

Python 2.3 Quick Reference

Contents

- [Front matter](#)
- [Invocation Options](#)
- [Environment variables](#)
- [Lexical entities](#) : [keywords](#), [identifiers](#), [string literals](#), [boolean constants](#), [numbers](#), [sequences](#), [dictionaries](#), [operators](#)
- [Basic types and their operations](#): [None](#), [bool](#), [Numeric types](#), [sequence types](#), [list](#), [dictionary](#), [string](#), [file](#), [sets](#)
- [Advanced types](#)
- [Statements](#)
- [Iterators](#); [Generators](#); [Descriptors](#)
- [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](#), [getopt](#)
- [List of modules in the base distribution](#)
- [Workspace exploration and idiom hints](#)
- [Python mode for Emacs](#)

Front matter

Version 2.3

The latest version is to be found [here](#).

Please **report** errors, inaccuracies and suggestions to [Richard Gruet](#) (pqr at rgruet.net).

19 Sep 2004, rev 4

upgraded by Richard Gruet for Python 2.3

11 May 2003, rev 4

upgraded by Richard Gruet for Python 2.2 (restyled by [Andrei](#))

7 Aug 2001

upgraded by Simon Brunning for Python 2.1

16 May 2001

upgraded by Richard Gruet and [Simon Brunning](#) for Python 2.0

18 Jun 2000

upgraded by [Richard Gruet](#) for Python 1.5.2

30 Oct 1995

created by [Chris Hoffmann](#) for Python 1.3

Color coding:

[Features added in 2.3 since 2.2.](#)

[Features added in 2.2 since 2.1.](#)

[Features added in 2.1 since 2.0.](#)

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](#)

- **Python's nest**: <http://www.python.org>
 - **Official documentation**: <http://www.python.org/doc/>
 - **Other doc & books**: [Dive into Python](#) , [Python Cookbook](#) , [Faqs](#) , [Thinking in Python](#) , [Text processing in Python](#)
 - **Packages**: [Python Package Index \(PyPI\)](#) , [Vaults of Parnassus](#) , [SourceForge \(search "python"\)](#) , [O'Reilly Python DevCenter](#) , [Starship Python](#)
 - **Wiki**: [moinmoin](#)
 - **Newsgroups**: [comp.lang.python](#) and [comp.lang.python.announce](#)
 - **Misc pages**: [Daily Python URL](#) , [Kevin Altis' WebLog](#)
 - **Development**: <http://python.sourceforge.net/>
 - **Jython** (Java impl. of Python): <http://www.jython.org/>
 - **ActivePython**: <http://www.ActiveState.com/ASPN/Python/>
 - **Help desk**: help@python.org
 - An excellent Python reference book: [Python Essential Reference](#) by David Beazley & Guido Van Rossum (Other New Riders)

Tip: From within the Python interpreter, use `help` or `help(object)`.

Invocation Options

python[w] [-dEhiOQStuUvVWxX?] [-c *command* | *scriptFile* | -] [*args*]
(pythonw does not open a terminal/console; python does)

Invocation Options

Option	Effect
-d	Output parser debugging information (also PYTHONDEBUG=x)
-E	Ignore environment variables (such as PYTHONPATH)
-h	Print a help message and exit (formerly -?)
-i	Inspect interactively after running script (also PYTHONINSPECT=x) and force prompts, even if stdin appears not to be a terminal.
-O	Optimize generated bytecode (also PYTHONOPTIMIZE=x). Asserts are suppressed.
-OO	Remove doc-strings in addition to the -O optimizations.
-Q <i>arg</i>	Division options: -Qold (default), -Qwarn, -Qwarnall, -Qnew
-S	Don't perform <code>import site</code> on initialization.
-t	Issue warnings about inconsistent tab usage (-tt: issue errors).
-u	Unbuffered binary stdout and stderr (also PYTHONUNBUFFERED=x).
-U	Force Python to interpret all string literals as Unicode literals.
-v	Verbose (trace import statements) (also PYTHONVERBOSE=x).
-V	Print the Python version number and exit.
-W <i>arg</i>	Warning control (<i>arg</i> is action:message:category:module:lineno)
-x	Skip first line of source, allowing use of non-unix Forms of <code>#!/cmd</code>
-X	Disable class-based built-in exceptions (for backward compatibility management of exceptions)
-c <i>command</i>	Specify the command to execute (see next section). This terminates the option list (following options are passed as arguments to the command).
<i>scriptFile</i>	The name of a python file (.py) to execute. Read from stdin.
-	Program read from stdin (default; interactive mode if a tty).
<i>args</i>	Passed to script or command (in <code>sys.argv[1:]</code>)
	If no <i>scriptFile</i> or <i>command</i> , Python enters interactive mode.

- Available **IDEs** in std distrib: **IDLE** (tkinter based, portable), **Pythonwin** (on Windows). Other free IDEs: [BOA constructor](#), [SPE](#).
- 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.

Environment variables

Environment variables

Variable	Effect
PYTHONHOME	Alternate <i>prefix</i> directory (or <i>prefix;exec_prefix</i>). The default module search path uses <i>prefix/lib</i>
PYTHONPATH	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 first search for Registry key HKEY_LOCAL_MACHINE\Software\Python\PythonCore\X.Y\PythonPath (default value). You may also define a key named after your application with a default string value giving the root directory path of your app. Alternatively, you can create a text file in the Python home directory with a .pth extension, containing the path (one per line).
PYTHONSTARTUP	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).
PYTHONDEBUG	If non-empty, same as -d option
PYTHONINSPECT	If non-empty, same as -i option
PYTHONOPTIMIZE	If non-empty, same as -O option

PYTHONUNBUFFERED	If non-empty, same as -u option
PYTHONVERBOSE	If non-empty, same as -v option
PYTHONCASEOK	If non-empty, ignore case in file/module names (imports)

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	yield
def	finally	in	print	

- (List of keywords available in std module: **keyword**)
- Illegitimate Tokens (only valid in strings): @ \$?
- 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 by 'from module import *'); `__ident__` (system defined name); `__ident` (class-private name mangling).

String literals

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 3- double quotes as delimiters """"
u'a <u>unicode</u> string'
U"Another <u>unicode</u> string"
r'a <u>raw</u> string where \ are kept (literalized): handy for regular expressions and windows paths!
R"another raw string" -- raw strings cannot end with a \
ur'a <u>unicode</u> raw string'
UR"another raw <u>unicode</u> "

- 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

Escape	Meaning
<u>newline</u>	Ignored (escape newline)
\\	Backslash (\)
\e	Escape (ESC)
\v	Vertical Tab (VT)
\'	Single quote (')
\f	Formfeed (FF)
\ooo	char with octal value ooo
\"	Double quote (")
\n	Linefeed (LF)
\a	Bell (BEL)
\r	Carriage Return (CR)
\xhh	char with hex value hh
\b	Backspace (BS)
\t	Horizontal Tab (TAB)
\uxxxx	Character with 16-bit hex value xxxx (unicode only)
\Uxxxxxxxx	Character with 32-bit hex value xxxxxxxx (unicode only)
\N{name}	Character named in the Unicode database (unicode only), e.g. u'\N{Greek Small Letter Pi}' <=> u'\u03c0'.

	(Conversely, in module <code>unicodedata</code> , <code>unicodedata.name(u'\u03c0') == 'GREEK SMALL LETTER PI'</code>)
<code>\AnyOtherChar</code>	left as-is

- 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**

In 2.2, True and False are integers 1 and 0. Since 2.3, they are of new type `bool`.

Numbers

- **Decimal integer**: 1234, 1234567890546378940**L** (or **l**)
- **Octal integer**: **0**177, **0**1777777777777777777**L** (begin with a **0**)
- **Hex integer**: **0x**FF, **0X**FFFFFFfFfFfFfFfF**L** (begin with **0x** or **0X**)
- **Long integer** (unlimited precision): 1234567890123456**L** (ends with **L** or **l**) or **long**(1234)
- **Float** (double precision): 3.14**e-10**, .001, 10., 1E3
- **Complex**: 1**J**, 2+3**J**, 4+5**j** (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

- **String** of length 0, 1, 2 (see [above](#))
", '1', "12", 'hello\n'
- **Tuple** of length 0, 1, 2, etc:
() (1,) (1,2) # parentheses are optional if len > 0
- **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** [*starting-at-index* : *but-less-than-index* [: *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[2:4] == (2, 3)
a[1:] == (1, 2, 3, 4, 5, 6, 7)
a[:3] == (0, 1, 2)
a[:] == (0,1,2,3,4,5,6,7) # makes a copy of the sequence.
a[::2] == (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: {} {1 : 'first'} {1 : 'first', 'next': 'second'}

Operators and their evaluation order

Operators and their evaluation order

Highest	Operator	Comment
	, [...], {...} `...` s[i] s[i:j] s.attr f(...)	Tuple, list & dict. creation; string conv. indexing & slicing; attributes, fct calls
	+x, -x, ~x	Unary operators
	x**y	Power
	x*y x/y x%y	mult, division, modulo
	x+y x-y	addition, subtraction
	x<<y x>>y	Bit shifting
	x&y	Bitwise and
	x^y	Bitwise exclusive or
	x y	Bitwise or
	x<y x<=y x>y x>=y x==y x!=y x<>y x is y x is not y x in s x not in s not x x and y x or y	Comparison, identity, membership boolean negation boolean and boolean or
Lowest	lambda args: <i>expr</i>	anonymous function

- 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)

Comparisons

Comparison	Meaning	Notes
<	strictly less than	(1)
<=	less than or equal to	
>	strictly greater than	
>=	greater than or equal to	
==	equal to	
!= or <>	not equal to	
is	object identity	(2)
is not	negated object identity	(2)

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.

Boolean operators

Boolean values and operators

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

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 and long integers.

- 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`.

Operators on all numeric types

Operators on all numeric types

Operation	Result
abs (<i>x</i>)	the absolute value of <i>x</i>
int (<i>x</i>)	<i>x</i> converted to integer
long (<i>x</i>)	<i>x</i> converted to long integer
float (<i>x</i>)	<i>x</i> converted to floating point
- <i>x</i>	<i>x</i> negated
+ <i>x</i>	<i>x</i> unchanged
<i>x</i> + <i>y</i>	the sum of <i>x</i> and <i>y</i>
<i>x</i> - <i>y</i>	difference of <i>x</i> and <i>y</i>
<i>x</i> * <i>y</i>	product of <i>x</i> and <i>y</i>
<i>x</i> / <i>y</i>	true division of <i>x</i> by <i>y</i> : <code>1/2 -> 0.5</code> (1)
<i>x</i> // <i>y</i>	floor division operator: <code>1//2 -> 0</code> (1)

$x \% y$	remainder of x / y
<code>divmod(x, y)</code>	the tuple $(x/y, x\%y)$
$x ** y$	x to the power y (the same as <code>pow(x,y)</code>)

Notes:

- (1) / is still a *floor* division ($1/2 == 0$) unless validated by a `from __future__ import division`.
- classes may override methods `__truediv__` and `__floordiv__` to redefine these operators.

Bit operators on integers and long integers

Bit operators

Operation	Result
$\sim x$	the bits of x inverted
$x \wedge y$	bitwise exclusive or of x and y
$x \& y$	bitwise and of x and y
$x y$	bitwise or of x and y
$x \ll n$	x shifted left by n bits
$x \gg n$	x shifted right by n bits

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`

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

Operation	Result	Notes
$x \text{ in } s$	True if an item of s is equal to x , else False	(3)
$x \text{ not in } s$	False if an item of s is equal to x , else True	(3)
$s + t$	the concatenation of s and t	
$s * n, n * s$	n copies of s concatenated	
$s[i]$	i 'th item of s , origin 0	(1)
$s[i:j]$ $s[i:j:step]$	Slice of s from i (included) to j (excluded). Optional <i>step</i> value, possibly negative (default: 1).	(1), (2)
<code>len(s)</code>	length of s	
<code>min(s)</code>	smallest item of s	
<code>max(s)</code>	largest item of (s)	

Notes:

- (1) if i or j is negative, the index is relative to the end of the string, ie $\text{len}(s)+i$ or $\text{len}(s)+j$ is substituted. But note that -0 is still 0.
- (2) The slice of s from i to j is defined as the sequence of items with index k such that $i \leq k < j$.
If i or j is greater than $\text{len}(s)$, use $\text{len}(s)$. If i is omitted, use $\text{len}(s)$. If i is greater than or equal to j , the slice is empty.
- For strings: before 2.3, $x \text{ in } s$ must be a single character string; Since 2.3, $x \text{ in } s$ is True if x is a *substring* of s .

Operations on mutable sequences (type `list`)

Operations on mutable sequences

Operation	Result	Notes
$s[i] = x$	item i of s is replaced by x	
$s[i:j[:step]] = t$	slice of s from i to j is replaced by t	
<code>del s[i:j[:step]]</code>	same as $s[i:j] = []$	
<code>s.append(x)</code>	same as $s[\text{len}(s) : \text{len}(s)] = [x]$	
<code>s.extend(x)</code>	same as $s[\text{len}(s):\text{len}(s)] = x$	(5)
<code>s.count(x)</code>	return number of i 's for which $s[i] == x$	
<code>s.index(x[, start[, stop]])</code>	return smallest i such that $s[i] == x$. <i>start</i> and <i>stop</i> limit search to only part of the list.	(1)
<code>s.insert(i, x)</code>	same as $s[i:i] = [x]$ if $i \geq 0$. $i == -1$ inserts before the last element.	
<code>s.remove(x)</code>	same as <code>del s[s.index(x)]</code>	(1)
<code>s.pop([i])</code>	same as $x = s[i]$; <code>del s[i]</code> ; return x	(4)
<code>s.reverse()</code>	reverse the items of s in place	(3)
<code>s.sort(cmpFct)</code>	sort the items of s in place	(2), (3)

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 specifying a comparison function of 2 arguments (list items) which should return -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.
- (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.

Operations on mappings / dictionaries (type `dict`)

Operations on mappings

Operation	Result	Notes
<code>len(d)</code>	The number of items in d	
<code>dict(**kw)</code>	Creates a dictionary. Use keyword list kw as initialization list.	
<code>d.fromkeys(iterable, value=None)</code>	Class method to create a dictionary with keys provided by <i>iterator</i> , and all values set to <i>value</i> .	
<code>d[k]</code>	The item of d with key k	(1)
<code>d[k] = x</code>	Set $d[k]$ to x	
<code>del d[k]</code>	Remove $d[k]$ from d	(1)
<code>d.clear()</code>	Remove all items from d	
<code>d.copy()</code>	A shallow copy of d	
<code>d.has_key(k)</code> <code>k in d</code>	True if d has key k , else False	
<code>d.items()</code>	A copy of d 's list of (key, item) pairs	(2)
<code>d.keys()</code>	A copy of d 's list of keys	(2)
<code>d1.update(d2)</code>	for k, v in $d2.items()$: $d1[k] = v$	(3)
<code>d.values()</code>	A copy of d 's list of values	(2)
<code>d.get(k, defaultval)</code>	The item of d with key k	(4)
<code>d.setdefault(k[, defaultval])</code>	$d[k]$ if k in d , else <i>defaultval</i> (also setting it)	(5)
<code>d.iteritems()</code>	Returns an <u>iterator</u> over (key, value) pairs.	
<code>d.iterkeys()</code>	Returns an <u>iterator</u> over the mapping's keys.	
<code>d.itervalues()</code>	Returns an <u>iterator</u> over the mapping's values.	
<code>d.pop(k[, default])</code>	Removes key k and returns the corresponding value. If key is not found, <i>default</i> is returned if given, otherwise <code>KeyError</code> is raised.	
<code>d.popitem()</code>	Remove and return an arbitrary (key, value) pair from d	

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) $d2$ must be of the same type as $d1$
- (4) 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.
- (5) Never raises an exception if k is not in the map, instead it 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.

Operations on strings (type `str`)

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

Operation	Result	Notes
<code>s.capitalize()</code>	Returns a copy of s with only its first character capitalized.	
<code>s.center(width)</code>	Returns a copy of s centered in a string of length <i>width</i> .	(1)
<code>s.count(sub[, start[, end]])</code>	Returns the number of occurrences of substring sub in string s .	(2)
<code>s.encode([encoding[, errors]])</code>	Returns an encoded version of s . Default encoding is the current default string encoding.	(3)
<code>s.endswith(suffix [, start[, end]])</code>	Returns <code>True</code> if s ends with the specified <i>suffix</i> , otherwise return false.	(2)
<code>s.expandtabs([tabsize])</code>	Returns a copy of s where all tab characters are expanded using spaces.	(4)
<code>s.find(sub[, start[, end]])</code>	Returns the lowest index in s where substring sub is found. Return -1 if sub is not found.	(2)
<code>s.index(sub[, start[, end]])</code>	like <code>find()</code> , but raises <code>ValueError</code> when the substring is not found.	(2)
<code>s.isalnum()</code>	Returns <code>True</code> if all characters in s are alphanumeric, <code>False</code> otherwise.	(5)

<code>s.isalpha()</code>	Returns <code>True</code> if all characters in <code>s</code> are alphabetic, <code>False</code> otherwise.	(5)
<code>s.isdigit()</code>	Returns <code>True</code> if all characters in <code>s</code> are digit characters, <code>False</code> otherwise.	(5)
<code>s.islower()</code>	Returns <code>True</code> if all characters in <code>s</code> are lowercase, <code>False</code> otherwise.	(6)
<code>s.isspace()</code>	Returns <code>True</code> if all characters in <code>s</code> are whitespace characters, <code>False</code> otherwise.	(5)
<code>s.istitle()</code>	Returns <code>True</code> if string <code>s</code> is a titlecased string, <code>False</code> otherwise.	(7)
<code>s.isupper()</code>	Returns <code>True</code> if all characters in <code>s</code> are uppercase, <code>False</code> otherwise.	(6)
<code>separator.join(seq)</code>	Returns a concatenation of the strings in the sequence <code>seq</code> , separated by string <code>separator</code> , e.g.: <code>" ".join(['A', 'B', 'C']) -> "A, B, C"</code>	
<code>s.ljust(width)</code>	Returns <code>s</code> left justified in a string of length <code>width</code> .	(1), (8)
<code>s.lower()</code>	Returns a copy of <code>s</code> converted to lowercase.	
<code>s.lstrip([chars])</code>	Returns a copy of <code>s</code> with leading <code>chars</code> (default: whitespaces) removed.	
<code>s.replace(old, new[, maxsplit])</code>	Returns a copy of <code>s</code> with all occurrences of substring <code>old</code> replaced by <code>new</code> .	(9)
<code>s.rfind(sub[, start[, end]])</code>	Returns the highest index in <code>s</code> where substring <code>sub</code> is found. Returns <code>-1</code> if <code>sub</code> is not found.	(2)
<code>s.rindex(sub[, start[, end]])</code>	like <code>rfind()</code> , but raises <code>ValueError</code> when the substring is not found.	(2)
<code>s.rjust(width)</code>	Returns <code>s</code> right justified in a string of length <code>width</code> .	(1), (8)
<code>s.rstrip([chars])</code>	Returns a copy of <code>s</code> with trailing <code>chars</code> (default: whitespaces) removed.	
<code>s.split([separator[, maxsplit]])</code>	Returns a list of the words in <code>s</code> , using <code>separator</code> as the delimiter string.	(10)
<code>s.splitlines([keepends])</code>	Returns a list of the lines in <code>s</code> , breaking at line boundaries.	(11)
<code>s.startswith(prefix[, start[, end]])</code>	Returns <code>True</code> if <code>s</code> starts with the specified <code>prefix</code> , otherwise returns <code>False</code> . Negative numbers may be used for <code>start</code> and <code>end</code>	(2)
<code>s.strip([chars])</code>	Returns a copy of <code>s</code> with leading and trailing <code>chars</code> (default: whitespaces) removed.	
<code>s.swapcase()</code>	Returns a copy of <code>s</code> with uppercase characters converted to lowercase and vice versa.	
<code>s.title()</code>	Returns a titlecased copy of <code>s</code> , i.e. words start with uppercase characters, all remaining cased characters are lowercase.	
<code>s.translate(table[, deletechars])</code>	Returns a copy of <code>s</code> mapped through translation table <code>table</code> .	(12)
<code>s.upper()</code>	Returns a copy of <code>s</code> converted to uppercase.	
<code>s.zfill(width)</code>	Returns the numeric string left filled with zeros in a string of length <code>width</code> .	

Notes:

- (1) Padding is done using spaces.
- (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.
- (3) 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'**.
- (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 `maxsplit` is given, only the first `maxsplit` occurrences are replaced.
- (10) If `sep` 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. All characters occurring in the optional argument `deletechars` are removed prior to translation.

String formatting with the % operator

`formatString % args --> evaluates to a string`

- `formatString` uses C printf format codes : %, c, s, i, d, u, o, x, X, e, E, f, g, G, r (details below).
- Width and precision may be a * to specify that an integer argument gives the actual width or precision.
- The flag characters -, +, blank, # and 0 are understood (details below).
- %s will convert any type argument to string (uses `str()` function)
- `args` may be a single arg or a tuple of args


```
's has %03d quote types.' % ('Python', 2) == 'Python has 002 quote types.'
```
- Right-hand-side can also be a *mapping*:


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

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

Format codes

Code	Meaning
d	Signed integer decimal.
i	Signed integer decimal.
o	Unsigned octal.
u	Unsigned decimal.
x	Unsigned hexadecimal (lowercase).
X	Unsigned hexadecimal (uppercase).

e	Floating point exponential format (lowercase).
E	Floating point exponential format (uppercase).
f	Floating point decimal format.
F	Floating point decimal format.
g	Same as "e" if exponent is greater than -4 or less than precision, "f" otherwise.
G	Same as "E" if exponent is greater than -4 or less than precision, "F" otherwise.
c	Single character (accepts integer or single character string).
r	String (converts any python object using <code>repr()</code>).
s	String (converts any python object using <code>str()</code>).
%	No argument is converted, results in a "%" character in the result. (The complete specification is %%.)

Conversion flag characters

Flag	Meaning
#	The value conversion will use the "alternate form".
0	The conversion will be zero padded.
-	The converted value is left adjusted (overrides "-").
	(a space) A blank should be left before a positive number (or empty string) produced by a signed conversion.
+	A sign character ("+" or "-") will precede the conversion (overrides a "space" flag).

File objects

(Type `file`). Created with built-in functions `open()` [deprecated] or `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
<code>f.close()</code>	Close file <i>f</i> .
<code>f.fileno()</code>	Get fileno (fd) for file <i>f</i> .
<code>f.flush()</code>	Flush file <i>f</i> 's internal buffer.
<code>f.isatty()</code>	1 if file <i>f</i> is connected to a tty-like dev, else 0.
<code>f.read([size])</code>	Read at most <i>size</i> bytes from file <i>f</i> and return as a string object. If <i>size</i> omitted, read to EOF.
<code>f.readline()</code>	Read one entire line from file <i>f</i> .
<code>f.readlines()</code>	Read until EOF with <code>readline()</code> and return list of lines read.
<code>f.xreadlines()</code>	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...	Iterate over the lines of a file (using readline)
<code>f.seek(offset[, whence=0])</code>	Set file <i>f</i> 's position, like "stdio's fseek()". <i>whence</i> == 0 then use absolute indexing. <i>whence</i> == 1 then offset relative to current pos. <i>whence</i> == 2 then offset relative to file end.
<code>f.tell()</code>	Return file <i>f</i> 's current position (byte offset).
<code>f.write(str)</code>	Write string to file <i>f</i> .
<code>f.writelines(list)</code>	Write list of strings to file <i>f</i> .

File Exceptions

`EOFError`

End-of-file hit when reading (may be raised many times, e.g. if *f* is a tty).

`IOError`

Other I/O-related I/O operation failure

Sets

The Module `sets` define a new `set` data type implementation, via class `Set` for mutable sets, and class `ImmutableSet` for non mutable sets. Sets are unordered collections of unique (non duplicate) elements. Elements must be hashable.

Main Set operations

Operation	Result
<code>Set/ImmutableSet([iterable=None])</code>	Builds a <code>Set</code> or <code>ImmutableSet</code> from the given <i>iterable</i> (default: empty), e.g. <code>Set([1, 2, 3])</code> .
<code>elt in S</code>	<code>True</code> if element <i>elt</i> belongs to <i>S</i> .
<code>S.add(elt)</code>	Adds element <i>elt</i> to <i>S</i> (if it doesn't already exist).
<code>S.remove(elt)</code>	Removes element <i>elt</i> from <i>S</i> (if it doesn't already exist). <code>KeyError</code> if element not found.
<code>S1.intersection(S2)</code>	Returns the (Set) intersection of sets <i>S1</i> and <i>S2</i> . Also noted <i>S1</i> & <i>S2</i> .
<code>S1.union(S2)</code>	Returns the (Set) union of sets <i>S1</i> and <i>S2</i> . Also noted <i>S1</i> <i>S2</i> .
<code>S1.symmetric_difference(S2)</code>	Returns the (Set) difference of sets <i>S1</i> and <i>S2</i> . Also noted <i>S1</i> ^ <i>S2</i> .

<code>S1.issubset(S2)</code>	True if <i>S1</i> is a subset of <i>S2</i> .
<code>S1.issuperset(S2)</code>	True if <i>S1</i> is a superset of <i>S2</i> .

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

Statements

Statement	Result
pass	Null statement
del <i>name</i> [, <i>name</i>]*	Unbind <i>name</i> (s) from object. Object will be indirectly (and automatically) deleted only if no longer referenced.
print [>> <i>fileobject</i>][, [<i>s1</i> [, <i>s2</i>]]* [,]]	Writes to <code>sys.stdout</code> , or to <i>fileobject</i> if supplied. Puts spaces between arguments. Puts newline at end unless statement ends with comma . Print is not required when running interactively, simply typing an expression will print its value, unless the value is <code>None</code> .
exec <i>x</i> [in <i>globals</i> [, <i>locals</i>]]	Executes <i>x</i> in namespaces provided. Defaults to current namespaces. <i>x</i> can be a string, file object or a function object.
<i>callable</i> (<i>value</i> ,... [<i>id=</i> <i>value</i>] [, [<i>*args</i>] [, [<i>**kw</i>]])	Call function <i>callable</i> with parameters. Parameters can be passed by name or be omitted if function defines default values. E.g. if <i>callable</i> is defined as <code>"def callable(p1=1, p2=2)"</code> <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <pre>"callable()" <=> "callable(1, 2)" "callable(10)" <=> "callable(10, 2)" "callable(p2=99)" <=> "callable(1, 99)"</pre> </div> <i>*args</i> is a tuple of positional arguments. <i>**kw</i> is a dictionary of keyword arguments.
yield <i>expression</i>	(Only used within the body of a generator function, outside a try of a <code>try..finally</code>). "Returns" the evaluated <i>expression</i> .

Assignment operators

Assignment operators

Operator	Result	Notes
<code>a = b</code>	Basic assignment - assign object <i>b</i> to label <i>a</i>	(1)
<code>a += b</code>	Roughly equivalent to <code>a = a + b</code>	(2)
<code>a -= b</code>	Roughly equivalent to <code>a = a - b</code>	(2)
<code>a *= b</code>	Roughly equivalent to <code>a = a * b</code>	(2)
<code>a /= b</code>	Roughly equivalent to <code>a = a / b</code>	(2)
<code>a //= b</code>	Roughly equivalent to <code>a = a // b</code>	(2)
<code>a %= b</code>	Roughly equivalent to <code>a = a % b</code>	(2)
<code>a **= b</code>	Roughly equivalent to <code>a = a ** b</code>	(2)
<code>a &= b</code>	Roughly equivalent to <code>a = a & b</code>	(2)
<code>a = b</code>	Roughly equivalent to <code>a = a b</code>	(2)
<code>a ^= b</code>	Roughly equivalent to <code>a = a ^ b</code>	(2)

<code>a >>= b</code>	Roughly equivalent to <code>a = a >> b</code>	(2)
<code>a <<= b</code>	Roughly equivalent to <code>a = a << b</code>	(2)

Notes:

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

```
first, second = a[0:2]
[f, s] = range(2)
c1, c2, c3 = 'abc'
```

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

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

Control flow statements

Statement	Result
if <i>condition</i> : <i>suite</i> [elif <i>condition</i> : <i>suite</i>]* [else : <i>suite</i>]	Usual if/else if/else statement
while <i>condition</i> : <i>suite</i> [else : <i>suite</i>]	Usual while statement. The <i>else suite</i> is executed after loop exits, unless the loop is exited with <code>break</code> .
for <i>element in sequence</i> : <i>suite</i> [else : <i>suite</i>]	Iterates over <i>sequence</i> , assigning each element to <i>element</i> . Use built-in <code>range</code> function to iterate a number of times. The <i>else suite</i> is executed at end unless loop exited with <code>break</code> .
break	Immediately exits <code>for</code> or <code>while</code> loop.
continue	Immediately does next iteration of <code>for</code> or <code>while</code> loop.
return [<i>result</i>]	Exits from function (or method) and returns <i>result</i> (use a tuple to return more than one value). If no result given, then returns <code>None</code> .

Exception statements

Statement	Result
assert <i>expr</i> [, <i>message</i>]	<i>expr</i> is evaluated. if false, raises exception <code>AssertionError</code> with message. Before 2.3, inhibited if <code>__debug__</code> is 0.
try : <i>suite1</i> [except [<i>exception</i> [, <i>value</i>]: <i>suite2</i>]+ [else : <i>suite3</i>]	Statements in <i>suite1</i> are executed. If an exception occurs, look in <code>except</code> clauses for matching <i>exception</i> . If matches or bare <code>except</code> , execute <i>suite</i> of that clause. If no exception happens, <i>suite</i> in <code>else</code> clause is executed after <i>suite1</i> . If <i>exception</i> has a value, it is put in variable <i>value</i> . <i>exception</i> can also be tuple of exceptions, e.g. <code>except (KeyError, NameError), val: print val</code> .
try : <i>suite1</i> finally : <i>suite2</i>	Statements in <i>suite1</i> are executed. If no exception, execute <i>suite2</i> (even if <i>suite1</i> is exited with a <code>return</code> , <code>break</code> or <code>continue</code> statement). If exception did occur, executes <i>suite2</i> and then immediately re-raises exception.
raise <i>exceptionInstance</i>	Raises an instance of a class derived from <code>Exception</code> (preferred form of raise).
raise <i>exceptionClass</i> [, <i>value</i> [, <i>traceback</i>]]	Raises <i>exception</i> of given class <i>exceptionClass</i> with optional value <i>value</i> . Arg <i>traceback</i> specifies a <code>traceback</code> object to use when printing the exception's backtrace.
raise	A raise statement without arguments re-raises the last exception raised in the current function.

- An exception is an *instance* of an *exception class* (before 2.0, it may also be a mere *string*).
- 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`.

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, "theZipFile.zip")`].

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

Statement	Result
import <i>module1</i> [as <i>name1</i>] [, <i>module2</i>]*	Imports modules. Members of module must be referred to by qualifying with <code>[package.]module</code> name, e.g.: <pre>import sys; print sys.argv import package1.subpackage.module package1.subpackage.module.foo()</pre> <i>module1</i> renamed as <i>name1</i> , if supplied.
from <i>module</i> import <i>name1</i> [as <i>othername1</i>] [, <i>name2</i>]*	Imports names from module <i>module</i> in current namespace. <pre>from sys import argv; print argv from package1 import module; module.foo() from package1.module import foo; foo()</pre> <i>name1</i> renamed as <i>othername1</i> , if supplied.
from <i>module</i> import *	Imports all names in <i>module</i> , except those starting with "_" Use sparsely, beware of name clashes! <pre>from sys import *; print argv from package.module import *; print x</pre> Only legal at the top level of a module. If <i>module</i> defines an <code>__all__</code> attribute, only names listed in <code>__all__</code> will be imported. NB: "from <i>package</i> import *" only imports the symbols defined in the package's <code>__init__.py</code> file, not those in the package's modules !
global <i>name1</i> [, <i>name2</i>]	Names are from global scope (usually meaning from module) rather than local (usually 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" -> <code>NameError</code> - Try to write to "x" -> creates "x" local to function If "x" not defined in fct, but is in module, then: - Try to read from "x", gets value from module - Try to write to "x", creates "x" local to fct But note "x[0]=3" starts with search for "x", will use to global "x" if no local "x".

Function Definition

```
def func_id ([param_list]):
    suite
```

Creates a function object and binds it to name *func_id*.

```
param_list ::= [id [, id]*]
id ::= value | id = value | *id | **id
```

Args are passed by **value**. Thus only args representing a *mutable* object can be modified (are *inout* parameters). Use a **tuple** to return more than one value.

Example:

```
def test (p1, p2 = 1+1, *rest, **keywords):
```

Parameters with "=" have default value (v is evaluated when function defined).

If list has "**id*" then *id* is assigned a tuple of all remaining args passed to function (like C *vararg*) If list has "***id*" then *id* is assigned a dictionary of all extra arguments passed as keywords.

Class Definition

```
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".

```
class MyClass (object): ...
```

Creates a *new-style* class/type (inheriting from `object` makes a class a *new-style* class). 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 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!)
- Before 2.2 it was not possible to subclass built-in classes like list, dict (you had to "wrap" them, using UserDict & UserList modules); since 2.2 you can subclass them directly (see [Types/Classes unification](#)).

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:

```
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:

```
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 enumerates their lines.
- Python uses implicitly 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 new 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.
- 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.
- In 2.2, feature needs to be **enabled** by the statement: `from __future__ import generators` (not required since 2.3+)

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 that **describe attribute and methods of a class**, and allow their manipulation via methods `__get__`, `__set__`, etc... Python now transparently uses descriptors to access members.
- By using special descriptors it is now possible to define:
 - **Static methods** : like static in C => Use `staticmethod(f)` to make method `f(x)` static.
 - **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.
 - **Properties** (in Delphi/Corba/JavaBean style). A *property* is an instance of the new built-in type `property`, which implements the *descriptor* protocol for attributes => Use `propertyName = property(getter=None, setter=None, deleter=None, description=None)` to define a property inside or outside a class.
 - **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.

Misc

```
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

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

is equivalent to:

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

Nested scopes

In 2.2 *nested scopes* no longer need to be specially enabled by a `from __future__ import nested_scopes` directive, and are now always present.

Built-In Functions

Built-In Functions

Function	Result
<code>import_(name[, globals[, locals[, from list]])</code>	Imports module within the given context (see library reference for more details)
<code>abs(x)</code>	Returns the absolute value of number <i>x</i> .
<code>apply(f, args[, keywords])</code>	Calls func/method <i>f</i> with arguments <i>args</i> and optional keywords.
<code>buffer(object[, offset[, size]])</code>	Returns a <code>Buffer</code> from a slice of <i>object</i> , which must support the buffer call interface (String, array, buffer).

callable(x)	Returns 1 if <i>x</i> callable, else 0.
chr(i)	Returns one-character string whose ASCII code is integer <i>i</i> .
classmethod(function)	<p>Returns a class method for <i>function</i>. 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:</p> <pre>class C: def f(cls, arg1, arg2, ...): ... f = classmethod(f)</pre> <p>Then call it on the class <i>C.f()</i> or on an instance <i>C().f()</i>. The instance is ignored except for its class. If a class method is called for a derived class, the derived class object is passed as the implied first argument.</p>
cmp(x,y)	Returns negative, 0, positive if <i>x</i> <, ==, > to <i>y</i>
coerce(x,y)	Returns a tuple of the two <i>numeric</i> arguments converted to a common type.
compile(string, filename, kind[, flags[, dont_inherit]])	Compiles <i>string</i> into a code object. <i>filename</i> is used in error message, can be any string. It is usually the file from which the code was read, or eg. '<string>' if not read from file. <i>kind</i> can be 'eval' if <i>string</i> 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 <i>flags</i> and <i>dont_inherit</i> concern <i>future</i> statements.
complex(real[, image])	Creates a complex object (can also be done using <i>J</i> or <i>j</i> suffix, e.g. 1+3J).
delattr(obj, name)	Deletes attribute named <i>name</i> of object <i>obj</i> <=> <code>del obj.name</code>
dict([mapping-or-sequence])	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).
dir([object])	Without args, returns the list of names in the current local symbol table. With a module, class or class instance object as <i>arg</i> , returns the list of names in its attr. dictionary.
divmod(a,b)	Returns tuple of (<i>a/b</i> , <i>a%b</i>)
enumerate(iterable)	Iterator returning pairs (index, value) of <i>iterable</i> , e.g. <code>List(enumerate('Py')) -> [(0, 'P'), (1, 'y')]</code> .
eval(s[, globals[, locals]])	Evaluate string <i>s</i> in (optional) <i>globals</i> , <i>locals</i> contexts. <i>s</i> must have no NUL's or newlines. <i>s</i> can also be a code object. Example: <pre>x = 1; incr_x = eval('x + 1')</pre>
execfile(file[, globals[, locals]])	Executes a file without creating a new module, unlike <code>import</code> .
file(filename[, mode[, bufsize]])	<p>Opens a file and returns a new file object. Replaces <code>open</code>.</p> <ul style="list-style-type: none"> <i>filename</i> is the file name to be opened <i>mode</i> indicates how the file is to be opened: <ul style="list-style-type: none"> 'r' for reading 'w' for writing (truncating an existing file) 'a' opens it for appending '+' (appended to any of the previous modes) open the file for updating (note that 'w+' truncates the file) 'b' (appended to any of the previous modes) open the file in binary mode 'U' (or 'rU') open the file for reading in <i>Universal Newline mode</i>: all variants of EOL (CR, LF, CR+LF) will be translated to a single LF ('\n'). <i>bufsize</i> is 0 for unbuffered, 1 for line-buffered, negative for sys-default, all else, of (about) given size.
filter(function, sequence)	Constructs a list from those elements of <i>sequence</i> for which <i>function</i> returns true. <i>function</i> takes one parameter.
float(x)	Converts a number or a string to floating point.
getattr(object, name[, default])	Gets attribute called <i>name</i> from <i>object</i> , e.g. <code>getattr(x, 'f') <=> x.f</code> . If not found, raises <code>AttributeError</code> or returns <i>default</i> if specified.
globals()	Returns a dictionary containing the current global variables.
hasattr(object, name)	Returns true if <i>object</i> has an attribute called <i>name</i> .
hash(object)	Returns the hash value of the object (if it has one).
help([object])	Invoke the built-in help system. No argument -> interactive help; if <i>object</i> 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 <i>object</i> is generated.
hex(x)	Converts a number <i>x</i> to a hexadecimal string.
id(object)	Returns a unique integer identifier for <i>object</i> .
input([prompt])	Prints <i>prompt</i> if given. Reads input and evaluates it. Uses line editing / history if module <code>readline</code> available.
int(x[, base])	Converts a number or a string to a plain integer. Optional <i>base</i> parameter specifies base from which to convert string values.
intern(aString)	Enters <i>aString</i> in the table of interned strings and returns the string. Before 2.3, interned strings were 'immortals' (never garbage collected). This is no longer true in 2.3+.
isinstance(obj, classInfo)	Returns true if <i>obj</i> is an instance of class <i>classInfo</i> or an object of type <i>classInfo</i> (<i>classInfo</i> may also be a tuple of classes or types). If <code>issubclass(A, B)</code> then <code>isinstance(x, A) => isinstance(x, B)</code>
issubclass(class1, class2)	Returns true if <i>class1</i> is derived from <i>class2</i> (or if <i>class1</i> is <i>class2</i>).

iter(obj[,sentinel])	Returns an iterator on <i>obj</i> . If <i>sentinel</i> is absent, <i>obj</i> must be a collection implementing either <code>__iter__()</code> or <code>__getitem__()</code> . If <i>sentinel</i> is given, <i>obj</i> will be called with no arg; if the value returned is equal to <i>sentinel</i> , <code>StopIteration</code> will be raised, otherwise the value will be returned. See Iterators .
len(obj)	Returns the length (the number of items) of an object (sequence, dictionary, or instance of class implementing <code>__len__</code>).
list(sequence)	Converts <i>sequence</i> into a list. If already a list, returns a 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 <i>base</i> parameter specifies the base from which to convert string values.
map(function, list, ...)	Applies <i>function</i> to every item of <i>list</i> and returns a list of the results. If additional arguments are passed, <i>function</i> must take that many arguments and they are given to <i>function</i> on each call.
max(seq[, args...])	With a single argument <i>seq</i> , returns the largest item of a non-empty sequence (such as a string, tuple or list). With more than one argument, returns the largest of the arguments.
min(seq[, args...])	With a single argument <i>seq</i> , returns the smallest item of a non-empty sequence (such as a string, tuple or list). With more than one argument, returns the smallest of the arguments.
oct(x)	Converts a number to an octal string.
open(filename [, mode='r', [bufsize=implementation dependent]])	Returns a new file object. Alias for the new function file().
ord(c)	Returns integer ASCII value of <i>c</i> (a string of len 1). Works with Unicode char.
pow(x, y [, z])	Returns <i>x</i> to power <i>y</i> [modulo <i>z</i>]. See also <code>**</code> operator.
range(start [,end [, step]])	Returns list of ints from \geq start and $<$ end. With 1 arg, list from 0.. <i>arg</i> -1 With 2 args, list from <i>start</i> .. <i>end</i> -1 With 3 args, list from <i>start</i> up to <i>end</i> by <i>step</i>
raw_input([prompt])	Prints <i>prompt</i> if given, then reads string from std input (no trailing <code>\n</code>). See also input() .
reduce(f, list [, init])	Applies the binary function <i>f</i> to the items of <i>list</i> so as to reduce the list to a single value. If <i>init</i> is given, it is "prepended" to <i>list</i> .
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 <code>reload()</code> .
repr(object)	Returns a string containing a printable and if possible evaluable representation of an object. <code><=></code> <code>`object`</code> (using backquotes). Class redefinable (<code>__repr__</code>). See also str()
round(x, n=0)	Returns the floating point value <i>x</i> rounded to <i>n</i> digits after the decimal point.
setattr(object, name, value)	This is the counterpart of getattr() . <code>setattr(o, 'foobar', 3) <=> o.foobar = 3</code> . Creates attribute if it doesn't exist!
slice([start,] stop[, step])	Returns a <i>slice object</i> representing a range, with R/O attributes: start, stop, step.
staticmethod(function)	Returns a static method for <i>function</i>. A static method does not receive an implicit first argument. To declare a static method, use this idiom: <pre>class C: def f(arg1, arg2, ...): ... f = staticmethod(f)</pre> Then call it on the class <code>C.f()</code> or on an instance <code>C().f()</code>. The instance is ignored except for its class.
str(object)	Returns a string containing a nicely printable representation of an object. Class overridable (<code>__str__</code>). See also repr() .
sum(iterable[, start=0])	Returns the sum of a sequence of numbers (not strings), plus the value of parameter. Returns <i>start</i> when the sequence is empty.
super(type[, object-or-type])	Returns the superclass of <i>type</i> . If the second argument is omitted the super object returned is unbound. If the second argument is an object, <code>isinstance(obj, type)</code> must be true. If the second argument is a type, <code>issubclass(type2, type)</code> must be true. Typical use : <pre>class C(B): def meth(self, arg): super(C, self).meth(arg)</pre>
tuple([sequence])	Creates an empty tuple or a tuple with same elements as <i>sequence</i> . <i>sequence</i> may be a sequence, a container that supports iteration, or an iterator object. If <i>sequence</i> is already a tuple, return itself (not a copy).
type(obj)	Returns a <i>type object</i> [see module <i>types</i>] representing the type of <i>obj</i> . Example: <code>import types if type(x) == types.StringType: print 'It is a string'</code> . NB: it is better to use instead: <code>if isinstance(x, types.StringType)...</code>
unichr(code)	Returns a unicode string 1 char long with given <i>code</i> .
unicode(string[, encoding [,error]]])	Creates a Unicode string from a 8-bit string, using the given encoding name and error treatment ('strict', 'ignore', or 'replace'). For objects which provide a <code>__unicode__()</code> method, it will call this method without arguments to create a Unicode string.
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 <code>"%"</code> string formatting operator.

xrange (start [, end [, step]])	Like range(), but doesn't actually store entire list all at once. Good to use in "for" loops when there is a big range and little memory.
zip (seq1[, seq2,...])	Returns a list of tuples where each tuple contains the <i>n</i> th element of each of the argument sequences.

Built-In Exception classes

Exception

The mother of all exceptions. `exception.args` is a tuple of the arguments passed to the constructor.

- **StopIteration**
Raised by an iterator's `next()` method to signal that there are no further values.
- **SystemExit**
On `sys.exit()`
- **Warning**
Base class for warnings (see module `warning`)
 - **UserWarning**
Warning generated by user code.
 - **PendingDeprecationWarning**
Warning about future deprecated code.
 - **DeprecationWarning**
Warning about deprecated code.
 - **SyntaxWarning**
Warning about dubious syntax.
 - **RuntimeWarning**
Warning about dubious runtime behavior.
- **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** [new in 1.5.2]
On error outside Python; error arg. tuple is (`errno`, `errmsg`...)
 - **IOError** [changed in 1.5.2]
I/O-related operation failure.
 - **OSError** [new in 1.5.2]
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**
On user entry of the interrupt key (often 'CTRL-C')
- **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** [new in 1.5.2]

On method not implemented.

- **SyntaxError**

On parser encountering a syntax error

- **IndentationError**

On parser encountering an indentation syntax error

- **TabError**

On parser encountering an indentation syntax error

- **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.

- **UnicodeError**

On Unicode-related encoding or decoding error.

Standard methods & operators redefinition in classes

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

```
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

Method	Description
<code>__init__(self, args)</code>	Instance initialization (on construction)
<code>__del__(self)</code>	Called on object demise (refcount becomes 0)
<code>__repr__(self)</code>	<code>repr()</code> and <code>`...`</code> conversions
<code>__str__(self)</code>	<code>str()</code> and <code>print</code> statement
<code>__cmp__(self, other)</code>	Compares <i>self</i> to <i>other</i> and returns <0, 0, or >0. Implements >, <, == etc...
<code>__lt__(self, other)</code>	Called for <i>self</i> < <i>other</i> comparisons. Can return anything, or can raise an exception.
<code>__le__(self, other)</code>	Called for <i>self</i> <= <i>other</i> comparisons. Can return anything, or can raise an exception.
<code>__gt__(self, other)</code>	Called for <i>self</i> > <i>other</i> comparisons. Can return anything, or can raise an exception.
<code>__ge__(self, other)</code>	Called for <i>self</i> >= <i>other</i> comparisons. Can return anything, or can raise an exception.
<code>__eq__(self, other)</code>	Called for <i>self</i> == <i>other</i> comparisons. Can return anything, or can raise an exception.
<code>__ne__(self, other)</code>	Called for <i>self</i> != <i>other</i> (and <i>self</i> <> <i>other</i>) comparisons. Can return anything, or can raise an exception.
<code>__hash__(self)</code>	Compute a 32 bit hash code; <code>hash()</code> and dictionary ops
<code>__nonzero__(self)</code>	Returns 0 or 1 for truth value testing. when this method is not defined, <code>__len__()</code> is called if defined; otherwise all class instances are considered "true".
<code>__getattr__(self, name)</code>	Called when attribute lookup doesn't find <i>name</i> . See also <code>__getattribute__</code> .
<code>__getattribute__(self, name)</code>	Same as <code>__getattr__</code> but always called whenever the attribute <i>name</i> is accessed.
<code>__setattr__(self, name, value)</code>	Called when setting an attribute (inside, don't use " <i>self.name</i> = <i>value</i> ", use instead " <i>self.__dict__[name]</i> = <i>value</i> ")
<code>__delattr__(self, name)</code>	Called to delete attribute < <i>name</i> >.
<code>__call__(self, *args, **kwargs)</code>	Called when an instance is called as function: <code>obj(arg1, arg2, ...)</code> is a shorthand for <code>obj.__call__(arg1, arg2, ...)</code> .

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

Operator	Special method
<i>self</i> + <i>other</i>	<code>__add__(self, other)</code>
<i>self</i> - <i>other</i>	<code>__sub__(self, other)</code>
<i>self</i> * <i>other</i>	<code>__mul__(self, other)</code>
<i>self</i> / <i>other</i>	<code>__div__(self, other)</code> or <code>__truediv__(self, other)</code> if <code>__future__.division</code> is active.
<i>self</i> // <i>other</i>	<code>__floordiv__(self, other)</code>
<i>self</i> % <i>other</i>	<code>__mod__(self, other)</code>
<code>divmod(self, other)</code>	<code>__divmod__(self, other)</code>
<i>self</i> ** <i>other</i>	<code>__pow__(self, other)</code>

<i>self</i> & <i>other</i>	<code>__and__(self, other)</code>
<i>self</i> ^ <i>other</i>	<code>__xor__(self, other)</code>
<i>self</i> <i>other</i>	<code>__or__(self, other)</code>
<i>self</i> << <i>other</i>	<code>__lshift__(self, other)</code>
<i>self</i> >> <i>other</i>	<code>__rshift__(self, other)</code>
<code>nonzero(self)</code>	<code>__nonzero__(self)</code> (used in boolean testing)
<code>-self</code>	<code>__neg__(self)</code>
<code>+self</code>	<code>__pos__(self)</code>
<code>abs(self)</code>	<code>__abs__(self)</code>
<code>~self</code>	<code>__invert__(self)</code> (bitwise)
<i>self</i> += <i>other</i>	<code>__iadd__(self, other)</code>
<i>self</i> -= <i>other</i>	<code>__isub__(self, other)</code>
<i>self</i> *= <i>other</i>	<code>__imul__(self, other)</code>
<i>self</i> /= <i>other</i>	<code>__idiv__(self, other)</code> or <code>__itruediv__(self, other)</code> if <code>__future__.division</code> is in effect.
<i>self</i> // <i>other</i>	<code>__ifloordiv__(self, other)</code>
<i>self</i> %= <i>other</i>	<code>__imod__(self, other)</code>
<i>self</i> **= <i>other</i>	<code>__ipow__(self, other)</code>
<i>self</i> &= <i>other</i>	<code>__iand__(self, other)</code>
<i>self</i> ^= <i>other</i>	<code>__ixor__(self, other)</code>
<i>self</i> = <i>other</i>	<code>__ior__(self, other)</code>
<i>self</i> <<= <i>other</i>	<code>__ilshift__(self, other)</code>
<i>self</i> >>= <i>other</i>	<code>__irshift__(self, other)</code>

Conversions

built-in function	Special method
<code>int(self)</code>	<code>__int__(self)</code>
<code>long(self)</code>	<code>__long__(self)</code>
<code>float(self)</code>	<code>__float__(self)</code>
<code>complex(self)</code>	<code>__complex__(self)</code>
<code>oct(self)</code>	<code>__oct__(self)</code>
<code>hex(self)</code>	<code>__hex__(self)</code>
<code>coerce(self, other)</code>	<code>__coerce__(self, other)</code>

Right-hand-side equivalents for all binary operators exist; 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

Operation	Special method	Notes
All sequences and maps :		
<code>len(self)</code>	<code>__len__(self)</code>	length of object, >= 0. Length 0 == false
<code>self[k]</code>	<code>__getitem__(self, k)</code>	Get element at indice /key k (indice starts at 0). Or, if <i>k</i> is a slice object, return a slice.
<code>self[k] = value</code>	<code>__setitem__(self, k, value)</code>	Set element at indice/key/slice <i>k</i> .
<code>del self[k]</code>	<code>__delitem__(self, k)</code>	Delete element at indice/key/slice <i>k</i> .
<code>elt in self</code> <code>elt not in self</code>	<code>__contains__(self, elt)</code> <code>__notcontains__(self, elt)</code>	More efficient than std iteration thru sequence.
<code>iter(self)</code>	<code>__iter__(self)</code>	Returns an iterator on elements (keys for mappings <=> <code>self.iterkeys()</code>). See iterators .
Sequences, general methods, plus:		
<code>self[i:j]</code>	<code>__getslice__(self, i, j)</code>	Deprecated since 2.0, replaced by <code>__getitem__</code> with a slice object as parameter.
<code>self[i:j] = seq</code>	<code>__setslice__(self, i, j, seq)</code>	Deprecated since 2.0, replaced by <code>__setitem__</code> with a slice object as parameter.
<code>del self[i:j]</code>	<code>__delslice__(self, i, j)</code>	Same as <code>self[i:j] = []</code> - Deprecated since 2.0, replaced by <code>__delitem__</code> with a slice object as parameter.
<code>self * n</code>	<code>__repeat__(self, n)</code>	
<code>self + other</code>	<code>__concat__(self, other)</code>	
Mappings, general methods, plus:		
<code>hash(self)</code>	<code>__hash__(self)</code>	hashed value of object <i>self</i> is used for dictionary keys

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 <code>dir()</code> instead

Modules

Attribute	Meaning
<code>__doc__</code>	(string/None, R/O): doc string (<code><=> __dict__['__doc__']</code>)
<code>__name__</code>	(string, R/O): module name (also in <code>__dict__['__name__']</code>)
<code>__dict__</code>	(dict, R/O): module's name space
<code>__file__</code>	(string/undefined, R/O): pathname of .pyc, .pyo or .pyd (undef for modules statically linked to the interpreter)
<code>__path__</code>	(list/undefined, R/W): List of directory paths where to find the package (for packages only).

Classes

Attribute	Meaning
<code>__doc__</code>	(string/None, R/W): doc string (<code><=> __dict__['__doc__']</code>)
<code>__name__</code>	(string, R/W): class name (also in <code>__dict__['__name__']</code>)
<code>__bases__</code>	(tuple, R/W): parent classes
<code>__dict__</code>	(dict, R/W): attributes (class name space)

Instances

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

User defined functions

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

User-defined Methods

Attribute	Meaning
<code>__doc__</code>	(string/None, R/O): doc string
<code>__name__</code>	(string, R/O): method name (same as <code>im_func.__name__</code>)
<code>im_class</code>	(class, R/O): class defining the method (may be a base class)
<code>im_self</code>	(instance/None, R/O): target instance object (None if unbound)
<code>im_func</code>	(function, R/O): function object

Built-in Functions & methods

Attribute	Meaning
<code>__doc__</code>	(string/None, R/O): doc string
<code>__name__</code>	(string, R/O): function name
<code>__self__</code>	[methods only] target object
__members__	list of attr names: [<code>__doc__</code>, <code>__name__</code>, <code>__self__</code>]) Deprecated , use <code>dir()</code> instead

Codes

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

Frames

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

Tracebacks

Attribute	Meaning
tb_next	(frame/None, R/O): next level in stack trace (toward the frame where the exception occurred)
tb_frame	(frame, R/O): execution frame of the current level
tb_lineno	(int, R/O): line number where the exception occurred
tb_lasti	(int, R/O): precise instruction (index into bytecode)

Slices

Attribute	Meaning
start	(any/None, R/O): lowerbound, included
stop	(any/None, R/O): upperbound, excluded
step	(any/None, R/O): step value

Complex numbers

Attribute	Meaning
real	(float, R/O): real part
imag	(float, R/O): imaginary part

xranges

Attribute	Meaning
tolist	(Built-in method, R/O): ?

Important Modules

sys

System-specific parameters and functions. [\[Full doc\]](#)

Some sys variables

Variable	Content
<code>argv</code>	The list of command line arguments passed to a Python script. <code>sys.argv[0]</code> is the script name.
<code>builtin_module_names</code>	A list of strings giving the names of all modules written in C that are linked into this interpreter.
<code>byteorder</code>	Native byte order, either 'big'(-endian) or 'little'(-endian).
<code>check_interval</code>	How often to check for thread switches or signals (measured in number of virtual machine instructions)
<code>copyright</code>	A string containing the copyright pertaining to the Python interpreter.
<code>exec_prefix</code> <code>prefix</code>	Root directory where platform-dependent Python files are installed, e.g. 'C:\Python23', '/usr'.
<code>executable</code>	Name of executable binary of the Python interpreter (e.g. 'C:\Python23\python.exe', '/usr/bin/python')
<code>exitfunc</code>	User can set to a parameterless function. It will get called before interpreter exits.
<code>last_type</code> , <code>last_value</code> , <code>last_traceback</code>	Set only when an exception not handled and interpreter prints an error. Used by debuggers.
<code>maxint</code>	Maximum positive value for integers. Since 2.2 integers and long integers are unified, thus integers have no limit.
<code>maxunicode</code>	Largest supported code point for a Unicode character.
<code>modules</code>	Dictionary of modules that have already been loaded.
<code>path</code>	Search path for external modules. Can be modified by program. <code>sys.path[0]</code> == directory of script currently executed.
<code>platform</code>	The current platform, e.g. "sunos5", "win32"
<code>ps1</code> , <code>ps2</code>	Prompts to use in interactive mode, normally ">>>" and "..."

stdin, stdout, stderr	File objects used for I/O. One can redirect by assigning a new file object to them (or any object: with a method <code>write(string)</code> for stdout/stderr, or with a method <code>readline()</code> for stdin). <code>__stdin__</code> , <code>__stdout__</code> and <code>__stderr__</code> are the default values.
version	String containing version info about Python interpreter.
version_info	Tuple containing Python version info - (<i>major, minor, micro, level, serial</i>).
winver	Version number used to form registry keys on Windows platforms (e.g. '2.2').

Some sys functions

Function	Result
displayhook	The function used to display the output of commands issued in interactive mode - defaults to the builtin <code>repr()</code> . <code>__displayhook__</code> is the original value.
excepthook	Can be set to a user defined function, to which any uncaught exceptions are passed. <code>__excepthook__</code> is the original value.
exit(<i>n</i>)	Exits with status <i>n</i> (usually 0 means OK). Raises <code>SystemExit</code> exception (hence can be caught and ignored by program)
getrefcount(<i>object</i>)	Returns the reference count of the object. Generally 1 higher than you might expect, because of <i>object</i> arg temp reference.
setcheckinterval(<i>interval</i>)	Sets the interpreter's thread switching interval (in number of bytecode instructions, default: 10 until 2.2, 100 from 2.3).
settrace(<i>func</i>)	Sets a trace function: called before each line of code is exited.
setprofile(<i>func</i>)	Sets a profile function for performance profiling.
exc_info()	Info on exception currently being handled; this is a tuple (<code>exc_type</code> , <code>exc_value</code> , <code>exc_traceback</code>). Warning: assigning the traceback return value to a local variable in a function handling an exception will cause a circular reference.
setdefaultencoding(<i>encoding</i>)	Change default Unicode encoding - defaults to 7-bit ASCII.
getrecursionlimit()	Retrieve maximum recursion depth.
setrecursionlimit()	Set maximum recursion depth (default 1000).

os

Miscellaneous operating system interfaces. [\[Full doc\]](#)

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

Some os functions

Function	Result
<code>makedirs(path[, mode=0777])</code>	Recursive directory creation (create required intermediary dirs); <code>os.error</code> if fails.
<code>removedirs(path)</code>	Recursive directory delete (delete intermediary empty dirs); fails (<code>os.error</code>) if the directories are not empty.
<code>renames(old, new)</code>	Recursive directory or file renaming; <code>os.error</code> if fails.

posix

Posix OS interfaces. [\[Full doc\]](#)

Do not import this module directly, import `os` instead ! (see also module: [shutil](#) for file copy & remove functions)

posix Variables

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

Some posix functions

Function	Result
<code>chdir(path)</code>	Changes current directory to <i>path</i> .
<code>chmod(path, mode)</code>	Changes the mode of <i>path</i> to the numeric <i>mode</i>
<code>close(fd)</code>	Closes file descriptor <i>fd</i> opened with <code>posix.open</code> .
<code>_exit(n)</code>	Immediate exit, with no cleanups, no <code>SystemExit</code> , etc... Should use this to exit a child process.
<code>execv(p, args)</code>	"Become" executable <i>p</i> with args <i>args</i>
<code>getcwd()</code>	Returns a string representing the current working directory.
<code>getcwdu()</code>	Returns a Unicode string representing the current working directory.
<code>getpid()</code>	Returns the current process id.
<code>fork()</code>	Like C's <code>fork()</code> . Returns 0 to child, child pid to parent [Not on Windows].
<code>kill(pid, signal)</code>	Like C's <code>kill</code> [Not on Windows].
<code>listdir(path)</code>	Lists (base)names of entries in directory <i>path</i> , excluding '.' and '..'. If <i>path</i> is a Unicode string, so will be the returned strings.
<code>lseek(fd, pos, how)</code>	Sets current position in file <i>fd</i> to position <i>pos</i> , expressed as an offset relative to beginning of file (<i>how</i> =0), to current position (<i>how</i> =1), or to end of file (<i>how</i> =2).
<code>makedirs(path[, mode])</code>	Creates a directory named <i>path</i> with numeric <i>mode</i> (default 0777).
<code>open(file, flags, mode)</code>	Like C's <code>open()</code> . Returns file descriptor. Use file object functions rather than this low level ones.
<code>pipe()</code>	Creates a pipe. Returns pair of file descriptors (r, w) [Not on Windows].
<code>popen(command, mode='r', bufsize=0)</code>	Opens a pipe to or from <i>command</i> . Result is a file object to read to or write from, as indicated by <i>mode</i> being 'r' or 'w'. Use it to catch a command output ('r' mode), or to feed it ('w' mode).
<code>remove(path)</code>	See <code>unlink</code> .
<code>rename(old, new)</code>	Renames/moves the file or directory <i>old</i> to <i>new</i> . [error if target name already exists]
<code>renames(old, new)</code>	Recursive directory or file renaming function. Works like <code>rename()</code> , 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 <code>removedirs()</code> .
<code>rmdir(path)</code>	Removes the empty directory <i>path</i>
<code>read(fd, n)</code>	Reads <i>n</i> bytes from file descriptor <i>fd</i> and return as string.
<code>stat(path)</code>	Returns <code>st_mode</code> , <code>st_ino</code> , <code>st_dev</code> , <code>st_nlink</code> , <code>st_uid</code> , <code>st_gid</code> , <code>st_size</code> , <code>st_atime</code> , <code>st_mtime</code> , <code>st_ctime</code> . [<code>st_ino</code> , <code>st_uid</code> , <code>st_gid</code> are dummy on Windows]
<code>system(command)</code>	Executes string <i>command</i> in a subshell. Returns exit status of subshell (usually 0 means OK).
<code>times()</code>	Returns accumulated CPU times in sec (user, system, children's user, children's sys, elapsed real time) [3 last not on Windows].
<code>unlink(path)</code>	Unlinks ("deletes") the file (not dir!) <i>path</i> . Same as: <code>remove</code> .
<code>utime(path, (aTime, mTime))</code>	Sets the access & modified time of the file to the given tuple of values.
<code>wait()</code>	Waits for child process completion. Returns tuple of <code>pid</code> , <code>exit_status</code> [Not on Windows].
<code>waitpid(pid, options)</code>	Waits for process <i>pid</i> to complete. Returns tuple of <i>pid</i> , <code>exit_status</code> [Not on Windows].
<code>write(fd, str)</code>	Writes <i>str</i> to file <i>fd</i> . Returns nb of bytes written.

posixpath

Posix pathname operations.

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

Some *posixpath* functions

Function	Result
<code>abspath(p)</code>	Returns absolute path for path <i>p</i> , taking current working dir in account.
<code>commonprefix(list)</code>	Returns the longest path prefix (taken character-by-character) that is a prefix of all paths in list (or "" if <i>list</i> empty).
<code>dirname/basename(p)</code>	directory and name parts of the path <i>p</i> . See also <code>split</code> .
<code>exists(p)</code>	True if string <i>p</i> is an existing path (file or directory).
<code>expanduser(p)</code>	Returns string that is (a copy of) <i>p</i> with "~" expansion done.
<code>expandvars(p)</code>	Returns string that is (a copy of) <i>p</i> with environment vars expanded. [Windows: case significant; must use Unix: <code>\$var</code> notation, not <code>%var%</code>]
<code>getmtime(filepath)</code>	Returns last modification time of <i>filepath</i> (integer nb of seconds since epoch).
<code>getatime(filepath)</code>	Returns last access time of <i>filepath</i> (integer nb of seconds since epoch).
<code>getsize(filepath)</code>	Returns the size in bytes of <i>filepath</i> . <code>os.error</code> if file inexistent or inaccessible.
<code>isabs(p)</code>	True if string <i>p</i> is an absolute path.
<code>isdir(p)</code>	True if string <i>p</i> is a directory.
<code>islink(p)</code>	True if string <i>p</i> is a symbolic link.
<code>ismount(p)</code>	True if string <i>p</i> is a mount point [true for all dirs on Windows].
<code>join(p[,q[,...]])</code>	Joins one or more path components intelligently.
<code>split(p)</code>	Splits <i>p</i> into (head, tail) where <i>tail</i> is last pathname component and <i>head</i> is everything leading up to that. <=>

	<code>(dirname(p), basename(p))</code>
<code>splitdrive(p)</code>	Splits path <i>p</i> in a pair ('drive:', tail) [Windows]
<code>splittext(p)</code>	Splits into (root, ext) where last comp of <i>root</i> contains no periods and <i>ext</i> is empty or starts with a period.
<code>walk(p, visit, arg)</code>	Calls the function <i>visit</i> with arguments (<i>arg</i> , <i>dirname</i> , <i>names</i>) for each directory recursively in the directory tree rooted at <i>p</i> (including <i>p</i> itself if it's a dir). The argument <i>dirname</i> specifies the visited directory, the argument <i>names</i> lists the files in the directory. The <i>visit</i> function may modify <i>names</i> to influence the set of directories visited below <i>dirname</i> , e.g. to avoid visiting certain parts of the tree.

shutil

High-level file operations (copying, deleting). [\[Full doc\]](#)

Main *shutil* functions

Function	Result
<code>copy(src, dest)</code>	Copies the contents of file <i>src</i> to file <i>dest</i> , retaining file permissions.
<code>copytree(src, dest[, symlinks])</code>	Recursively copies an entire directory tree rooted at <i>src</i> into <i>dest</i> (which should not already exist). If <i>symlinks</i> is true, links in <i>src</i> are kept as such in <i>dest</i> .
<code>move(src, dest)</code>	Recursively moves a file or directory to a new location.
<code>rmtree(path[, ignore_errors[, onerror]])</code>	Deletes an entire directory tree, ignoring errors if <i>ignore_errors</i> is true, or calling <i>onerror</i> (func, path, sys.exc_info()) if supplied, with arguments <i>func</i> (faulty function), and <i>path</i> (concerned file).

(and also: *copyfile*, *copymode*, *copystat*, *copy2*)

time

Time access and conversions. [\[Full doc\]](#)

Variables

Variable	Meaning
<code>altzone</code>	Signed offset of local DST timezone in sec west of the 0th meridian.
<code>daylight</code>	Non zero if a DST timezone is specified.

Some functions

Function	Result
<code>time()</code>	Returns a float representing UTC time in seconds since the epoch.
<code>gmtime(secs), localtime(secs)</code>	Returns a tuple representing time : (year aaaa, month(1-12), day(1-31), hour(0-23), minute(0-59), second(0-59), weekday(0-6, 0 is monday), Julian day(1-366), daylight flag(-1,0 or 1)).
<code>asctime(timeTuple),</code>	24-character string of the following form: 'Sun Jun 20 23:21:05 1993'.
<code>strftime(format, timeTuple)</code>	Returns a formatted string representing time. See format in table below.
<code>mktime(tuple)</code>	Inverse of <code>localtime()</code> . Returns a float.
<code>strptime(string[, format])</code>	Parses a formatted string representing time, return tuple as in <code>gmtime()</code> .
<code>sleep(secs)</code>	Suspends execution for <i>secs</i> seconds. <i>secs</i> can be a float.

and also: *clock*, *ctime*.

Formatting in `strftime()`

Directive	Meaning
<code>%a</code>	Locale's abbreviated weekday name.
<code>%A</code>	Locale's full weekday name.
<code>%b</code>	Locale's abbreviated month name.
<code>%B</code>	Locale's full month name.
<code>%c</code>	Locale's appropriate date and time representation.
<code>%d</code>	Day of the month as a decimal number [01,31].
<code>%H</code>	Hour (24-hour clock) as a decimal number [00,23].
<code>%I</code>	Hour (12-hour clock) as a decimal number [01,12].
<code>%j</code>	Day of the year as a decimal number [001,366].
<code>%m</code>	Month as a decimal number [01,12].
<code>%M</code>	Minute as a decimal number [00,59].
<code>%p</code>	Locale's equivalent of either AM or PM.
<code>%S</code>	Second as a decimal number [00,61]. Yes, 61 !
<code>%U</code>	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.
<code>%w</code>	Weekday as a decimal number [0(Sunday),6].
<code>%W</code>	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.
<code>%x</code>	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 (or by no characters if no time zone exists).
%%	A literal "%" character.

string

Common string operations. [\[Full doc\]](#)

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 - see [Operations on strings](#) for details.

Some *string* variables

Variable	Meaning
digits	The string '0123456789'.
hexdigits, octdigits	Legal hexadecimal & octal digits.
letters, uppercase, lowercase, whitespace	Strings containing the appropriate characters.
ascii_letters, ascii_lowercase, ascii_uppercase	Same, taking the current <i>locale</i> in account.
index_error	Exception raised by <code>index()</code> if substring not found.

Some *string* functions

Function	Result
<code>expandtabs(s, tabSize)</code>	Returns a copy of string <i>s</i> with tabs expanded.
<code>find/rfind(s, sub[, start=0[, end=0]])</code>	Returns the lowest/highest index in <i>s</i> where the substring <i>sub</i> is found such that <i>sub</i> is wholly contained in <i>s</i> [<i>start</i> : <i>end</i>]. Return -1 if <i>sub</i> not found.
<code>ljust/rjust/center(s, width)</code>	Returns a copy of string <i>s</i> ; left/right justified/centered in a field of given width, padded with spaces. <i>s</i> is never truncated.
<code>lower/upper(s)</code>	Returns a string that is (a copy of) <i>s</i> in lowercase/uppercase.
<code>split(s[, sep=whitespace[, maxsplit=0]])</code>	Returns a list containing the words of the string <i>s</i> , using the string <i>sep</i> as a separator.
<code>join(words[, sep=' '])</code>	Concatenates a list or tuple of words with intervening separators; inverse of <code>split</code> .
<code>replace(s, old, new[, maxsplit=0])</code>	Returns a copy of string <i>s</i> with all occurrences of substring <i>old</i> replaced by <i>new</i> . Limits to <i>maxsplit</i> first substitutions if specified.
<code>strip(s[, chars=None])</code>	Returns a string that is (a copy of) <i>s</i> without leading and trailing <i>chars</i> (default: whitespace). Also: <code>lstrip</code> , <code>rstrip</code> .

re (sre)

Regular expression operations. [\[Full doc\]](#)

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

Regular expression syntax

Form	Description
.	Matches any character (including newline if DOTALL flag specified).
^	Matches start of the string (of every line in MULTILINE mode).
\$	Matches end of the string (of every line in MULTILINE mode).
*	0 or more of preceding regular expression (as many as possible).
+	1 or more of preceding regular expression (as many as possible).
?	0 or 1 occurrence of preceding regular expression.
*?, +?, ??	Same as *, + and ? but matches as few characters as possible.
{m,n}	Matches from m to n repetitions of preceding RE.
{m,n}?	Idem, attempting to match as few repetitions as possible.
[]	Defines character set: e.g. '[a-zA-Z]' to match all letters (see also <code>\w \S</code>).
[^]	Defines complemented character set: matches if char is NOT in set.
\	Escapes special chars '*?+&\$ ()' and introduces special sequences (see below). Due to Python string rules, write as <code>'\\'</code> or <code>r''</code> in the pattern string.
\\	Matches a literal <code>'\'</code> ; due to Python string rules, write as <code>'\\\\'</code> in pattern string, or better using raw string: <code>r'\\'</code> .
	Specifies alternative: 'foo bar' matches 'foo' or 'bar'.
(...)	Matches any RE inside (), and delimits a <i>group</i> .
(?:...)	Idem but doesn't delimit a <i>group</i> (<i>non capturing</i> parenthesis).
(?P<name>...)	Matches any RE inside (), and delimits a named group , (e.g. <code>r'(?Pid[a-zA-Z_]\w*)'</code> defines a group named <i>id</i>).
(?P=name)	Matches whatever text was matched by the earlier group named <i>name</i> .
(?=...)	Matches if ... matches next, but doesn't consume any of the string e.g. 'Isaac (=?Asimov)' matches 'Isaac' only if followed by 'Asimov'.

(?!...)	Matches if ... doesn't match next. Negative of (?=...).
(?<=...)	Matches if the current position in the string is preceded by a match for ... that ends at the current position. This is called a <i>positive lookbehind assertion</i> .
(?<!...)	Matches if the current position in the string is not preceded by a match for This is called a <i>negative lookbehind assertion</i> .
(?#...)	A comment; ignored.
(?letters)	<i>letters</i> is one or more of 'i','l', 'm', 's', 'x'. Sets the corresponding flags (re.I, re.L, re.M, re.S, re.X) for the entire RE.

Special sequences

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

Variables

Variable	Meaning
error	Exception when pattern string isn't a valid regexp.

Functions

Function	Result
compile(<i>pattern</i> [, <i>flags</i> =0])	Compiles a RE pattern string into a <i>regular expression object</i> . Flags (combinable by): I or IGNORECASE or (?i) case insensitive matching L or LOCALE or (?L) make \w, \W, \b, \B dependent on the current locale M or MULTILINE or (?m) matches every new line and not only start/end of the whole string S or DOTALL or (?s) '.' matches ALL chars, including newline X or VERBOSE or (?x) Ignores whitespace outside character sets
escape(<i>string</i>)	Returns (a copy of) <i>string</i> with all non-alphanumerics backslashed.
match(<i>pattern</i> , <i>string</i> [, <i>flags</i>])	If 0 or more chars at beginning of <i>string</i> matches the RE pattern string, returns a corresponding <i>MatchObject</i> instance, or <i>None</i> if no match.
search(<i>pattern</i> , <i>string</i> [, <i>flags</i>])	Scans thru <i>string</i> for a location matching <i>pattern</i> , returns a corresponding <i>MatchObject</i> instance, or <i>None</i> if no match.
split(<i>pattern</i> , <i>string</i> [, <i>maxsplit</i> =0])	Splits <i>string</i> by occurrences of <i>pattern</i> . If capturing () are used in pattern, then occurrences of patterns or subpatterns are also returned.
findall(<i>pattern</i> , <i>string</i>)	Returns a list of non-overlapping matches in <i>pattern</i> , either a list of groups or a list of tuples if the pattern has more than 1 group.
sub(<i>pattern</i> , <i>repl</i> , <i>string</i> [, <i>count</i> =0])	Returns string obtained by replacing the (<i>count</i> first) leftmost non-overlapping occurrences of <i>pattern</i> (a string or a RE object) in <i>string</i> by <i>repl</i> ; <i>repl</i> can be a string or a function called with a single <i>MatchObj</i> arg, which must return the replacement string.
subn(<i>pattern</i> , <i>repl</i> , <i>string</i> [, <i>count</i> =0])	Same as sub(), but returns a tuple (newString, numberOfSubsMade).

Regular Expression Objects

RE objects are returned by the [compile](#) function.

re object attributes

Attribute	Description
flags	Flags arg used when RE obj was compiled, or 0 if none provided.
groupindex	Dictionary of {group name: group number} in pattern.
pattern	Pattern string from which RE obj was compiled.

re object methods

Method	Result

<code>match(string[, pos][, endpos])</code>	<p>If zero or more characters at the beginning of string match this regular expression, return a corresponding <code>MatchObject</code> instance. Return <code>None</code> if the string does not match the pattern; note that this is different from a zero-length match.</p> <p>The optional second parameter <code>pos</code> gives an index in the string where the search is to start; it defaults to 0. This is not completely equivalent to slicing the string; the <code>"</code> pattern character matches at the real beginning of the string and at positions just after a newline, but not necessarily at the index where the search is to start.</p> <p>The optional parameter <code>endpos</code> limits how far the string will be searched; it will be as if the string is <code>endpos</code> characters long, so only the characters from <code>pos</code> to <code>endpos</code> will be searched for a match.</p>
<code>search(string[, pos][, endpos])</code>	<p>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.</p> <p>The optional <code>pos</code> and <code>endpos</code> parameters have the same meaning as for the <code>match()</code> method.</p>
<code>split(string[, maxsplit=0])</code>	Identical to the <code>split()</code> function, using the compiled pattern.
<code>findall(string)</code>	Identical to the <code>findall()</code> function, using the compiled pattern.
<code>sub(repl, string[, count=0])</code>	Identical to the <code>sub()</code> function, using the compiled pattern.
<code>subn(repl, string[, count=0])</code>	Identical to the <code>subn()</code> function, using the compiled pattern.

Match Objects

Match objects are returned by the `match` & `search` functions.

Match object attributes

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

Match object functions

Function	Result
<code>group([g1, g2, ...])</code>	Returns one or more groups of the match. If one arg, result is a string; if multiple args, result is a tuple with one item per arg. If <code>gi</code> is 0, return value is entire matching string; if <code>1 <= gi <= 99</code> , return string matching group <code>#gi</code> (or <code>None</code> if no such group); <code>gi</code> may also be a group <i>name</i> .
<code>groups()</code>	Returns a tuple of all groups of the match; groups not participating to the match have a value of <code>None</code> . Returns a string instead of tuple if <code>len(tuple)==1</code> .
<code>start(group), end(group)</code>	Returns indices of start & end of substring matched by group (or <code>None</code> if group exists but didn't contribute to the match).
<code>span(group)</code>	Returns the 2-tuple (<code>start(group), end(group)</code>); can be (<code>None, None</code>) if group didn't contribute to the match.

math

For intensive number crunching, see also [Numerical Python](#) and the [Python and Scientific computing](#) page. [\[Full doc\]](#)

Constants

Name	Value
<code>pi</code>	3.1415926535897931
<code>e</code>	2.7182818284590451

Functions

Name	Result
<code>acos(x)</code>	Return the arc cosine (measured in radians) of <code>x</code> .
<code>asin(x)</code>	Return the arc sine (measured in radians) of <code>x</code> .
<code>atan(x)</code>	Return the arc tangent (measured in radians) of <code>x</code> .
<code>atan2(x, y)</code>	Return the arc tangent (measured in radians) of <code>y/x</code> . Unlike <code>atan(y/x)</code> , the signs of both <code>x</code> and <code>y</code> are considered.
<code>ceil(x)</code>	Return the ceiling of <code>x</code> as a float. This is the smallest integral value <code>>= x</code> .
<code>cos(x)</code>	Return the cosine of <code>x</code> (measured in radians).
<code>cosh(x)</code>	Return the hyperbolic cosine of <code>x</code> .
<code>degrees(x)</code>	Convert angle <code>x</code> from radians to degrees.
<code>exp(x)</code>	Return <code>e</code> raised to the power of <code>x</code> .
<code>fabs(x)</code>	Return the absolute value of the float <code>x</code> .
<code>floor(x)</code>	Return the floor of <code>x</code> as a float. This is the largest integral value <code><= x</code> .
<code>fmod(x, y)</code>	Return <code>fmod(x, y)</code> , according to platform C. <code>x % y</code> may differ.
<code>frexp(x)</code>	Return the mantissa and exponent of <code>x</code> , as pair <code>(m, e)</code> . <code>m</code> is a float and <code>e</code> is an int, such that <code>x = m * 2.**e</code> . If <code>x</code> is 0, <code>m</code> and <code>e</code> are both 0. Else <code>0.5 <= abs(m) < 1.0</code> .
<code>hypot(x, y)</code>	Return the Euclidean distance <code>sqrt(x*x + y*y)</code> .

ldexp(<i>x</i> , <i>i</i>)	<i>x</i> * (2** <i>i</i>)
log(<i>x</i> [, <i>base</i>])	Return the logarithm of <i>x</i> to the given <i>base</i> . If the base is not specified, returns the natural logarithm (base e) of <i>x</i> .
log10(<i>x</i>)	Return the base 10 logarithm of <i>x</i> .
modf(<i>x</i>)	Return the fractional and integer parts of <i>x</i> . Both results carry the sign of <i>x</i> . The integer part is returned as a float.
pow(<i>x</i> , <i>y</i>)	Return <i>x</i> ** <i>y</i> (<i>x</i> to the power of <i>y</i>). Note that for <i>y</i> =2, it is more efficient to use <i>x</i> * <i>x</i> .
radians(<i>x</i>)	Convert angle <i>x</i> from degrees to radians.
sin(<i>x</i>)	Return the sine (measured in radians) of <i>x</i> .
sinh(<i>x</i>)	Return the hyperbolic sine of <i>x</i> .
sqrt(<i>x</i>)	Return the square root of <i>x</i> .
tan(<i>x</i>)	Return the tangent (measured in radians) of <i>x</i> .
tanh(<i>x</i>)	Return the hyperbolic tangent of <i>x</i> .

getopt
Parser for command line options. [\[Full doc\]](#)

This was the standard parser until Python 2.3, now superseded by [optparse](#).
[see also: Richard Gruet's simple parser [getargs.py](#) (shameless self promotion)]

Functions:

```
getopt(list, optstr)    -- Similar to C. <optstr> is option letters to look for.
                        Put ':' after letter if option takes arg. E.g.
# invocation was "python test.py -c hi -a arg1 arg2"
  opts, args = getopt.getopt(sys.argv[1:], 'ab:c:')
# opts would be
  [('-c', 'hi'), ('-a', '')]
# args would be
  ['arg1', 'arg2']
```

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.
[Python NT distribution, may be slightly different in other distributions]

Standard library modules

Operation	Result
aifc	Stuff to parse AIFF-C and AIFF files.
anydbm	Generic interface to all dbm clones. (dbhash, gdbm, dbm, dumbdbm).
asynchat	A class supporting chat-style (command/response) protocols.
asyncore	Basic infrastructure for asynchronous socket service clients and servers.
atexit	Register functions to be called at exit of Python interpreter.
<code>audiodev</code>	Classes for manipulating audio devices (currently only for Sun and SGI).
base64	Conversions to/from base64 transport encoding as per RFC-1521.
BaseHTTPServer	HTTP server base class
Bastion	"Bastionification" utility (control access to instance vars).
<code>bdb</code>	A generic Python debugger base class.
bsddb	(Optional) improved BSD database interface [package].
binhex	Macintosh binhex compression/decompression.
bisect	Bisection algorithms.
bz2	BZ2 compression.
calendar	Calendar printing functions.
cgi	Wraps the WWW Forms Common Gateway Interface (CGI).
CGIHTTPServer	CGI-savvy HTTP Server.
cmd	A generic class to build line-oriented command interpreters.
cmp	Efficiently compare files, boolean outcome only.
cmpcache	Same, but caches 'stat' results for speed.
code	Utilities needed to emulate Python's interactive interpreter.
codecs	Lookup existing Unicode encodings and register new ones.
codeop	Utilities to compile possibly incomplete Python source code.
colorsys	Conversion functions between RGB and other color systems.
commands	Execute shell commands via <code>os.popen</code> [Unix only].
compileall	Force "compilation" of all <code>.py</code> files in a directory.
ConfigParser	Configuration file parser (much like windows <code>.ini</code> files).
Cookie	HTTP state (cookies) management.
copy	Generic shallow and deep copying operations.

copy_reg	Helper to provide extensibility for modules pickle/cPickle.
csv	Tools to read comma-separated files (of variations thereof).
datetime	Improved date/time types (date , time , datetime , timedelta).
dbhash	(g)dbm-compatible interface to bsddb.hashopen .
difflib	Tool for comparing sequences, and computing the changes required to convert one into another.
dircache	Sorted list of files in a dir, using a cache.
dircmp	Defines a class to build directory diff tools on.
dis	Bytecode disassembler.
distutils	Package installation system.
distutils.command.register	Register module in the Python package index (PyPI). This command plugin adds the register command to distutil scripts.
distutils.debug	
distutils.emxcompiler	
distutils.log	
doctest	Unit testing framework based on running examples embedded in docstrings.
DocXMLRPCServer	Creation of self-documenting XML-RPC servers, using pydoc to create HTML API doc on the fly.
dospath	Common operations on DOS pathnames.
dumbdbm	A dumb and slow but simple dbm clone.
dump	Print python code that reconstructs a variable.
dummy_thread	
dummy_threading	Helpers to make it easier to write code that use threads where supported, but still runs on Python versions without thread support. The dummy modules simply run the threads sequentially.
encodings	New codecs: idna (IDNA strings), koi8_u (Ukrainian), palmos (PalmOS 3.5), punycode (Punycode IDNA codec), string_escape (Python string escape codec: replaces non-printable chars w/ Python-style string escapes).
exceptions	Class based built-in exception hierarchy.
filecmp	File and directory comparison.
fileinput	Helper class to quickly write a loop over all standard input files.
find	Find files directory hierarchy matching a pattern.
fnmatch	Filename matching with shell patterns.
formatter	Generic output formatting.
fpformat	General floating point formatting functions.
ftplib	An FTP client class. Based on RFC 959.
gc	Perform garbage collection, obtain GC debug stats, and tune GC parameters.
getopt	Standard command line processing. See also optparse .
getpass	Utilities to get a password and/or the current user name.
gettext	Internationalization and localization support.
glob	Filename "globbing" utility.
gopherlib	Gopher protocol client interface.
grep	'grep' utilities.
gzip	Read & write gzipped files.
heapq	Heap queue (priority queue) helpers.
hmac	HMAC (Keyed-Hashing for Message Authentication).
hotshot.stones	Helper to run the pystone benchmark under the Hotshot profiler.
htmlentitydefs	HTML character entity references.
htmllib	HTML2 parsing utilities
HTMLParser	Simple HTML and XHTML parser.
httplib	HTTP1 client class.
idlelib	(package) Support library for the IDLE development environment.
ihooks	Hooks into the "import" mechanism.
imaplib	IMAP4 client. Based on RFC 2060.
imghdr	Recognizing image files based on their first few bytes.
imputil	Provides a way of writing customized import hooks.
inspect	Get information about live Python objects.
itertools	Tools to work with iterators and lazy sequences.
keyword	List of Python keywords.
knee	A Python re-implementation of hierarchical module import.
linecache	Cache lines from files.
linuxaudiodev	Linux /dev/audio support. Replaced by ossaudiodev (Linux).
locale	Support for number formatting using the current locale settings.
logging	(package) Tools for structured logging in log4j style.
macpath	Pathname (or related) operations for the Macintosh [Mac].
macurl2path	Mac specific module for conversion between pathnames and URLs [Mac].
mailbox	Classes to handle Unix style, MMDF style, and MH style mailboxes.

mailcap	Mailcap file handling (RFC 1524).
markupbase	Shared support for scanning document type declarations in HTML and XHTML.
mhlib	MH (mailbox) interface.
mimetools	Various tools used by MIME-reading or MIME-writing programs.
mimetypes	Guess the MIME type of a file.
MimeWriter	Generic MIME writer. Deprecated since release 2.3. Use the <code>email</code> package instead.
mimify	Mimification and unmimification of mail messages.
mmap	Interface to memory-mapped files - they behave like mutable strings.
modulefinder	Tools to find what modules a given Python program uses, without actually running the program.
multifile	A readline()-style interface to the parts of a multipart message.
mutex	Mutual exclusion -- for use with module <code>sched</code> . See also std module <code>threading</code> , and glock .
netrc	Parses and encapsulates the netrc file format.
nntplib	An NNTP client class. Based on RFC 977.
ntpath	Common operations on Windows pathnames.
nturl2path	Convert a NT pathname to a file URL and vice versa.
olddifflib	Old version of <code>difflib</code> (helpers for computing deltas between objects)?
optparse	Improved command-line option parsing library (see also <code>getopt</code>).
os	OS routines for Mac, DOS, NT, or Posix depending on what system we're on.
os2emxpath	os.path support for OS/2 EMX.
packmail	Create a self-unpacking shell archive.
pdb	A Python debugger.
pickle	Pickling (save and restore) of Python objects (a faster C implementation exists in built-in module: <code>cPickle</code>).
pickletools	Tools to analyze and disassemble pickles.
pipes	Conversion pipeline templates.
pkgutil	Tools to extend the module search path for a given package.
platform	Get info about the underlying platform.
poly	Polynomials.
popen2	Spawn a command with pipes to its stdin, stdout, and optionally stderr.
poplib	A POP3 client class.
posixfile	Extended file operations available in POSIX.
posixpath	Common operations on POSIX pathnames.
pprint	Support to pretty-print lists, tuples, & dictionaries recursively.
re	Support for regular expressions (RE) - see <code>re</code> .
profile	Class for profiling python code.
pstats	Class for printing reports on profiled python code.
pty	Pseudo terminal utilities.
py_compile	Routine to "compile" a .py file to a .pyc file.
pyclbr	Parse a Python file and retrieve classes and methods.
pydoc	Generate Python documentation in HTML or text for interactive use.
pyexpat	Interface to the Expat XML parser.
PyUnit	Unit test framework inspired by JUnit. See <code>unittest</code>.
Queue	A multi-producer, multi-consumer queue.
quopri	Conversions to/from quoted-printable transport encoding as per RFC 1521.
rand	Don't use unless you want compatibility with C's <code>rand()</code> .
random	Random variable generators (obsolete, use <code>whrandom</code>).
re	Regular Expressions.
readline	GNU readline interface [Unix] .
reconvert	Convert old ("regex") regular expressions to new syntax ("re").
regex_syntax	Flags for <code>regex.set_syntax()</code> .
regexp	Backward compatibility for module "regexp" using "regex".
regsub	Regexp-based split and replace using the obsolete <code>regex</code> module.
repr	Redo <code>repr()</code> but with limits on most sizes.
rexec	Restricted execution facilities ("safe" <code>exec</code>, <code>eval</code>, etc).
rfc822	Parse RFC-822 mail headers.
rlcompleter	Word completion for GNU readline 2.0.
robotparser	Parse robot.txt files, useful for web spiders.
sched	A generally useful event scheduler class.
sets	A Set datatype implementation based on dictionaries (see <code>Sets</code>).
sgmlib	A parser for SGML, using the derived class as a static DTD.
shelve	Manage shelves of pickled objects.
shlex	Lexical analyzer class for simple shell-like syntaxes.
shutil	Utility functions for copying files and directory trees.
SimpleHTTPServer	Simple HTTP Server.

SimpleXMLRPCServer	Simple XML-RPC Server
site	Append module search paths for third-party packages to <code>sys.path</code> .
smtpd	An RFC 2821 smtp proxy.
smtpplib	SMTP/ESMTP client class.
sndhdr	Several routines that help recognizing sound.
socket	Socket operations and some related functions. Now supports timeouts thru function <code>settimeout(t)</code>. Also supports SSL on Windows.
SocketServer	Generic socket server classes.
sre	Support for regular expressions (RE). See re .
stat	Constants/functions for interpreting results of <code>os</code> .
statcache	Maintain a cache of <code>stat()</code> information on files.
statvfs	Constants for interpreting <code>statvfs</code> struct as returned by <code>os.statvfs()</code> and <code>os.fstatvfs()</code> (if they exist).
string	A collection of string operations (see Strings).
stringprep	Normalization and manipulation of Unicode strings.
StringIO	File-like objects that read/write a string buffer (a faster C implementation exists in built-in module: <code>cStringIO</code>).
sunau	Stuff to parse Sun and NeXT audio files.
sunaudio	Interpret sun audio headers.
symbol	Non-terminal symbols of Python grammar (from "graminit.h").
symtable	Interface to the compiler's internal symbol tables.
tabnanny	Check Python source for ambiguous indentation.
tarfile	Tools to read and create TAR archives.
telnetlib	TELNET client class. Based on RFC 854.
tempfile	Temporary files and filenames.
textwrap	Tools to wrap paragraphs of text.
threading	Proposed new threading module, emulating a subset of Java's threading model.
threading_api	(doc of the threading module).
timeit	Benchmark tool.
toaiff	Convert "arbitrary" sound files to AIFF (Apple and SGI's audio format).
token	Token constants (from "token.h").
tokenize	Tokenizer for Python source.
traceback	Extract, format and print information about Python stack traces.
trace	Tools to trace execution of a function or program.
tty	Terminal utilities [Unix].
turtle	LogoMation-like turtle graphics.
types	Define names for all type symbols in the std interpreter.
tzparse	Parse a timezone specification.
unicodedata	Interface to unicode properties.
unittest	Python unit testing framework, based on Erich Gamma's JUnit and Kent Beck's.
urllib	Open an arbitrary URL.
urllib2	An extensible library for opening URLs using a variety of protocols.
urlparse	Parse (absolute and relative) URLs.
user	Hook to allow user-specified customization code to run.
UserDict	A wrapper to allow subclassing of built-in dict class (useless with <i>new-style</i> classes. Since Python 2.2, dict is subclassable).
UserList	A wrapper to allow subclassing of built-in list class (useless with <i>new-style</i> classes. Since Python 2.2, list is subclassable).
UserString	A wrapper to allow subclassing of built-in string class (useless with <i>new-style</i> classes. Since Python 2.2, str is subclassable).
util	some useful functions that don't fit elsewhere !!
uu	Implementation of the UUencode 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.
webbrowser	Platform independent URL launcher.
whatsound	Several routines that help recognizing sound files.
whichdb	Guess which db package to use to open a db file.
whrandom	Wichmann-Hill random number generator.
xdrlib	Implements (a subset of) Sun XDR (eXternal Data Representation).
xmllib	A parser for XML, using the derived class as static DTD.
xml.dom	Classes for processing XML using the DOM (Document Object Model). 2.3: New modules <i>expatbuilder</i>, <i>minicompat</i>, <i>NodeFilter</i>, <i>xmlbuilder</i>.
xml.sax	Classes for processing XML using the SAX API.
xmlrpclib	An XML-RPC client interface for Python.

xreadlines	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.
zipfile	Read & write PK zipped files.
zipimport	ZIP archive importer.
zmod	Demonstration of abstruse mathematical concepts.

Workspace exploration and idiom hints

```

dir(<module>)          list functions, variables in <module>
dir()                 get object keys, defaults to local name space
if __name__ == '__main__': main()  invoke main if running as script
map(None, lst1, lst2, ...)  merge lists
b = a[:]              create copy of seq structure
_                     (underscore) in interactive mode, is 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
LFD      insert newline, then indent
DEL      reduce indentation, or delete single character

```

Primarily for reindenting existing code:

```

C-c :    guess py-indent-offset from file content; change locally
C-u C-c : ditto, but change globally
C-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 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-a     move to start of def
C-u M-C-a move to start of class
M-C-e     move to end of def
C-u M-C-e move to end of class

```

EXECUTING PYTHON CODE

```

C-c C-c  sends the entire buffer to the Python interpreter
C-c |    sends the current region
C-c !    starts a Python interpreter window; this will be used by
          subsequent C-c C-c or C-c | commands

```

VARIABLES

```

py-indent-offset      indentation increment
py-block-comment-prefix  comment string used by py-comment-region
py-python-command     shell command to invoke Python interpreter
py-scroll-process-buffer  t means always scroll Python process buffer
py-temp-directory      directory used for temp files (if needed)
py-beep-if-tab-change   ring the bell if tab-width is changed

```