
</tr>
<tr>
<td>apply(func, args[,</td>
<td>Calls function with arguments args and optional keyword arguments. Depreciated since 2.3.</td>
</tr>
<tr>
<td>keywords])</td>
<td></td>
</tr>
<tr>
<td>basestring()</td>
<td>Abstract superclass of str and unicode can't be called or instantiated directly, but useful in:</td>
</tr>
<tr>
<td>bin(x)</td>
<td></td>
</tr>
<tr>
<td>bool(x)</td>
<td></td>
</tr>
<tr>
<td>buffer(obj[, offsets[, size]])</td>
<td></td>
</tr>
<tr>
<td>bytearray(iterable)</td>
<td>Constructs a mutable sequence of bytes. This type supports many of the same operations as str and list. The latter form sets the size and initializes to all zero bytes.</td>
</tr>
<tr>
<td>bytestr(array[length])</td>
<td></td>
</tr>
<tr>
<td>bytes(obj)</td>
<td></td>
</tr>
<tr>
<td>callable(x)</td>
<td></td>
</tr>
<tr>
<td>chr(i)</td>
<td>Returns true if x is a character string whose ASCII code is integer i.</td>
</tr>
<tr>
<td>classmethod(function)</td>
<td>Returns a class method for function. A class method receives the class as implicit first argument, just like an instance method receives the instance. To declare a class method, use this idiom:</td>
</tr>
<tr>
<td>class C:</td>
<td></td>
</tr>
<tr>
<td>def f(cls, arg1,</td>
<td></td>
</tr>
<tr>
<td>arg2, ...): ...</td>
<td></td>
</tr>
<tr>
<td>f = classmethod(f)</td>
<td></td>
</tr>
<tr>
<td>cmp(x, y)</td>
<td>Returns negative, 0, positive if x &lt;, ==, &gt; to y respectively.</td>
</tr>
<tr>
<td>coercion(x, y)</td>
<td></td>
</tr>
<tr>
<td>compile(string, filename, kind[, flags[, dont_inherit]])</td>
<td>Compiles string into a code object. filename is used in error message, can be any string. It is usually the file from which the code was read, or e.g. '&lt;string&gt;' if not read from file. kind can be 'eval' if string is a single stmt, or 'single' which prints the output of expression statements that evaluate to something else than None, or be 'exec'. New args 'flags' and 'dont_inherit' concern future statements. Since 2.6 the function accepts 'keyword' arguments as well as positional parameters.</td>
</tr>
<tr>
<td>complex(real, image)</td>
<td>Creates a complex object (can also be done using J or j suffix, e.g. 1+3J). Since 2.6, also accepts strings, with or without parenthesis, e.g. complex('1+3J') or complex('(1+3J)').</td>
</tr>
<tr>
<td>delattr(obj, name)</td>
<td></td>
</tr>
<tr>
<td>dict(mapping-or-sequence)</td>
<td>Returns a new dictionary initialized from the optional argument (or an empty dictionary if no argument). Argument may be a sequence (or anything iterable) of pairs (key, value).</td>
</tr>
<tr>
<td>dir(obj)</td>
<td></td>
</tr>
<tr>
<td>divmod(a, b)</td>
<td>Returns tuple (a/b, a%b)</td>
</tr>
<tr>
<td>enumerate(iterable[, start=0])</td>
<td></td>
</tr>
<tr>
<td>eval(s[, globals[, locals]])</td>
<td>Evaluates string s, representing a single python expression, in (optional) globals, locals contexts. s must have no NUL or newlines. s can also be a code object. Locals can be any mapping type, not only a regular Python dict.</td>
</tr>
<tr>
<td>Example:</td>
<td></td>
</tr>
<tr>
<td>x = 1; assert eval('x + 1') == 2</td>
<td>(To execute statements rather than a single expression, use Python statement exec or built-in function execfile)</td>
</tr>
<tr>
<td>execfile(file[, globals[, locals]])</td>
<td></td>
</tr>
<tr>
<td>exec(filename[, mode[, bufsize]])</td>
<td></td>
</tr>
<tr>
<td>filter(function, sequence)</td>
<td></td>
</tr>
<tr>
<td>float(x)</td>
<td>Converts a number or a string to floating point. Since 2.6, x can be one of the strings 'nan', '+inf', or '-inf' to represent respectively IEEE 754 Not A Number, positive and negative infinity. Use module math functions isnan() and isinf() to check for NaN or infinity.</td>
</tr>
<tr>
<td>format(value[, format_spec])</td>
<td>Formats an object with the given specification (default *) by calling its <strong>format</strong> method.</td>
</tr>
<tr>
<td>frozenset(iterable)</td>
<td>Returns a frozenset (immutable set) object whose (immutable) elements are taken from iterable or empty by default. See also sets</td>
</tr>
</tbody>
</table>
getattr(object, name[, default]))

getattribute called name from object, e.g. getattribute(x, 'Y') == x.Y. If not found, raises
AttributeError or returns default if specified.

globals()

Returns a dictionary containing the current global variables.

hasattr(object, name)

Returns true if object has an attribute called name.

hash(object)

Returns the hash value of the object (if it has one).

help([object])

Invokes the built-in help system. No argument -> interactive help; if object is a string (name of
a module, function, class, method, keyword, or documentation topic), a help page is printed on
the console; otherwise a help page on object is generated.

hex(x)

Converts a number to a hexadecimal string.

id(object)

Returns a unique integer identifier for object. Since 2.5 always returns non-negative numbers.

input([prompt])

Prints prompt if given. Reads input and evaluates it. Uses line editing / history if module
readline available.

For un-evaluated input, see raw_input().

int(x[, base])

Converts a number or a string to a plain integer. Optional base parameter specifies base from
which to convert string values.

 intern(aString)

Enters aString in the table of interned strings and returns the string. Since 2.3, interned
strings are no longer 'immortal' (never garbage collected), see [details]

isinstance(obj, class1[,...])

Returns true if obj is an instance of classclass2 or an object of type classInfo (class2 may
also be a tuple of classes or types). If issubclass(A,B) then isinstance(x,A) =>
issubclass(x,B).

issubclass((class1, class2)

Returns true if class1 is a subclass of class2 (or if class1 is class).

iter(obj, sentinel)

Returns an iterator on obj. If sentinel is absent, obj must be a collection implementing either
__iter__() or __getitem__(). If sentinel is given, obj will be called with no arg; if the value
returned is equal to sentinel, StopIteration will be raised, otherwise the value will be returned.
See Iterators.

len(obj)

Returns the length of the (number of items) of an object (sequence, dictionary, or instance of class
implementing _len_).

list([seq])

Creates an empty list or a list with same elements as seq. seq may be a sequence, a container
that supports iteration, or an iterator object. If seq is already a list, returns a shallow copy of it.

locals()

Returns a dictionary containing current local variables.

long(x[, base])

Converts a number or a string to a long integer. Optional base parameter specifies the base from
which to convert string values.

map(function, sequence[, sequence, ...])

Returns a list of the results of applying function to each item from sequence(s). If more than one
sequence is given, the function is called with an argument list consisting of the corresponding
item of each sequence, substituting None for missing values when not all sequences have the same
length. If function is None, returns a list of the items of the sequence (or a list of tuples if
more than one sequence). => You might also consider using list comprehensions instead of
map().

max(iterable[, key=func])

With a single argument iterable, returns the largest item of a non-empty iterable (such as a
string, tuple or list). With more than one argument, returns the largest of the arguments.

The optional key arg is a function that takes a single argument and is called for every value in the
list.

min(iterable[, key=func])

With a single argument iterable, returns the smallest item of a non-empty iterable (such as a
string, tuple or list). With more than one argument, returns the smallest of the arguments.

The optional key arg is a function that takes a single argument and is called for every value in the
list.

next(iterator[, default])

Returns the next item from iterator. If iterator exhausted, returns default if specified, or raises
StopIteration otherwise.

object()

Returns a new featureless object. object is the base class for all new style classes, its methods are
common to all instances of new style classes.

oct(x)

Converts a number to an octal string.

open(filename[, mode='r', (bfsname)])

Returns a new file object. See also alias file(). Use codecs.open() instead to open an encoded
file and provide transparent encoding / decoding.

* filename is the file name to be opened.
  * mode indicates how the file is to be opened:
    * 'r' for reading
    * 'w' for writing (truncating an existing file)
    * 'a' opens it for appending
    * '+'(applied to any of the previous modes) open the file for updating (note that
      'w+' truncates the file)
    * 'b'(applied to any of the previous modes) open the file in binary mode
    * 'U' (or 'U') open the file for reading in Universal Newline mode: all variants of EOL
      (CR, LF, CR+LF) will be translated to a single LF ('
').
    * bufferize is o for unbuffered, 1 for line buffered, negative or omitted for system default, >1
      for a buffer of (about) the given size.

ord(c)

Returns integer ASCII value of c (a string of len 1). Works with Unicode char.

pow(x, y[, z])

Returns x to power y [modulo z]. See also ** operator.

property([fget[, fset[, fdel[, doc]]]])

Returns a property attribute for new-style classes (classes deriving from object). fget, fset, and
fdel are functions to get the property value, set the property value, and delete the property,
respectively. Typical use:

class C(object):
  def __init__(self): self._x = None
  def getx(self): return self._x
  def setx(self, value): self._x = value
  def delx(self): del self._x

  x = property(getx, setx, delx, "I'm the 'x' property.")

print(*args[, sep=']
[', end='\n'] [, file=sys.stdout])

When __future__.print_function is active, the print statement is replaced by this function
[PEP3105]. Each item in args is printed to file with sep as the delimiter, and finally followed by
Each of these statements:

```python
print 'foo' 42
print 'foo', 42,
print >> sys.stderr 'warning'
```

can now be written in this functional form:

```python
print('foo', 42)
print('foo', 42, end='')
print('warning', file=sys.stderr)
```

```python
range([start], end [, step])
```

Returns list of ints from \( \geq \) start and \(< \) end.

With 1 arg, list from 0..arg-1
With 2 args, list from start..end-1
With 3 args, list from start up to end by step

```python
raw_input([prompt])
```

Prints prompt if given, then reads string from std input (no trailing \n). See also input().

```python
reduce(f, [list, [init]])
```

Applies the binary function \( f \) to the items of list so as to reduce the list to a single value. If init is given, it is "prepending" to list.

```python
reload(module)
```

Re-parses and re-initializes an already imported module. Useful in interactive mode, if you want to reload a module after fixing it. If module was syntactically correct but had an error in initialization, must import it one more time before calling reload().

```python
repr(object)
```

Returns a string containing a printable and if possible \texttt{evaluable} representation of an object.

```python
round(x, n=0)
```

Returns the floating point value \( x \) rounded to \( n \) digits after the decimal point.

```python
set(iterable)
```

Returns a \texttt{set} object whose elements are taken from \texttt{iterable}, or empty by default. See also \texttt{Sets}.

This is the counterpart of \texttt{getattr().setattr(o, 'foo', bar', 3) \texttt{<>} o.foobar = 3}. \texttt{Creates attribute if it doesn't exist!}

```python
slice([start], [stop], [step])
```

Returns a \texttt{slice} object representing a range, with \texttt{R/O} attributes: start, stop, step.

```python
sorted(iterable[, cmp[, key[, reverse]]])
```

Returns a new \texttt{sorted} list from the items in \texttt{iterable}. This contrasts with \texttt{list.sort()} that sorts lists in \texttt{place} and doesn't apply to immutable sequences like strings or tuples. See \texttt{sequences.sort method.}

```python
staticmethod(function)
```

Returns a static method for \texttt{function}. A static method does not receive an implicit first argument. To declare a static method, use this idiom:

```python
class C:
    def f(arg1, arg2, ...):
        f = 
```

Then call it on the \texttt{class C.f()} or on an instance \texttt{C().f()}. The instance is ignored except for its class.

Since \texttt{2.4} you can alternatively use the decorator notation:

```python
class C:
    @staticmethod
def f(arg1, arg2, ...):
    ...
```

```python
str(object)
```

Returns a string containing a nicely printable representation of an object. Class overridable (_\texttt{str}_). See also \texttt{repr()}. 

```python
sum(iterable[, start=0])
```

Returns the sum of a sequence of numbers (\texttt{not} strings), plus the value of parameter. Returns \texttt{start} when the sequence is empty.

```python
super(type[, object-or-type])
```

Returns the superclass of \texttt{type}. If the second argument is omitted the super object returned is unbound. If the second argument is an object, \texttt{isinstance(obj, type)} must be true. If the second argument is a type, \texttt{issubclass(type2, type)} must be true. Typical use:

```python
class C(B):
    def meth(self, arg):
        super(C, self).meth(arg)
```

```python
tuple(seq)
```

Creates an empty tuple or a tuple with same elements as \texttt{seq}. \texttt{seq} may be a sequence, a container that supports iteration, or an iterator object. If \texttt{seq} is already a tuple, returns \texttt{itself} (not a copy).

```python
type(obj)
```

Returns a \texttt{type} object [see module \texttt{types}] representing the type of \texttt{obj}. \texttt{Examples}: \texttt{import types} if \texttt{type(x) == types.StringType: print 'It is a string'.} \texttt{X: it is better to use instead: if isinstance(x, x.StringType)...}

```python
unicode(str[, encoding[, error]])
```

Creates a \texttt{Unicode} string from a \texttt{8-bit} string, using the given encoding name and error \texttt{treatment ('strict', 'ignore', or 'replace'). For objects which provide a \_\texttt{unicode}\_() method, it will call this method without arguments to create a \texttt{Unicode} string.}

```python
vars(object)
```

Without arguments, returns a dictionary corresponding to the current local symbol table. With a module, class or class instance object as argument, returns a dictionary corresponding to the object's symbol table. Useful with the \%s\ string formatting operator.

```python
xrange(start [, end [, step]])
```

Like \texttt{range()}, but doesn't actually store entire list all at once. Good to use in "for" loops where there is a big range and little memory.

```python
zip(seq[, seq2,...])
```

No, that's not a compression tool! For that, see module \texttt{zipfile}] Returns a list of tuples where each tuple contains the \( n \)th element of each of the argument sequences. Since \texttt{2.4} returns an empty list if called with no arguments (was raising \texttt{TypeError} before).

## Built-In Exception classes

### BaseException

Mother of all exceptions (\texttt{was} \texttt{Exception before 2.5}). New-style class. \texttt{exception.args} is a tuple of the arguments passed to the constructor. Since \texttt{2.6} the \texttt{exception.message} attribute is deprecated.

```
KeyboardInterrupt & SystemExit were moved out of \texttt{Exception} because they don't really represent errors. so now a
```
try: ... except Exception: will only catch errors, while a try: ... except BaseException: (or simply try: ... except:) will still catch everything.

- **GeneratorExit**
  Raised by the close() method of generators to terminate the iteration. Before 2.6 was derived from Exception.

- **KeyboardInterrupt**
  On user entry of the interrupt key (often `CTRL-C`). Before 2.5 was derived from Exception.

- **SystemExit**
  On *sys.exit(). Before 2.5 was derived from Exception.

- **Exception**
  Base of all errors. Before 2.5 was the base of all exceptions.
  - **GeneratorExit**
    Moved under BaseException.
  - **StandardError**
    Base class for all built-in exceptions; derived from Exception root class.
    - **ArithmeticError**
      Base class for arithmetic errors.
      - **FloatingPointError**
        When a floating point operation fails.
      - **OverflowError**
        On excessively large arithmetic operation.
      - **ZeroDivisionError**
        On division or modulo operation with 0 as 2nd argument.
    - **AssertionError**
      When an assert statement fails.
    - **AttributeError**
      On attribute reference or assignment failure.
    - **EnvironmentError**
      On error outside Python; error arg. tuple is (errno, err_msg...)
      - **IOError**
        I/O-related operation failure.
      - **OSError**
        Used by the os module’s os.error exception.
    - **WindowsError**
      When a Windows-specific error occurs or when the error number does not correspond to an errno value.
    - **EOFError**
      Immediate end-of-file hit by input() or raw_input()
    - **ImportError**
      On failure of import to find module or name.
    - **KeyboardInterrupt**
      Moved under BaseException.
    - **LookupError**
      base class for IndexError, KeyError
      - **IndexError**
        On out-of-range sequence subscript
      - **KeyError**
        On reference to a non-existent mapping (dict) key
    - **MemoryError**
      On recoverable memory exhaustion
    - **NameError**
      On failure to find a local or global (unqualified) name.
    - **UnboundLocalError**
      On reference to an unassigned local variable.
    - **ReferenceError**
      On attempt to access to a garbage-collected object via a weak reference proxy.
    - **RuntimeError**
      Obsolete catch-all; define a suitable error instead.
      - **NotImplementedError**
        On method not implemented.
    - **SyntaxError**
      On parser encountering a syntax error
      - **IndentationError**
        On parser encountering an indentation syntax error
        - **TabError**
          On improper mixture of spaces and tabs
    - **SystemError**
      On non-fatal interpreter error - bug - report it!
    - **TypeError**
      On passing inappropriate type to built-in operator or function.
    - **ValueError**
      On argument error not covered by TypeError or more precise.
On Unicode-related encoding or decoding error.
- **UnicodeDecodeError**
  On Unicode decoding error.
- **UnicodeEncodeError**
  On Unicode encoding error.
- **UnicodeTranslateError**
  On Unicode translation error.

- **StopIteration**
  Raised by an iterator's `next()` method to signal that there are no further values.
- **SystemExit**
  Moved under `BaseException`.
- **Warning**
  Base class for warnings (see module `warnings`)
  - **DeprecationWarning**
    Warning about deprecated code.
  - **FutureWarning**
    Warning about a construct that will change semantically in the future.
  - **ImportWarning**
    Warning about probable mistake in module import (e.g. missing `__init__.py`).
  - **OverflowWarning**
    Warning about numeric overflow. Won't exist in Python 2.5.
  - **PendingDeprecationWarning**
    Warning about future deprecated code.
  - **RuntimeWarning**
    Warning about dubious runtime behavior.
  - **SyntaxWarning**
    Warning about dubious syntax.
  - **UnicodeWarning**
    When attempting to compare a Unicode string and an 8-bit string that can't be converted to Unicode using default ASCII encoding (raised a `UnicodeDecodeError` before 2.5).
  - **UserWarning**
    Warning generated by user code.

---

**Standard methods & operators redefinition in classes**

Standard methods & operators map to special methods '``method``' and thus can be **redefined** (mostly in user-defined classes), e.g.:

```python
class C:
    def __init__(self, v): self.value = v
    def __add__(self, r): return self.value + r

a = C(3)  # sort of like calling C.__init__(a, 3)
a + 4    # is equivalent to a.__add__(4)
```

**Special methods for any class**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__new__(cls[, ...])</code></td>
<td>Instance creation (on construction). If <code>__new__</code> returns an instance of <code>cls</code> then <code>__init__</code> is called with the rest of the arguments (...), otherwise <code>__init__</code> is not invoked. More details here.</td>
</tr>
<tr>
<td><code>__init__(self, *args)</code></td>
<td>Instance initialization (on construction)</td>
</tr>
<tr>
<td><code>__del__(self)</code></td>
<td>Called on object demise (refcount becomes 0)</td>
</tr>
<tr>
<td><code>__repr__(self)</code></td>
<td><code>repr()</code> and <code>...</code> conversions</td>
</tr>
<tr>
<td><code>__str__(self)</code></td>
<td><code>str()</code> and <code>print()</code> statements</td>
</tr>
<tr>
<td><code>__sizeof__(self)</code></td>
<td>Returns amount of memory used by object, in bytes (called by <code>sys.getsizeof()</code>).</td>
</tr>
<tr>
<td><code>format_(self, format_spec)</code></td>
<td><code>format()</code> and <code>str.format()</code> conversions</td>
</tr>
<tr>
<td><code>__cmp__(self, other)</code></td>
<td>Compares <code>self</code> to <code>other</code> and returns <code>&lt;0, 0, or &gt;0. Implements &gt;, &lt;, == etc...</code></td>
</tr>
<tr>
<td><code>__index__(self)</code></td>
<td><code>[PEP347]</code> Allows using any object as integer index (e.g. for slicing). Must return a single integer or long integer value.</td>
</tr>
<tr>
<td><code>__lt__(self, other)</code></td>
<td>Called for <code>self &lt; other</code> comparisons. Can return anything, or can raise an exception.</td>
</tr>
</tbody>
</table>
| `__le__(self, other)` | Called for `self <= other` comparisons. Can return anything, or can raise an exception. 
| `__gt__(self, other)` | Called for `self > other` comparisons. Can return anything, or can raise an exception.     |
| `__ge__(self, other)` | Called for `self >= other` comparisons. Can return anything, or can raise an exception.    |
| `__eq__(self, other)` | Called for `self == other` comparisons. Can return anything, or can raise an exception.    |
| `__ne__(self, other)` | Called for `self != other` (and `self <> other`) comparisons. Can return anything, or can raise an exception. |
| `__hash__(self)` | Computes 32 bit hash code; `hash()` and dictionary ops. Since 2.5 can also return a `long` integer, in which case the hash of that value will be taken. Since 2.6 can set `__hash__` = None to void class inherited hashability. |
| `__nonzero__(self)` | Returns 0 or 1 for truth value testing. when this method is not defined, `__len__` is called if defined; otherwise all class instances are considered 'true'. |
| `__getattribute__(self, name)` | Called when attribute lookup doesn't find `name`. See also `__getattribute__`. |
| `__getattribute__` | Same as `__getattribute__` but always called whenever the attribute `name` is accessed. |
| `__dir__(self)` | Returns the list of names of valid attributes for the object. Called by built-in function `dir()`, |
but ignored unless __getattr__ or __getattribute__ is defined.

__setattr__(self, name, value)  Called when setting an attribute (inside, don't use 'self.name = value', use instead 'sef._dict__[name] = value')

__delattr__(self, name)  Called to delete attribute <name>.

__call__(self, *args, **kwargs)  Called when an instance is called as function: obj(arg1, arg2, ...) is a shorthand for obj._call__(arg1, arg2, ...).

__enter__(self)  For use with context managers, i.e. when entering the block in a with-statement. The with statement binds this method's return value to the as object.

__exit__(self, type, value, traceback)  When exiting the block of a with-statement. If no errors occurred, type, value, traceback are None. If an error occurred, they will contain information about the class of the exception, the exception object and a traceback object, respectively. If the exception is handled properly, return True. If it returns False, the with-block re-raises the exception.

Operators

See list in the operator module. Operator function names are provided with 2 variants, with or without leading & trailing '__' (e.g. __add__ or add).

**Numeric operations special methods**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Special method</th>
</tr>
</thead>
<tbody>
<tr>
<td>self + other</td>
<td><strong>add</strong>(self, other)</td>
</tr>
<tr>
<td>self - other</td>
<td><strong>sub</strong>(self, other)</td>
</tr>
<tr>
<td>self * other</td>
<td><strong>mul</strong>(self, other)</td>
</tr>
<tr>
<td>self / other</td>
<td><strong>div</strong>(self, other) or <strong>truediv</strong>(self, other) if <strong>future</strong>.division is active.</td>
</tr>
<tr>
<td>self // other</td>
<td><strong>floordiv</strong>(self, other)</td>
</tr>
<tr>
<td>self % other</td>
<td><strong>mod</strong>(self, other)</td>
</tr>
<tr>
<td>divmod(self, other)</td>
<td><strong>divmod</strong>(self, other)</td>
</tr>
<tr>
<td>self ** other</td>
<td><strong>pow</strong>(self, other)</td>
</tr>
<tr>
<td>self &amp; other</td>
<td><strong>and</strong>(self, other)</td>
</tr>
<tr>
<td>self ^ other</td>
<td><strong>xor</strong>(self, other)</td>
</tr>
<tr>
<td>self</td>
<td><strong>or</strong>(self, other)</td>
</tr>
<tr>
<td>self &lt;&lt; other</td>
<td><strong>lshift</strong>(self, other)</td>
</tr>
<tr>
<td>self &gt;&gt;&gt; other</td>
<td><strong>rshift</strong>(self, other)</td>
</tr>
<tr>
<td>bool(self)</td>
<td><strong>nonzero</strong>(self) (used in boolean testing)</td>
</tr>
<tr>
<td>-self</td>
<td><strong>neg</strong>(self)</td>
</tr>
<tr>
<td>+self</td>
<td><strong>pos</strong>(self)</td>
</tr>
<tr>
<td>abs(self)</td>
<td><strong>abs</strong>(self)</td>
</tr>
<tr>
<td>~</td>
<td>self</td>
</tr>
<tr>
<td>self += other</td>
<td><strong>iadd</strong>(self, other)</td>
</tr>
<tr>
<td>self -= other</td>
<td><strong>isub</strong>(self, other)</td>
</tr>
<tr>
<td>self *= other</td>
<td><strong>imul</strong>(self, other)</td>
</tr>
<tr>
<td>self /= other</td>
<td><strong>idiv</strong>(self, other) or <strong>itruediv</strong>(self, other) if <strong>future</strong>.division is in effect.</td>
</tr>
<tr>
<td>self //= other</td>
<td><strong>ifloordiv</strong>(self, other)</td>
</tr>
<tr>
<td>self %= other</td>
<td><strong>imod</strong>(self, other)</td>
</tr>
<tr>
<td>self **= other</td>
<td><strong>ipow</strong>(self, other)</td>
</tr>
<tr>
<td>self &amp;= other</td>
<td><strong>iand</strong>(self, other)</td>
</tr>
<tr>
<td>self ^= other</td>
<td><strong>ixor</strong>(self, other)</td>
</tr>
<tr>
<td>self</td>
<td>= other</td>
</tr>
<tr>
<td>self &lt;&lt;= other</td>
<td><strong>ilshift</strong>(self, other)</td>
</tr>
<tr>
<td>self &gt;&gt;= other</td>
<td><strong>irshift</strong>(self, other)</td>
</tr>
</tbody>
</table>

**Conversions**

<table>
<thead>
<tr>
<th>built-in function</th>
<th>Special method</th>
</tr>
</thead>
<tbody>
<tr>
<td>int(self)</td>
<td><strong>int</strong>(self)</td>
</tr>
<tr>
<td>long(self)</td>
<td><strong>long</strong>(self)</td>
</tr>
<tr>
<td>float(self)</td>
<td><strong>float</strong>(self)</td>
</tr>
<tr>
<td>complex(self)</td>
<td><strong>complex</strong>(self)</td>
</tr>
<tr>
<td>oct(self)</td>
<td><strong>oct</strong>(self)</td>
</tr>
<tr>
<td>hex(self)</td>
<td><strong>hex</strong>(self)</td>
</tr>
<tr>
<td>coerce(self, other)</td>
<td><strong>coerce</strong>(self, other)</td>
</tr>
</tbody>
</table>

Right-hand-side equivalents for all binary operators exist (__radd__, __rsub__, __rmul__, __rdiv__, ...).
They are called when class instance is on r-h-s of operator:

* a + 3 calls __add__(a, 3)
* 3 + a calls __radd__(a, 3)

**Special operations for containers**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Special method</th>
</tr>
</thead>
<tbody>
<tr>
<td>len(self)</td>
<td><strong>len</strong>(self)</td>
</tr>
<tr>
<td>self[k]</td>
<td><strong>getitem</strong>(self, k)</td>
</tr>
<tr>
<td>self[k] = value</td>
<td><strong>setitem</strong>(self, k, value)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>All sequences and maps:</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>self</td>
<td>length of object, &gt;= 0. Length o == False</td>
</tr>
<tr>
<td>Get element at indice /key k (indice starts at 0). Or, if k is a slice object, return a slice.</td>
<td></td>
</tr>
<tr>
<td>Hook called when key is not found in the dictionary, returns the default value.</td>
<td></td>
</tr>
<tr>
<td>Set element at indice/key/slice k.</td>
<td></td>
</tr>
</tbody>
</table>
Special informative state attributes for some types:

Tip: use module inspect to inspect live objects.

Lists & Dictionaries
Attribute Meaning
__methods__ (list, R/O): list of method names of the object Deprecated, use dir() instead

Modules
Attribute Meaning
__doc__ (string/None, R/O): doc string (<> __dict__['__doc__'])
__name__ (string, R/O): module name (also in __dict__['__name__'])
__package__ (string/None, R/W): If defined, package name used for relative imports (also in __dict__['__package__']).

[PEP366]:
__dict__ (dict, R/O): module's name space
__file__ (string/undefined, R/O): pathname of .pyc, .pyo or .pyd (undefined for modules statically linked to the interpreter).
__path__ (list/undefined, R/W): List of directory paths to find the package (for packages only).

Classes
Attribute Meaning
__doc__ (string/None, R/W): doc string (<> __dict__['__doc__'])
__name__ (string, R/W): class name (also in __dict__['__name__'])
__module__ (string, R/W): module name in which the class was defined
__bases__ (tuple, R/W): parent classes
__dict__ (dict, R/W): attributes (class name space)

Instances
Attribute Meaning
__class__ (class, R/W): instance's class
__dict__ (dict, R/W): attributes

User-defined functions
Attribute Meaning
__doc__ (string/None, R/W): doc string
__name__ (string, R/O): function name
func_doc (R/W): same as __doc__
func_name (R/O, R/W from 2.4): same as __name__
func_defaults (tuple/None, R/W): default args values if any
func_code (code, R/W): code object representing the compiled function body
func_globals (dict, R/O): ref to dictionary of func global variables

User-defined Methods
Attribute Meaning
__doc__ (string/None, R/O): Doc string
__name__ (string, R/O): Method name (same as im_func.__name__)
im_class (class, R/O): Class defining the method (may be a base class)
im_self (instance/None, R/O): Target instance object (None if unbound). Since 2.6 use __self__ instead, will be deprecated in 3.0.
im_self__ (instance/None, R/O): Target instance object (None if unbound).
im_func__ (function, R/O): Function object. Since 2.6 use __func__ instead, will be deprecated in 3.0.

Built-in Functions & methods
Attribute Meaning
__doc__ (string/None, R/O): doc string
__name__ (string, R/O): function name
_self__ (methods only) target object
__members__ (list of attr names: (__doc__, __name__, __self__)) Deprecated, use dir() instead.
Codes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>co_name</td>
<td>(string, R/O): function name</td>
</tr>
<tr>
<td>co_argcount</td>
<td>(int, R/o): number of positional args</td>
</tr>
<tr>
<td>co_nlocals</td>
<td>(int, R/O): number of local vars (including args)</td>
</tr>
<tr>
<td>co_varnames</td>
<td>(tuple, R/O): names of local vars (starting with args)</td>
</tr>
<tr>
<td>co_code</td>
<td>(string, R/O): sequence of bytecode instructions</td>
</tr>
<tr>
<td>co_consts</td>
<td>(tuple, R/O): literals used by the bytecode, 1st one is function doc (or None)</td>
</tr>
<tr>
<td>co_names</td>
<td>(tuple, R/O): names used by the bytecode</td>
</tr>
<tr>
<td>co_filename</td>
<td>(string, R/O): filename from which the code was compiled</td>
</tr>
<tr>
<td>co_firstlineno</td>
<td>(int, R/O): first line number of the function</td>
</tr>
<tr>
<td>co_lineno</td>
<td>(string, R/O): string encoding by code offsets to line numbers.</td>
</tr>
<tr>
<td>co_stacksize</td>
<td>(int, R/O): required stack size (including local vars)</td>
</tr>
<tr>
<td>co_flags</td>
<td>(int, R/O): flags for the interpreter bit 2 set if function uses &quot;*arg&quot; syntax, bit 3 set if function uses &quot;**key words/syntax&quot;</td>
</tr>
</tbody>
</table>

Frames

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>f_back</td>
<td>(frame/None, R/O): previous stack frame (towards the caller)</td>
</tr>
<tr>
<td>f_code</td>
<td>(code, R/O): code object being executed in this frame</td>
</tr>
<tr>
<td>f_locals</td>
<td>(dict, R/O): local vars</td>
</tr>
<tr>
<td>f_globals</td>
<td>(dict, R/O): global vars</td>
</tr>
<tr>
<td>f_builtin</td>
<td>(dict, R/O): built-in (intrinsic) names</td>
</tr>
<tr>
<td>f_restricted</td>
<td>(int, R/O): flag indicating whether function is executed in restricted mode</td>
</tr>
<tr>
<td>f_lineno</td>
<td>(int, R/O): current line number</td>
</tr>
<tr>
<td>f_lasti</td>
<td>(int, R/O): precise instruction (index into bycode)</td>
</tr>
<tr>
<td>f_trace</td>
<td>(function/None, R/W): debug hook called at start of each source line</td>
</tr>
<tr>
<td>f_exc_type</td>
<td>(Type/None, R/W): Most recent exception type</td>
</tr>
<tr>
<td>f_exc_value</td>
<td>(any, R/W): Most recent exception value</td>
</tr>
<tr>
<td>f_exc_traceback</td>
<td>(traceback/None, R/W): Most recent exception traceback</td>
</tr>
</tbody>
</table>

Tracebacks

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>tb_next</td>
<td>(frame/None, R/O): next level in stack trace (towards the frame where the exception occurred)</td>
</tr>
<tr>
<td>tb_frame</td>
<td>(frame, R/O): execution frame of the current level</td>
</tr>
<tr>
<td>tb_lineno</td>
<td>(int, R/O): line number where the exception occurred</td>
</tr>
<tr>
<td>tb_lasti</td>
<td>(int, R/O): precise instruction (index into bycode)</td>
</tr>
</tbody>
</table>

Slices

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>(any /None, R/O): lowerbound, included</td>
</tr>
<tr>
<td>stop</td>
<td>(any /None, R/O): upperbound, excluded</td>
</tr>
<tr>
<td>step</td>
<td>(any /None, R/O): step value</td>
</tr>
</tbody>
</table>

Complex numbers

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>(float, R/O): real part</td>
</tr>
<tr>
<td>imag</td>
<td>(float, R/O): imaginary part</td>
</tr>
</tbody>
</table>

xranges

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>tolist</td>
<td>(Built-in method, R/O): ?</td>
</tr>
</tbody>
</table>

Important Modules

sys

System-specific parameters and functions.

Some sys variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>argv</td>
<td>The list of command line arguments passed to a Python script. <code>sys.argv[0]</code> is the script name.</td>
</tr>
<tr>
<td>builtin_module_names</td>
<td>A list of strings giving the names of all modules written in C that are linked into this interpreter.</td>
</tr>
<tr>
<td>byteorder</td>
<td>Native byte order, either ‘big’(-endian) or ‘little’(-endian).</td>
</tr>
<tr>
<td>copyright</td>
<td>A string containing the copyright pertaining to the Python interpreter.</td>
</tr>
<tr>
<td>dont_write_bytecode</td>
<td>If True, prevents Python from from writing .pyc or .pyo files (same as invocation option -B).</td>
</tr>
<tr>
<td>exec_prefix</td>
<td>Root directory where platform-dependent Python files are installed, e.g. <code>‘C:\Python23’</code> for Windows or <code>/usr</code>.</td>
</tr>
<tr>
<td>prefix</td>
<td>Name of executable binary of the Python interpreter (e.g. <code>‘C:\Python23\python.exe’</code>, <code>‘/usr/bin/python’</code>).</td>
</tr>
<tr>
<td>exitstatus</td>
<td>User can set this parameterless function. It will get called before interpreter exits. Deprecated since 2.4.</td>
</tr>
<tr>
<td>flags</td>
<td>Status of command line flags, as a R/O struct. [details]</td>
</tr>
<tr>
<td>float_info</td>
<td>A struct <code>seq</code> holding information about the float type (precision, internal representation, etc...).</td>
</tr>
</tbody>
</table>
last_type, last_value, last_traceback
maxint
maxunicode
modules
path
platform
ps1, ps2
stdin, stdout, stderr
subversion
version
version_info
winver

Some sys functions

Function
_result
_current_frames()
display
hook
doctest
except
hook
exit(n)
getcheckinterval() /
setcheckinterval(interval)
gelseif(objec
t, default)
setprofile(func)
exc_info()
setdefaultencoding(encoding)
getrecursionlimit()
setrecursionlimit()

Result
Returns the current stack frames for all running threads, as a
dictionary mapping thread identifiers to the topmost stack
frame currently active in that thread at the time the function
is called.
The function used to display the output of commands issued in
interactive mode - defaults to the built-in repr(). __displayhook__ is
the original value.
Can be set to a user defined function, to which any uncaught
exceptions are passed. __excepthook__ is the original value.
Exits with status n (usually 0 means OK). Raises SystemExit
exception (hence can be caught and ignored by program).
Gets / Sets the interpreter's thread switching interval (in number
of bytecode instructions, default: 10 until 2.2, 100 from 2.3).
Returns the reference count of the object. Generally 1 higher
than you might expect, because of object arg temp reference.
Returns the amount of memory used by object, in bytes. Calls
o.__sizeof__() if available. default returned if size can't be
determined. [details]
Sets a trace function: called before each line of code is exited.
Sets a profile function for performance profiling.
Info on exception currently being handled; this is a tuple
(exc_type, exc_value, exc_traceback). Warning: assigning the
traceback return value to a local variable in a function handling
an exception will cause a circular reference.
Change default Unicode encoding - defaults to 7-bit ASCII
Retrieve maximum recursion depth.
Set maximum recursion depth (default 1000).

os

Miscellaneous operating system interfaces. Many functions, see the for a comprehensive list!

'synonym' for whatever OS-specific module (nt, mac, posix...) is proper for current environment. This module uses posix
whenever possible. See also M.A. Lemburg's utility platform.py (now included in 2.3+).

Some os variables

Variable
name
path
curdir
sep
pardir
sep
altsep
pathsep
linesep

Meaning
name of O/S-specific module (e.g. "posix", "mac", "nt")
O/S-specific module for path manipulations.
On Unix, os.path.split() <= > posixpath.split()
string used to represent current directory (eg ':')
string used to represent parent directory (eg ':')
string used to separate directories ('/' or '\'). Tip: Use os.path.join() to build portable paths.
Alternate separator if applicable (None otherwise)
character used to separate search path components (as in $PATH), eg. ':' for windows.
line separator as used in text files, ie \n on Unix, \r\n on Dos/Win, \r' on Mac.

Some os functions

Function
makedir(path, mode=0777))
removendir(path)
renames(old, new)
urandom(n)

Result
Recursive directory creation (create required intermediary dirs); os.error if fails.
Recursive directory delete (delete intermediary empty dirs); fails (os.error) if the
directories are not empty.
Recursive directory or file renaming; os.error if fails.
Returns a string containing n bytes of random data.
Do not import this module directly, import os instead ! (see also module: shutil for file copy & remove functions)

**posix Variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
</table>
| environ | dictionary of environment variables, e.g. `posix.environ['HOME']`.
| error   | exception raised on POSIX-related error. |
|         | Corresponding value is tuple of errno code and `errno()` string. |

**Some posix functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>access(path, mode)</td>
<td>Returns True if the requested access to path is granted. Use mode=F_OK to check for existence, or an OR-ed combination of R_OK, W_OK, and X_OK to check for r, w, x permissions.</td>
</tr>
<tr>
<td>chdir(path)</td>
<td>Changes current directory to path.</td>
</tr>
<tr>
<td>chmod(path, mode)</td>
<td>Changes the mode of path to the numeric mode</td>
</tr>
<tr>
<td>close(fd)</td>
<td>Closes file descriptor fd opened with posix.open.</td>
</tr>
<tr>
<td>exe(p, args)</td>
<td>&quot;Become&quot; executable p with args args</td>
</tr>
<tr>
<td>getcwd()</td>
<td>Returns a string representing the current working directory.</td>
</tr>
<tr>
<td>getcwdu()</td>
<td>Returns a Unicode string representing the current working directory.</td>
</tr>
<tr>
<td>getpid()</td>
<td>Returns the current process id.</td>
</tr>
<tr>
<td>getsid()</td>
<td>Calls the system call getsid() [Unix].</td>
</tr>
<tr>
<td>fork()</td>
<td>Like C's fork(). Returns 0 to child, child pid to parent [Not on Windows].</td>
</tr>
<tr>
<td>kill(pid, signal)</td>
<td>Like C's kill [Not on Windows].</td>
</tr>
<tr>
<td>listdir(path)</td>
<td>Lists (base)names of entries in directory path, excluding <code>.</code> and <code>.</code>. If path is a Unicode string, so will be the returned strings.</td>
</tr>
<tr>
<td>lseek(fd, pos, how)</td>
<td>Sets current position in file fd to position pos, expressed as an offset relative to beginning of file (how=0), to current position (how=1), or to end of file (how=2). Actual permissions = (mode &amp; ~umask &amp; 0777). To set directly the permissions, use chmod() after dir creation.</td>
</tr>
<tr>
<td>mkdir(path[, mode])</td>
<td>Creates a directory named path with numeric mode (default 0777).</td>
</tr>
<tr>
<td>mkfifo(path[, mode])</td>
<td>Creates a directory named path with numeric mode (default 0666).</td>
</tr>
<tr>
<td>open(file, flags, mode)</td>
<td>Like C's open(). Returns file descriptor. Use file object functions rather than this low level ones.</td>
</tr>
<tr>
<td>pipe()</td>
<td>Creates a pipe. Returns pair of file descriptors (r, w) [Not on Windows].</td>
</tr>
<tr>
<td>popen(command, mode='r', buffering=None, file_size=0)</td>
<td>Opens a pipe to or from command. Result is a file object to read or write from, as indicated by mode being 'r' or 'w'. Use it to catch a command output ('r' mode), or to feed it ('w' mode).</td>
</tr>
<tr>
<td>rename(old, new)</td>
<td>Renames/moves the file or directory old to new. [error if target name already exists]</td>
</tr>
<tr>
<td>renameat(old, new, dirfd)</td>
<td>Recursive directory or file renaming function. Works like rename(), except creation of any intermediate directories needed to make the new pathname good is attempted first. After the rename, directories corresponding to rightmost path segments of the old name will be pruned away using removefd(s).</td>
</tr>
<tr>
<td>rmdir(path, dirfd)</td>
<td>Removes the empty directory path</td>
</tr>
<tr>
<td>read(fd, n)</td>
<td>Reads n bytes from file descriptor fd and return as string.</td>
</tr>
<tr>
<td>system(command)</td>
<td>Executes string command in a subshell. Returns exit status of subshell (usually 0 means OK). Since 2.4 use subprocess.call() instead.</td>
</tr>
<tr>
<td>times()</td>
<td>Returns accumulated CPU times in sec (user, system, children's user, children's sys, elapsed real time) [3 last not on Windows].</td>
</tr>
<tr>
<td>unlink(path)</td>
<td>Unlinks &quot;delete&quot; the file (not dir!) path. Same as: remove.</td>
</tr>
<tr>
<td>utime(path, (atime, mtime))</td>
<td>Sets the access &amp; modified time of the file to the given tuple of values.</td>
</tr>
<tr>
<td>wait()</td>
<td>Waits for child process completion. Returns tuple of pid, exit_status [Not on Windows].</td>
</tr>
<tr>
<td>waitpid(pid, options)</td>
<td>Waits for process pid to complete. Returns tuple of pid, exit_status [Not on Windows].</td>
</tr>
<tr>
<td>walk(top[, topdown=True, onerror=None, followlinks=False)])</td>
<td>Generates a list of file names in a directory tree, by walking the tree either top down or bottom up. For each directory in the tree rooted at directory top (including top itself), it yields a 3-tuple (dirpath, dirnames, filenames) - more info here. See also <code>os.path.walk()</code>.</td>
</tr>
<tr>
<td>write(fd, str)</td>
<td>Writes str to file fd. Returns nb of bytes written.</td>
</tr>
</tbody>
</table>

**posixpath**

Do not import this module directly, import os instead and refer to this module as os.path. (e.g. os.path.exists(p))!

**posixpath functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>abspath(path)</td>
<td>Returns absolute path for path, taking current working dir in account.</td>
</tr>
</tbody>
</table>
| commonprefix(list) | Returns the longest path prefix (taken character-by-character) that is a prefix of all paths in list (or "if list empty."
| dirname(base_name) | directory and name parts of path. See also split. |
| exists(path) | True if path is the path of an existing file or directory. See also `exists()`. |
| expanduser(path) | Returns a copy of path with ~"-" expansion done. |
| expandvars(path) | Returns string that is a (copy of) path with environment vars ${NAME} or ${NAME} expanded. [Windows: case significant; must use Unix $VAR notation, not %VAR%: 2.6: Notation %NAME% also supported.] |
| getatime(path) | Returns last access time of path (integer nb of seconds since epoch). |
| getctime(path) | Returns the metadata change time of path (integer nb of seconds since epoch). |
shutil

High-level file operations (copying, deleting).

Main shutil functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>copy(src, dest)</td>
<td>Copies the contents of file src to file dest, retaining file permissions.</td>
</tr>
<tr>
<td>copytree(src, dest[, symlinks=False[, ignore=None]])</td>
<td>Recursively copies an entire directory tree rooted at src into dest (which should not already exist). symlinks is true, links in src are kept as such in dest. 2.6: New ignore callble argument. Will be called with each directory path and a list of the directory’s contents, must return a list of names to ignore. shutil.ignore_patterns() can be used to exclude glob-style patterns, e.g.:</td>
</tr>
<tr>
<td>shutil.copystat('projects/myProjUnderSvn', 'exportDir', ignore=shutil.ignore_patterns('*.svn'))</td>
<td></td>
</tr>
<tr>
<td>move(src, dest)</td>
<td>Recursively moves a file or directory to a new location.</td>
</tr>
<tr>
<td>rmtree(path[, ignore_errors[, onerror]])</td>
<td>Deletes an entire directory tree, ignoring errors if ignore_errors is true, or calling onerror(func, path, exc_info) if supplied, with arguments func (faulty function), and path (concerned file). This function fails when the files are Read Only.</td>
</tr>
<tr>
<td>make_archive(base_name, format[, root_dir[, base_dir[, verbose[, dry_run[, owner[, group[, logger]]]]]]]]]</td>
<td>Create an archive file (eg. zip or tar) and returns its name. base_name is the name of the file to create, including the path, minus any format-specific extension. format is the archive format: one of “zip”, “tar”, “bztar” or “gztar”. root_dir is a directory that will be the root directory of the archive; ie. we typically change into root_dir before creating the archive. base_dir is the directory where we start archiving from; ie. base_dir will be the common prefix of all files and directories in the archive. root_dir and base_dir both default to the current directory. owner and group are used when creating a tar archive. By default, uses the current owner and group. logger is an instance of logging.Logger.</td>
</tr>
</tbody>
</table>

(time and also: copyfile, copymode, copystats, copy2)

time

Time access and conversions. (see also module mdate/time if you need a more sophisticated date/time management)

Variables

Variable Meaning
altzone Signed offset of local DST timezone in sec west of the oth meridian.
daylight Non zero if a DST timezone is specified.
tzname The offset of the local (non-DST) timezone, in seconds west of UTC.

Some functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>clock()</td>
<td>On Unix: current processor time as a floating point number expressed in seconds. On Windows: wall-clock seconds elapsed since the 1st call to this function, as a floating point number (precision &lt; 1ms).</td>
</tr>
<tr>
<td>time()</td>
<td>Returns a float representing UTC time in seconds since the epoch.</td>
</tr>
<tr>
<td>gmtime([secs])</td>
<td>Returns a 9-tuple representing time. Current time is used if secs is not provided.</td>
</tr>
</tbody>
</table>
localetime([secs])

Since 2.2, returns a struct_time object (still accessible as a tuple) with the following attributes:

<table>
<thead>
<tr>
<th>Index</th>
<th>Attribute</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>tm_year</td>
<td>Year (e.g. 1993)</td>
</tr>
<tr>
<td>1</td>
<td>tm_mon</td>
<td>Month [1,12]</td>
</tr>
<tr>
<td>2</td>
<td>tm_mday</td>
<td>Day [1,31]</td>
</tr>
<tr>
<td>3</td>
<td>tm_hour</td>
<td>Hour [0,23]</td>
</tr>
<tr>
<td>4</td>
<td>tm_min</td>
<td>Minute [0,59]</td>
</tr>
<tr>
<td>5</td>
<td>tm_sec</td>
<td>Second [0,61]; The 61 accounts for leap seconds and (the very rare) double leap seconds.</td>
</tr>
<tr>
<td>6</td>
<td>tm_wday</td>
<td>Weekday [0,6], Monday is 0</td>
</tr>
<tr>
<td>7</td>
<td>tm_yday</td>
<td>Julian day [1,366]</td>
</tr>
<tr>
<td>8</td>
<td>tm_isdst</td>
<td>Daylight flag: 0, 1 or -1; -1 passed to mktime() will usually work</td>
</tr>
</tbody>
</table>

astime((timeTuple)),

24-character string of the following form: `Mon Apr 03 08:31:14 2006`; timeTuple defaults to localtime() if omitted.

cftime([secs])
equivalent to asctime(localtime([secs]))

mktime(timeTuple)

Inverse of localtime(). Returns a float representing a number of seconds.

strftime(format, timeTuple)

Formats a time tuple as a string, according to format (see table below). Current time is used if timeTuple is omitted.

strftime(string[, format])

Parses a string representing a time according to format (same format as for strftime(), see below), default `%a %b %d %H:%M:%S %Y` = asctime format.

Returns a time tuple/struct_time.

sleep(secs)

Suspends execution for secs seconds. secs can be a float.

### Formatting in strftime() and strftime()

**Directive**

- `%a`
  - Locale's abbreviated weekday name.
- `%A`
  - Locale's full weekday name.
- `%b`
  - Locale's abbreviated month name.
- `%B`
  - Locale's full month name.
- `%c`
  - Locale's appropriate date and time representation.
- `%d`
  - Day of the month as a decimal number [01,31].
- `%D`
  - Microsecond as a decimal number [0,999999], zero-padded on the left.
- `%H`
  - Hour (24-hour clock) as a decimal number [00,23].
- `%h`
  - Hour (12-hour clock) as a decimal number [01,12].
- `%j`
  - Day of the year as a decimal number [001,366].
- `%m`
  - Month as a decimal number [01,12].
- `%n`
  - Minute as a decimal number [00,59].
- `%p`
  - Locale's equivalent of either AM or PM.
- `%S`
  - Second as a decimal number [00,61]. Yes, 61!
- `%s`
  - Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
- `%S`
  - Weekday as a decimal number [0(Sunday),6].
- `%w`
  - Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
- `%x`
  - Locale's appropriate date representation.
- `%X`
  - Locale's appropriate time representation.
- `%y`
  - Year without century as a decimal number [00,99].
- `%Y`
  - Year with century as a decimal number.
- `%z`
  - Time zone name (no characters if no time zone exists).
- `%Z`
  - UTC offset in the form +HHMM or -HHMM (empty string if the date is naive).

### string

Common string operations.

As of Python 2.0, much (though not all) of the functionality provided by the string module have been superseded by built-in string methods.

Since 2.5 (7) all string module methods are considered deprecated => use built-in string methods instead.

#### Some string constant

<table>
<thead>
<tr>
<th>Constant</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>digits</td>
<td>The string '0123456789'.</td>
</tr>
<tr>
<td>hexdigits</td>
<td>Legal hexadecimal &amp; octal digits.</td>
</tr>
<tr>
<td>octdigits</td>
<td>Strings containing the appropriate characters, taking the current locale into account.</td>
</tr>
<tr>
<td>letters, uppercase, lowercase, whitespace</td>
<td>Strings containing Ascii characters.</td>
</tr>
<tr>
<td>ascii_letters, ascii_lower, ascii_uppercase</td>
<td></td>
</tr>
</tbody>
</table>

#### Some string functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>expandtabs(s, tabSize)</td>
<td>Returns a copy of string s with tabs expanded.</td>
</tr>
<tr>
<td>find(rfind(s, sub[, start=0[, end=0]])</td>
<td>Returns the lowest/highest index in s where the substring sub is found such that sub is wholly contained in s[start:end]. Return -1 if sub not found.</td>
</tr>
<tr>
<td>ljust(rjust/center(s, width[, fillChar= ’'])</td>
<td>Returns a copy of string s; left/right justified/centered in a field of given width, padded with spaces or the given character. s is never truncated.</td>
</tr>
</tbody>
</table>
lower/upper(s)
split(s[, sep=whitespace, maxsplit=0])
rsplit(s[, sep=whitespace, maxsplit=0])
join(words[, sep='])
replace(s, old, new[, maxsplit=0])
strip(s[, chars=None])

Returns a string that is (a copy of) s in lowercase/uppercase.
Returns a list containing the words of the string s, using the string sep as a separator.
Same as split above but starts splitting from the end of the string, e.g.
'\A,B,C'.split('\', l) == ['\A', '\B,C'] but 'A,B,C'.rsplit('\', l) == ['A,B', '\C']
Concatenates a list or tuple of words with intervening separators; inverse of split.
Returns a copy of string s with all occurrences of substring old replaced by new. Limits
to maxsplit first substitutions if specified.
Returns a string that is (a copy of) s without leading and trailing chars (default: whitespace), if any. Also lstrip, rstrip.

re (sre)

Regular expression operations.

Handles Unicode strings. Implemented in new module sre, re now is a mere front-end for compatibility.
Patterns are specified as strings. Tip: Use raw strings (e.g. r'\w\*') to literalize backslashes.

**Regular expression syntax**

<table>
<thead>
<tr>
<th>Form</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Matches any character (including newline if DOTALL flag specified).</td>
</tr>
<tr>
<td>^</td>
<td>Matches start of the string (of every line in MULTILINE mode).</td>
</tr>
<tr>
<td>$</td>
<td>Matches end of the string (of every line in MULTILINE mode).</td>
</tr>
<tr>
<td>*</td>
<td>o or more of preceding regular expression (as many as possible).</td>
</tr>
<tr>
<td>+</td>
<td>1 or more of preceding regular expression (as many as possible).</td>
</tr>
<tr>
<td>?</td>
<td>0 or 1 occurrence of preceding regular expression.</td>
</tr>
<tr>
<td>*, +, ??</td>
<td>Same as *, + and ? but matches as few characters as possible.</td>
</tr>
<tr>
<td>{m,n}</td>
<td>Matches from m to n repetitions of preceding RE.</td>
</tr>
<tr>
<td>{m,n}?</td>
<td>Idem, attempting to match as few repetitions as possible.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Defines character set: e.g. '[a-zA-Z]' to match all letters (see also \w \W).</td>
</tr>
<tr>
<td>[^ ]</td>
<td>Defines complemented character set: matches char if NOT in set.</td>
</tr>
<tr>
<td>\</td>
<td>Escapes special chars &quot;\$%(){}[]^|&quot; and introduces special sequences (see below). Due to Python string rules, write as &quot;\&quot; or x&quot; in the pattern string.</td>
</tr>
<tr>
<td>\w</td>
<td>Matches a literal \w; due to Python string rules, write as &quot;\w</td>
</tr>
<tr>
<td>\W</td>
<td>Specifies alternative: 'foo</td>
</tr>
<tr>
<td>(...)</td>
<td>Matches any RE inside (), and delimits a group.</td>
</tr>
<tr>
<td>(?...?)</td>
<td>Idem but doesn't delimit a group (non capturing parenthesis).</td>
</tr>
<tr>
<td>(?=...)</td>
<td>Matches any RE inside (), and delimits a named group, (e.g. r'(?P&lt;id&gt;[a-zA-Z._]\w*)' defines a group named id).</td>
</tr>
<tr>
<td>P&lt;name&gt;...</td>
<td>Matches whatever text was matched by the earlier group named name.</td>
</tr>
<tr>
<td>(?P=name)</td>
<td>Matches if ... matches next, but doesn't consume any of the string e.g. 'Isaac (?=Asimov)' matches 'Isaac' only if followed by 'Asimov'.</td>
</tr>
<tr>
<td>(?!...)</td>
<td>Matches if ... doesn't match next. Negative of (?...).</td>
</tr>
<tr>
<td>(?&lt;...)</td>
<td>Matches if the current position in the string is preceded by a match for ... that ends at the current position. This is called a positive lookbehind assertion.</td>
</tr>
<tr>
<td>(?&lt;!...)</td>
<td>Matches if the current position in the string is not preceded by a match for ... This is called a negative lookbehind assertion.</td>
</tr>
<tr>
<td>(?&lt;group&gt;)</td>
<td>[2.4+] group is either a numeric group ID or a group name defined with (?P&lt;group&gt;...). If the specified group matched, the regular expression pattern A will be tested against the string; if the group didn't match, the pattern B will be used instead.</td>
</tr>
<tr>
<td>(?!!)</td>
<td>A comment; ignored.</td>
</tr>
<tr>
<td>(?letters)</td>
<td>Letters is one or more of 'a', 'b', 'c', 'd', 'e', 'f'. Sets the corresponding flags (re.I, re.L, re.M, re.S, re.U, re.X) for the entire RE. See the compile() function for equivalent flags.</td>
</tr>
</tbody>
</table>

**Special sequences**

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\number</td>
<td>Matches content of the group of the same number; groups are numbered starting from 1.</td>
</tr>
<tr>
<td>\A</td>
<td>Matches only at the start of the string.</td>
</tr>
<tr>
<td>\b</td>
<td>Empty str at beginning or end of word: \bis\b' matches 'is', but not 'his'.</td>
</tr>
<tr>
<td>\B</td>
<td>Empty str NOT at beginning or end of word.</td>
</tr>
<tr>
<td>\d</td>
<td>Any decimal digit (&lt;= 0-9).</td>
</tr>
<tr>
<td>\D</td>
<td>Any non-decimal digit char (&lt;= 0-9).</td>
</tr>
<tr>
<td>\s</td>
<td>Any whitespace char (&lt;= [\ \t\n\r\f\v]).</td>
</tr>
<tr>
<td>\S</td>
<td>Any non-whitespace char (&lt;= [\ \t\n\r\f\v]).</td>
</tr>
<tr>
<td>\w</td>
<td>Any alphaNumeric char (depends on LOCALE flag).</td>
</tr>
<tr>
<td>\W</td>
<td>Any non-alphaNumeric char (depends on LOCALE flag).</td>
</tr>
<tr>
<td>\Z</td>
<td>Matches only at the end of the string.</td>
</tr>
</tbody>
</table>

**Variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>error</td>
<td>Exception when pattern string isn't a valid regexp.</td>
</tr>
</tbody>
</table>

**Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>compile(pattern[, flags=0])</td>
<td>Compiles a RE pattern string into a regular expression object.</td>
</tr>
</tbody>
</table>

Flags (combinable by |)
| I or IGNORECASE | ? |


case insensitive matching

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Lor LOCATE &lt;= (f)</code></td>
<td>makes <code>\w</code>, <code>\W</code>, <code>\b</code>, <code>\B</code> dependent on the current locale</td>
</tr>
<tr>
<td><code>M or MULTILINE &lt;= (m)</code></td>
<td>matches every new line and not only start/end of the whole string</td>
</tr>
<tr>
<td><code>S or DOTALL &lt;= (s)</code></td>
<td>matches ALL chars, including newline</td>
</tr>
<tr>
<td><code>U or UNICODE &lt;= (u)</code></td>
<td>Make <code>\w</code>, <code>\W</code>, <code>\b</code>, and <code>\B</code> dependent on the Unicode character properties database.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>escape(string)</code></td>
<td>Returns (a copy of) string with all non-alphanumerics backslashed.</td>
</tr>
<tr>
<td><code>match(pattern, string[, flags])</code></td>
<td>If 0 or more chars at beginning of string matches the RE pattern string, returns a corresponding <code>MatchObject</code> instance, or <code>None</code> if no match.</td>
</tr>
<tr>
<td><code>search(pattern, string[, flags])</code></td>
<td>Scans thru string for a location matching pattern, returns a corresponding <code>MatchObject</code> instance, or <code>None</code> if no match.</td>
</tr>
<tr>
<td><code>split(pattern, string[, maxsplit=0 [, flags=0]])</code></td>
<td>Splits string by occurrences of pattern. If capturing () are used in pattern, then occurrences of patterns or subpatterns are also returned.</td>
</tr>
<tr>
<td><code>findall(pattern, string)</code></td>
<td>Returns a list of non-overlapping matches of pattern in string, either a list of groups or a list of tuples if the pattern has more than 1 group.</td>
</tr>
<tr>
<td><code>finditer(pattern, string[, flags])</code></td>
<td>Returns an iterator over all non-overlapping matches of pattern in string. For each match, the iterator returns a <code>match</code> object. Empty matches are included in the result unless they touch the beginning of another match.</td>
</tr>
<tr>
<td><code>sub(pattern, repl, string[, count=0 [, flags]])</code></td>
<td>Returns string obtained by replacing the (count first) leftmost non-overlapping occurrences of pattern (a string or a RE object) in string by repl; repl can be a string or a function called with a single <code>MatchObj</code> arg, which must return the replacement string.</td>
</tr>
<tr>
<td><code>subn(pattern, repl, string[, count=0 [, flags]])</code></td>
<td>Same as <code>sub()</code>, but returns a tuple (newString, numberOfSubsMade).</td>
</tr>
</tbody>
</table>

### Regular Expression Objects

RE objects are returned by the compile function.

#### re object attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>flags</code></td>
<td>Flags arg used when RE obj was compiled, or <code>o</code> if none provided.</td>
</tr>
<tr>
<td><code>groupindex</code></td>
<td>Dictionary of {group name: group number} in pattern.</td>
</tr>
<tr>
<td><code>pattern</code></td>
<td>Pattern string from which RE obj was compiled.</td>
</tr>
</tbody>
</table>

#### re object methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>match(string[, pos][, enapos])</code></td>
<td>If zero or more characters at the beginning of string match this regular expression, returns a corresponding <code>MatchObject</code> instance. Returns <code>None</code> if the string does not match the pattern; note that this is different from a zero-length match.</td>
</tr>
<tr>
<td><code>search(string[, pos][, enapos])</code></td>
<td>Scans through string looking for a location where this regular expression produces a match, and returns a corresponding <code>MatchObject</code> instance. Returns <code>None</code> if no position in the string matches the pattern; note that this is different from finding a zero-length match at some point in the string.</td>
</tr>
<tr>
<td><code>split(string[, maxsplit=0])</code></td>
<td>Identical to the <code>split()</code> function, using the compiled pattern.</td>
</tr>
<tr>
<td><code>findall([string[, pos][, enapos]])</code></td>
<td>Identical to the <code>findall()</code> function, using the compiled pattern.</td>
</tr>
<tr>
<td><code>finditer([string[, pos][, enapos]])</code></td>
<td>Identical to the <code>finditer()</code> function, using the compiled pattern.</td>
</tr>
<tr>
<td><code>sub([repl, string[, count=0][, flags]])</code></td>
<td>Identical to the <code>sub()</code> function, using the compiled pattern.</td>
</tr>
<tr>
<td><code>subn([repl, string[, count=0][, flags]])</code></td>
<td>Identical to the <code>sub()</code> function, using the compiled pattern.</td>
</tr>
</tbody>
</table>

### Match Objects

Match objects are returned by the match & search functions.

#### Match object attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pos</code></td>
<td>Value of pos passed to search or match functions; index into string at which RE engine started search.</td>
</tr>
<tr>
<td><code>endpos</code></td>
<td>Value of endpos passed to search or match functions; index into string beyond which RE engine won't go.</td>
</tr>
<tr>
<td><code>re</code></td>
<td>RE object whose match or search function produced this <code>MatchObj</code> instance.</td>
</tr>
<tr>
<td><code>string</code></td>
<td>String passed to <code>match()</code> or <code>search()</code>.</td>
</tr>
</tbody>
</table>

#### Match object methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>group([a1, a2, ...])</code></td>
<td>Returns one or more groups of the match. If one <code>arg</code>, result is a string; if multiple args, result is a tuple</td>
</tr>
</tbody>
</table>
with one item per arg. If g is 0, returns the entire matching string; if 1 \leq g \leq 99, returns string matching group \#gi (or None if no such group); gi may also be a group name.

- **span(group)**
  - Returns a tuple of all groups of the match; groups not participating to the match have a value of None.
  - Returns a string instead of tuple if len(tuple) == 1.

- **start(group), end(group)**
  - Returns indices of start & end of substring matched by group (or None if group exists but didn't contribute to the match).

**Lexical scanners using regular expressions**

There's an undocumented class in the `re` module called `re.Scanner`. The following recipe is from stackoverflow:

```python
import re
scanner= re.Scanner([
    (r"[0-9]+", lambda scanner,token: ("INTEGER", token)),
    (r"\w+", lambda scanner,token: ("IDENTIFIER", token)),
    (r"\", lambda scanner,token: ("PUNCTUATION", token)),
    (r":", None), # None == skip token.
])
results, remainder = scanner.scan("45 pigeons, 23 cows, 11 spiders.")
print results
```

which results in

```python
(['INTEGER', '45'],
 ['IDENTIFIER', 'pigeons'],
 ['PUNCTUATION', '.'],
 ['INTEGER', '23'],
 ['IDENTIFIER', 'cows'],
 ['PUNCTUATION', '.'],
 ['IDENTIFIER', 'spiders'],
 ['PUNCTUATION', '.'])
```

**Math**

For complex number functions, see module `cmath`. For intensive number crunching, see Numerical Python and the Python and Scientific computing page.

**Constants**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>pi</td>
<td>3.1415926535897931</td>
</tr>
<tr>
<td>e</td>
<td>2.7182818284590451</td>
</tr>
</tbody>
</table>

**Functions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>acos(x)</td>
<td>Returns the arc cosine (measured in radians) of x.</td>
</tr>
<tr>
<td>acosh(x)</td>
<td>Returns the hyperbolic arc cosine (measured in radians) of x.</td>
</tr>
<tr>
<td>asin(x)</td>
<td>Returns the arc sine (measured in radians) of x.</td>
</tr>
<tr>
<td>asinh(x)</td>
<td>Returns the hyperbolic arc sine (measured in radians) of x.</td>
</tr>
<tr>
<td>atan(x)</td>
<td>Returns the arc tangent (measured in radians) of x.</td>
</tr>
<tr>
<td>atan2(y, x)</td>
<td>Returns the arc tangent (measured in radians) of y/x. The result is between -pi and pi. Unlike atan(y/x), the signs of both x and y are considered.</td>
</tr>
<tr>
<td>atanh(x)</td>
<td>Returns the hyperbolic arc tangent (measured in radians) of x.</td>
</tr>
<tr>
<td>ceil(x)</td>
<td>Returns the ceiling of x as a float. This is the smallest integral value \geq x.</td>
</tr>
<tr>
<td>copysign(x, y)</td>
<td>Copies the sign bit of an IEEE 754 number, returning the absolute value of x combined with the sign bit of y, e.g. copysign(1, -0.0) returns -1.0.</td>
</tr>
<tr>
<td>cos(x)</td>
<td>Returns the cosine of x (measured in radians).</td>
</tr>
<tr>
<td>cosh(x)</td>
<td>Returns the hyperbolic cosine of x.</td>
</tr>
<tr>
<td>degrees(x)</td>
<td>Converts angle x from radians to degrees.</td>
</tr>
<tr>
<td>erf(x)</td>
<td>Return the error function at x.</td>
</tr>
<tr>
<td>erfc(x)</td>
<td>Return the complementary error function at x.</td>
</tr>
<tr>
<td>exp(x)</td>
<td>Returns e raised to the power of x.</td>
</tr>
<tr>
<td>exp1(x)</td>
<td>Return e**x - 1 with less loss of precision at small floats than exp(x) - 1.</td>
</tr>
<tr>
<td>fabs(x)</td>
<td>Returns the absolute value of the float x.</td>
</tr>
<tr>
<td>factorial(n)</td>
<td>returns n!</td>
</tr>
<tr>
<td>floor(x)</td>
<td>Returns the floor of x as a float. This is the largest integral value \leq x.</td>
</tr>
<tr>
<td>fmod(x, y)</td>
<td>Returns fmod(x, y), according to platform C. x % y may differ.</td>
</tr>
<tr>
<td>frexp(x)</td>
<td>Returns the mantissa and exponent of x, as pair (m, e). m is a float and e is an int, such that x = m * 2**e. If x is 0, m and e are both 0. Else 0.5 &lt; abs(m) &lt; 1.0.</td>
</tr>
<tr>
<td>fsum(Iterable)</td>
<td>Returns an accurate floating point sum of values in iterable (assumes IEEE-754 floating point arithmetic).</td>
</tr>
<tr>
<td>gamma(x)</td>
<td>Return the Gamma function at x.</td>
</tr>
<tr>
<td>hypot(x, y)</td>
<td>Returns the Euclidean distance \sqrt{x<em>x + y</em>y}.</td>
</tr>
<tr>
<td>isinf(x)</td>
<td>Returns True if x is infinite (positive or negative).</td>
</tr>
<tr>
<td>isnan(x)</td>
<td>Returns True if x is not a number.</td>
</tr>
<tr>
<td>ldexp(x, l)</td>
<td>x * (2**l)</td>
</tr>
</tbody>
</table>
lgamma(x)  Return the natural logarithm of the absolute value of the Gamma function at x.
log(x, base)) Returns the logarithm of x to the given base. If the base is not specified, returns the natural logarithm (base e) of x.
log10(x) Returns the base 10 logarithm of x.
log1p(x) Returns the natural logarithm of 1+x (base e). The result is computed in a way which is accurate for x near zero.
mod(x) Returns the fractional and integer parts of x. Both results carry the sign of x. The integer part is returned as a float.
pow(x, y) Returns x**y (x to the power of y). Note that for y=2, it is more efficient to use x*x.
radians(x) Converts angle x from degrees to radians.
sin(x) Returns the sine (measured in radians) of x.
sinh(x) Returns the hyperbolic sine of x.
sqrt(x) Returns the square root of x.
tan(x) Returns the tangent (measured in radians) of x.
tanh(x) Returns the hyperbolic tangent of x.
trunc(x) Returns the real value x truncated to an Integral. Delegates to x.__trunc__().

Compressions

Python contains several modules for working with compressed files. The builtin function zip does not have anything to do with zipping, think instead of a zipper.

There are three different concepts with compressions:

- compression of data
- compression of a single file (e.g. gzip, bz2)
- compression of archives, i.e. zip-files with multiple files

**Compression of data**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zlib</td>
<td>Compression and decompression of data (strings), using the zlib library.</td>
</tr>
<tr>
<td>bz2</td>
<td>Sequential compression and decompression using classes BZ2Compressor and BZ2Decompressor, or One-shot decompression through functions compress() and decompress().</td>
</tr>
</tbody>
</table>

**Compression of a single file**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>Read and write gzip-compressed files as were they normal files, using the GZipFile class.</td>
</tr>
<tr>
<td>bz2</td>
<td>Read and write bz2-compressed files as were they normal files, using the BZ2File class.</td>
</tr>
</tbody>
</table>

**Compression of archives**

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>zipfile</td>
<td>Work with ZIP archives. See the method zipfile.open for reading a single file in the archive as a normal file.</td>
</tr>
<tr>
<td>tarfile</td>
<td>Read and write tar archive files.</td>
</tr>
<tr>
<td>shutil</td>
<td>The function make_archive provides means for packaging a directory into a archive.</td>
</tr>
</tbody>
</table>

List of modules and packages in base distribution

Built-ins and content of python Lib directory. The subdirectory Lib/site-packages contains platform-specific packages and modules.

[Main distributions (Windows, Unix), some OS specific modules may be missing]

**Standard library modules**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>builtin</strong></td>
<td>Provide direct access to all ‘built-in’ identifiers of Python, e.g. <strong>builtins</strong>.open is the full name for the built-in function open().</td>
</tr>
<tr>
<td><strong>future</strong></td>
<td>Future statement definitions. Used to progressively introduce new features in the language.</td>
</tr>
</tbody>
</table>
| __main__ | Represent the (otherwise anonymous) scope in which the interpreter’s main program executes — commands read either from standard input, from a script file, or from an interactive prompt. Typical idiom to check if a code was run as a script (as opposed to being imported):

```python
if __name__ == '__main__':
    main()  # (this code was run as script)
```

| abc       | (new in 2.6) Abstract Base Classes (ABC) [PEP 3119]. Equivalent of Java interfaces. The module collections defines interfaces/ABCs for many behaviors/protocols/data structures (Iterable, Hashable, Sequence, Set, etc.). |
| aifc      | Stuff to parse AIFF-C and AIFF files. |
| anydbm    | Generic interface to all dbm clones. (dbhash, gdbm, dbm, dumdbm). |
| argparse  | Parser for command-line options, arguments and sub-commands. For more C-like command-line processing, see getopt. |
| array     | Efficient arrays of numeric values. |
| ast       | (new in 2.6) Helpers to process Trees of the Python Abstract Syntax grammar. |
| asynchat  | A class supporting chat-style (command/response) protocols. |
| asynore    | Basic infrastructure for asynchronous socket service clients and servers. |
| atexit    | Register functions to be called at exit of Python interpreter. |
| audio      | Classes for manipulating audio devices (currently only for Sun and SGI). Deprecated since 2.6. |
| csv       | Manipulate raw audio data. |
| C         | Supports the ASCII encoding |

| python     | Supports the ASCII encoding |
| cli        | Supports the ASCII encoding |
| cli        | Supports the ASCII encoding |
manycore can unit test, 2.5: support for almost everything.

Converts to/from base64 transport encoding as per RFC-1521.

HTTP server base class

"Bastionification" utility (control access to instance vars).

A generic Python debugger base class.

Convert between binary and ASCII.

Macintosh binhex compression/decompression.

Bisection algorithms.

(Optional) improved BSD database interface [package].

BZ2 compression.

Calendar printing functions.

Wraps the WWW Forms Common Gateway Interface (CGI).

Traceback manager for CGI scripts.

Read IFF chunked data.

Mathematical functions for complex numbers. See also math.

A generic class to build line-oriented command interpreters.

Efficiently compare files, boolean outcome only.

Same, but caches start results for speed.

Utilities needed to emulate Python's interactive interpreter.

Lookup existing Unicode encodings and register new ones. 2.3: support for incremental codecs.

Utilities to compile possibly incomplete Python source code.

High-performance container datatypes. 2.4: The only datatype defined is a double-ended queue
deque. 2.5: Type deque has now a remove method. New type defaultdict. 2.6: New type
collections.namedtuple, Define many ABCs (Abstract Base Classes) like Container, Hashable, Iteratable,
Sequence, Set...

Conversion functions between RGB and other color systems.

Execute shell commands via os.popen [Unix].

Force "compilation" of all .py files in a directory.

Configuration file parser (much like windows .ini files).

Utilities for with statement contexts.

HTTP state (cookies) management.

Generic shallow and deep copying operations.

Helper to provide extensibility for modules pickle/cPickle.

Faster, C implementation of pickle.

Faster, C implementation of profile.

Function to check Unix passwords [Unix].

Faster, C implementation of StringIO.

Tools to read comma-separated files (and variations thereof). 2.5: Several enhancements.

"Foreign function" library for Python. Provides C compatible data types, and allows to call functions
in dlls/shared libraries. Can be used to wrap these libraries in pure Python.

Terminal handling for character-cell displays [Unix/OS2/DOS only].

Improved date/timetpes (date, time, datetime, timegm()). 2.5: New method
strftime(string, format) for class datetime. 2.6: strftime() new format code %f expanding to
number of s.

(g)dbm-compatible interface to bsdhash.hashopen.

Decimal floating point arithmetic.

Tool for comparing sequences, and computing the changes required to convert one into another.

2.5: Improved SequenceMatcher.get_matching_blocks() method.

Sorted list of files in a dir, using a cache. Deprecated since 2.6.

Defines a class to build directory ditto tools on.

Bytecode disassembler.

Package installation system. 2.5: Function setup enhanced with new keyword parameters
requires, provides, obsoletes, and download_url [PEP314].

Registers a module in the Python package index (PyPI). This command plugin adds the register
command to distutils scripts.

In 2.7 moved to separate module sysconfig.

Call C functions in shared objects [Unix]-Deprecated since 2.6.

Unit testing framework based on running examples embedded in doctests. 2.5: New SKIP option.

New encoding arg to testfile() function.

Creation of self-documenting XML-RPC servers, using pydoc to create HTML API doc on the fly. 2.5:
New attribute rpc_paths.

Common operations on OS pathnames.

A dumb and slow but simple dbm clone.

Print python code that reconstructs a variable.

Helpers to make it easier to write code that uses threads where supported, but still runs on Python
versions without thread support. The dummy modules simply run the threads sequentially.

A package for parsing, handling, and generating email messages. New version 3.0 dropped various
deprecated APIs and removes support for Python versions earlier than 2.3. 2.5: Updated to version
4.0.

New codecs: idna (IDNA strings), koi8_u (Ukrainian), palmos (PalmOS 3.5), punycode
(Puny code IDNA codec), string_escape (Python string escape codec: replaces non-printable chars
w/ Python-style string escapes). New codecs in 2.4: HP Roman8, ISO_8859-11, ISO_8859-16,
PCTP-154, TIS-620; Chinese, Japanese and Korean codes.

Standard errno system symbols. The value of each symbol is the corresponding integer value
exceptions  Class based built-in exception hierarchy.
fcntl  The fcntl() and ioctl() system calls [Unix].
filecmp  File and directory comparison.
fileinput  Helper class to quickly write a loop over all standard input files. 2.5: Made more flexible (Unicode filenames, mode parameter, etc...)
find  Find files directory hierarchy matching a pattern.
fnmatch  File name matching with shell patterns.
format  Generic output formatting.
fpel  Floating point exception control [Unix].
ftformat  General floating point format functions. Deprecated since 2.6.
fraction  (new in 2.6) Rational Numbers.
ftp  An FTP client class. Based on RFC 959.
funcetools  Tools for functional-style programming. See in particular function partial() [PEP209].
futur0_builtins  (new in 2.6) Python 3 builtins. Provides functions that exist in 2.x but have different behavior in Python 3 (ascii, map, filter, hex...). To write Python 3 compatible code, import the functions from this module, e.g.:

    from future_builtins import map
    ...code using Python3-style map()...

gc  Perform garbage collection, obtain GC debug stats, and tune GC parameters. 2.5: New get_count() function. gc.collect() takes a new generation argument.
gdbm  GNU's reinterpretation of dbm [Unix].
getopt  Standard command line processing in C getopt() style. See also argparser.
getpass  Utilities to get a password and/or the current user name.
ggettext  Internationalization and localization support.
glob  Filename "globbing" utility.
gopherlib  Gopher protocol client interface.
grt  The group database [Unix].
grep  grep utilities:
gzip  Read & write gzipped files.
hashlib  Secure hashes and message digests.
heapq  Heap queue (priority queue) helpers. 2.5: nsmallest() and nlargest() takes a key keyword param.
hmac  HMAC (Keyed-Hashing for Message Authentication).
hotshot. stones  Helper to run the pystone benchmark under the Hotshot profiler.
hmtlentitydefs  HTML character entity references.
httmilib  HTML2 parsing utilities. Deprecated since 2.6; see HTMLParser-class.
HTMLParser  Simple HTML and XHTML parser.
httpie  HTTP client class.
ideleib  (package) Support library for the IDLE development environment.
imageop  Manipulate raw image data. Deprecated since 2.6; removed in Python 3.
imaplib  IMAP4 client. Based on RFC 2060.
imghdr  Recognizing image files based on their first few bytes.
imp  Access the import internals.
imputil  Provides a way of writing customized import hooks.
inspect  Get information about live Python objects.
io  (new in 2.6) Core tools for working with streams [PEP 3116]. Define Abstract Base Classes RawIOBase (I/O operations: read, write, seek...), BufferedReader buffering), and TextIOBase (reading & writing strings).
iter tools  Tools to work with iterators and lazy sequences. 2.5: islice() accepts None for start & step args.
json  (new in 2.6) JSON (JavaScript Object Notation) interchange format support.
keyw  List of Python keywords.
knee  A Python reimplementation of hierarchical module import.
linecache  Cache lines from files.
lxmlaudiodev  Linux audio support. Replaced by ossaudiodev (Linux).
locale  Support for number formatting using the current locale settings. 2.5: format() modified; new functions format_string() and currency()
logging  (package) Tools for structured logging in log4j style.
macpath  Pathname (or related) operations for the Macintosh [Mac].
macularzpath  Mac specific module for conversion between pathnames and URLs [Mac].
mailbox  Classes to handle Unix style, MIME style, and MH style mailboxes. 2.5: added capability to modify mailboxes in addition to reading them.
mailcap  Mailcap file handling (RFC 1524).
marschal  Internal Python object serialization.
markups  Shared support for scanning document type declarations in HTML and XHTML.
math  Mathematical functions. See also emath
markdown  MDx message digest algorithm. 2.5: New a more wrapper around new library hashlib. Deprecated since 2.6, use hashlib module instead.
mblib  MH (mailbox) interface. Deprecated since 2.6.
nitrotools  Various tools used by MIME-reading or MIME-writing programs. Deprecated since 2.6.
mimetypes  Guess the MIME type of a file.
mime  Generic MIME writer. Deprecated since 2.3, use email package instead.
mimify  Minification and unminification of mail messages. Deprecated since 2.6, use email package instead.
mmap  Interface to memory-mapped files - they behave like mutable strings.
Deprecation warnings...
A wrapper to allow subclassing or built-in string class (useless with new-style classes since python 2.2, str is subclassable).

**util**

Some useful functions that don't fit elsewhere!

**uu**

Implementation of the UUnenc and UUdecode functions.

**warnings**

Python part of the warnings subsystem. Issue warnings, and filter unwanted warnings.

**wave**

Stuff to parse WAVE files.

**weakref**

Weak reference support for Python. Also allows the creation of proxy objects. 2.5: new methods

**webbrowser**

Several routines that help recognizing sound files.

**whichdb**

Guess which db package to use to open a db file.

**wdm**

Widmenn Hill random number generator (obsolete, use random instead).

**winsound**

Sound-playing interface for Windows [Windows].

**wsgiref**

WSGI Utilities and Reference Implementation.

**xml**

Implements (a subset of) Sun XDR (eXternal Data Representation).

**xml.etree.ElementTree**

Subset of Fredrik Lundh's ElementTree library for processing XML.

**xml.parsers.expat**

An interface to the Expat non-validating XML parser.

**xml.sax**

Classes for processing XML using the DOM (Document Object Model). 2.3: New modules

**xmlrpclib**

An XML-RPC client interface for Python. 2.5: Supports returning datetime objects for the XML-RPC date type.

**threadng**

Provides a sequence-like object for reading a file line-by-line without reading the entire file into memory. Deprecated since release 2.3. Use for line in file instead. Removed since 2.4

**zipfile**

Read & write PK zipped files. 2.3: Supports ZIP4 version, a .zip archive can now be larger than 4GB. 2.6: Class zipfile has new methods extract() and extractall().

**zipimport**

ZIP archive importer.

**zlib**

Compression compatible with gzip. 2.5: Compress and Decompress objects now support a copy() method.

**unrot**

Demonstration of abstract mathematical concepts.

---

**Workspace exploration and idiom hints**

```
  dir(object)
  dir()
  if __name__ == '__main__':
    main()
  map(None, lst1, lst2, ...)
  b = a[:]
  b = list(a)
  a, b, c = 1, 2, 3
  for key, value in dic.items(): ...
  if 1 < x <= 5: ...
  for line in fileinput.input(): ...
```

- list valid attributes of object (which can be a module, type or class object)
- list names in current local symbol table.
- invoke main() if running as script
- merge lists; see also zip(lst1, lst2, ...)
- create a copy b of sequence a
- If a is a list, create a copy of it.
- Works also in this context
- Works as expected
- Process each file in command line args, one line at a time (underscore) in interactive mode, refers to the last value printed.

---

**Python Mode for Emacs**

Emacs goodies available here.

(The following has not been revised, probably not up to date - any contribution welcome -)

Type C-c ? when in python-mode for extensive help.

**INDENTATION**

Primarily for entering new code:

- **TAB** indent line appropriately
- **LF** insert newline, then indent
- **DEL** reduce indentation, or delete single character

Primarily for reindenting existing code:

- **C-C-C** guess py-indent-offset from file content; change locally
- **C-M-C** : ditto, but change globally
- **C-TAB** reindent region to match its context
- **C-C** < shift region left by py-indent-offset
- **C-C >** shift region right by py-indent-offset

**MARKING & MANIPULATING REGIONS OF CODE**

- **C-C** b mark block of lines
- **M-C-h** mark smallest enclosing def
- **C-u M-C-h** mark smallest enclosing class
- **C-c #** comment out region of code
- **C-u C-c #** uncomment region of code

**MOVING POINT**

- **C-** c c-p move to statement preceding point
- **C-c C-n** move to statement following point
- **C-c C-u** move up to start of current block
- **M-** c move to start of def
Changes to this document

April, 2013 (Stefan McKinnon Høj-Edwards)

Corrections

- Added strikethrough to deprecated modules in module-list.
- Corrected links in modules list.
- Added a recipe for the secret re.Scanner.
- Added context manager methods to special methods in classes.

Oct, 2011 (Stefan McKinnon Høj-Edwards)
Upgraded to Python 2.7

Prior to Oct. 2011

see Last updated on-list