Python 1.52 Quick Reference

1999/06/04, upgraded by Richard Gruet, rgruet@ina.fr from V1.3 ref: 1995/10/30, by Chris Hoffmann, choffman@vicorp.com

Based on:
Python Bestiary, Author: Ken Manheimer, ken.manheimer@nist.gov
Python manuals, Author: Guido van Rossum, guido@CNRI.Reston.Va.US, guido@python.org
python-mode.el, Author: Tim Peters, tim_one@email.msn.com
and the readers of comp.lang.python

Python site: http://www.python.org

Contents

- Invocation Options
- Environment Variables
- Terms Used In This Document
- Lexical Entities
- Basic Types And Their Operations
- Advanced Types
- Statements
- 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, string, re, math, getopt
- List of modules In base distribution
- Workspace Exploration And Idiom Hints
- Python Mode for Emacs
- The Python Debugger

Invocation Options

python [-diOStuvxX?] [-c *command* | *script* | -] [*args*]

```
     -d   debug output from parser (also PYTHONDEBUG=x)
     -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 (set __debug__ = 0 =>s suppresses asserts)
     -S   don't imply 'import site' on initialization
     -t   issue warnings about inconsistent tab usage (-tt: issue errors)
     -u   unbuffered binary stdout and stderr (also PYTHONUNBUFFERED=x).
     -v   verbose (trace import statements) (also PYTHONVERBOSE=x)
     -x   skip first line of source, allowing use of non-unix
          forms of #!cmd
     -X   disable class based built-in exceptions (for backward
          compatibility management of exceptions)
     -?   Help!

         -c command
```

```
        Specify the command  to  execute  (see  next  section).
        This  terminates the option list (following options are
        passed as arguments to the command).
    script is the name of a python file (.py) to execute
    -  read from stdin.

anything afterward is passed as options to python script or
   command, not interpreted as an option to interpreter itself.

args are passed to script or command (in sys.argv[1:])
```

If no script or command, Python enters interactive mode.

ENVIRONMENT VARIABLES

PYTHONHOME
: Alternate *prefix* directory (or *prefix*;*exec_prefix*). The default module search path uses *prefix*/lib

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\v.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.

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

PYTHONSUPPRESS
: If non-empty, same as -s option

PYTHONUNBUFFERED
: If non-empty, same as -u option

PYTHONVERBOSE
: If non-empty, same as -v option

PYTHONCASEOK --to be verified--
: If non-empty, ignore case in file/module names (imports)

Terms Used In This Document

sequence
: a string, list or tuple

suite
: a series of statements, possibly separated by newlines. Must all be at same indentation level, except for suites inside compound statements

<x>
    in a syntax diagram means a token referred to as "x"
[xxx]
    in a syntax diagram means "xxx" is optional
x ==> y
    means the value of <x> is <y>
x <=> y
    means "x is equivalent to y"

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
def       finally   in        print
```

- (list of keywords in std module: keyword.py)
- 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)

Operators

```
 +    -    *     **    /     %
<<    >>    &     |     ^     ~
<     >     <=    >=    ==    !=    <>
```

Strings

"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 """

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

- Use **** at end of line to continue a string on next line.
- adjacent strings are concatened, e.g. 'Monty' ' Python' is the same as 'Monty Python'.

String Literal Escapes

```
\newline  Ignored (escape newline)
\\ Backslash (\)        \e Escape (ESC)          \v Vertical Tab (VT)
\' Single quote (')     \f Formfeed (FF)         \OOO  (zero) char with
\" Double quote (")     \n Linefeed (LF)               octal value OO
\a Bell (BEL)           \r Carriage Return (CR)  \xXX  char with
\b Backspace (BS)       \t Horizontal Tab (TAB)        hex value XX
\AnyOtherChar is left as-is
```

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

Numbers

Decimal integer: 1234, 12345678905463789**40L** (or **l)**
Octal integer: **0**177, **0**1777777777777777777L (begin with a **0**)
Hex integer: **0x**FF, **0X**FFFFffffFFFFFFFFFFL (begin with **0x** or **0X**)
Long integer (unlimited precision): 1234567890123456**L** (ends with **L** or **l**)
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 real and imaginary parts which are both floats)

Sequences

- String of length 0, 1, 2 (see above)
  '', '1', "12"
- 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***]**.Start defaults to '0'; End defaults to 'sequence-length'.

```
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.
```

Dictionaries (Mappings)

Dictionary of length 0, 1, 2, etc:
{} {1 : 'first'} {1 : 'first', 'next': 'second'}

Basic Types and Their Operations

Comparisons (defined between *any* types)

```
<       strictly less than
<=      less than or equal
>       strictly greater than
>=      greater than or equal
==      equal
!=      not equal  ( "<>" is also allowed)
is      object identity (are objects identical, not values)
is not  negated object identity
```

X < Y < Z < W has expected meaning, unlike C

Boolean values and operators

```
False values:     None, numeric zeros, empty sequences and mappings
True values:      all other values

not X: if X is false then 1, else 0
X or Y: if X is false then Y, else X
X and Y: if X is false then X, else Y

('or', 'and' evaluate second arg only if necessary to determine outcome)
```

None

None is used as default return value on functions.
Input that evaluates to None does not print when running Python interactively.

Numeric types

Floats, integers and long integers.

Floats are implemented with C doubles.
Integers are implemented with C longs.
Long integers have unlimited size (only limit is system resources)

Operators on all numeric types

```
abs(x)  absolute value of x
int(x)  x converted to integer
long(x) x converted to long integer
float(x) x converted to floating point
-x      x negated
+x      x unchanged
x + y   sum of x and y
x - y   difference of x and y
x * y   product of x and y
```

```
x / y   quotient of x and y
x % y   remainder of x / y
divmod(x, y) the tuple (x/y, x%y)
x ** y  x to the power y [same as: pow(x,y)]
```

Bit operators on integers and long integers

```
~x      the bits of x inverted
x ^ 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 << n  x shifted left by n bits
x >> n  x shifted right by n bits
```

Complex Numbers

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)

Operation	Result	Notes
x **in** *s*	1 if an item of *s* is equal to *x*, else 0	
x **not in** *s*	0 if an item of *s* is equal to *x*, else 1	
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*]	slice of *s* from *i* (included) to *j* (excluded)	(1), (2)
len(*s*)	length of *s*	
min(*s*)	smallest item of *s*	
max(*s*)	largest item of *(s*)	

Notes :

(1) if *i* or *j* is negative, the index is relative to the end of the string, ie len(*s*)+ *i* or 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* <= *k* < *j*. If *i* or *j* is greater thanlen(*s*), use len(*s*). If *i* is omitted, use len(*s*). If *i* is greater than or

equal to *j*, the slice is empty.

Operations on mutable (=modifiable) sequences (lists)

Operation	Result	Notes
s[*i*] =*x*	item *i* of *s* is replaced by *x*	
s[*i*:*j*] = *t*	slice of *s* from *i* to *j* is replaced by *t*	
del *s*[*i*:*j*]	same as *s*[*i*:*j*] = []	
s.**append**(*x*)	same as *s*[len(*s*) : len(*s*)] = [*x*]	
s.**extend**(*x*)	same as *s*[len(*s*):len(*s*)]= *x*	(5)
s.**count**(*x*)	return number of *i*'s for which *s*[*i*] == *x*	
s.**index**(*x*)	return smallest *i* such that *s*[*i*] == *x*	(1)
s.**insert**(*i*, *x*)	same as *s*[*i*:*i*] = [*x*] if *i* >= 0	
s.**remove**(*x*)	same as del *s*[*s*.index(*x*)]	(1)
s.**pop**([*i*])	same as x = s[i]; del s[i]; return x	(4)
s.**reverse**()	reverse the items of *s* in place	(3)
s.**sort**([*cmpFct*])	sort the items of *s* in place	(2), (3)

Notes :

(1) raise 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 fct 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) [*New 1.5.2*] The pop() method is experimental and not supported by other mutable sequence types than lists. The optional argument i defaults to -1, so that by default the last item is removed and returned.

(5) [*New 1.5.2*] Experimental ! Raises an exception when x is not a list object.

Operations on mappings (dictionaries)

Operation	Result	Notes
len(*d*)	the number of items in *d*	
*d***[***k***]**	the item of *d* with key *k*	(1)
d[*k*] **=** x	set *d*[*k*] to *x*	
del *d*[*k*]	remove *d*[*k*] from *d*	(1)
d.**clear**()	remove all items from *d*	
d.**copy**()	a shallow copy of *d*	
d.**has_key**(*k*)	1 if *d* has key *k*, else 0	
d.**items**()	a copy of *d*'s list of (key, item) pairs	(2)
d.**keys**()	a copy of *d*'s list of keys	(2)
d1.**update**(*d2*)	for k, v in d2.items(): d1[k] = v	(3)
d.**values**()	a copy of *d*'s list of values	(2)
d.**get**(*k*,*defaultval*)	the item of *d* with key *k*	(4)

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.

Format operator for strings (%)

Uses C printf format codes :

Supports: %, c, s, i, d, u, o, x, X, e, E, f, g, G.

Width and precision may be a * to specify that an integer argument specifies the actual width or precision.

The flag characters -, +, blank, # and 0 are understood.

%s will convert any type argument to string (uses *str()* function)

```
a = '%s has %03d quote types.' % ('Python', 2)
a ==> 'Python has 002 quote types.'
```

Right-hand-side can 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.)

File Objects

Created with built-in function *open()*; may be created by other modules's functions as well.

Operators on file objects

```
f.close()                       close file f.
f.fileno()                      get fileno (fd) for f.
f.flush()                       flush file's internal buffer.
f.isatty()                      1 if file is connected to a tty-like dev, else 0
f.read([size])                  read at most size bytes from file and return
                                as a string object. If size omitted, read to EOF.

f.readline()                    read one entire line from file
f.readlines()                   read until EOF with readline() and return list
                                of lines read.

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

f.tell()                        return file's current position
f.write(str)                    Write string to file.
f.writelines(list)              Write list of strings to file.
```

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

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

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

Statements

```
pass              -- Null statement
=                 -- Assignment operator. Can unpack tuples, lists, strings
                     first, second = a[0:2]; [f, s] = range(2); c1,c2,c3='abc'
                     Tip: x,y = y,x swaps x and y.
del <name>[,<name>]* -- Unbind name(s) from object. Object will be indirectly
                      (and automatically) deleted only if no longer referenced.
print [<c1> [, <c2> ]* [,]
                  -- Writes to sys.stdout.
                     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 None.
exec <x> [in <globals> [,<locals>]]
                  -- Executes <x> in namespaces provided. Defaults
                     to current namespaces. <x> can be a string, file
                     object or a function object.
f(<value>,... [<id>=<value>])
                  -- Call function 'f' with parameters. Parameters can
                     be passed by name or be omitted if function
                     defines default values. E.g. for 'f' is defined as
                     "def f(p1=1, p2=2)"
                     "f()"       <=>  "f(1, 2)"
                     "f(10)"     <=>  "f(10, 2)"
                     "f(p2=99)"  <=>  "f(1, 99)"
```

Control Flow

```
if <condition>: <suite>[elif <condition>: <suite>]*
[else: <suite>]
               -- usual if/else_if/else statement

while <condition>: <suite>
[else: <suite>]
               -- usual while statement. "else" suite is executed
                  after loop exits, unless the loop is exited with
                  "break"

for <target> in <condition-list>: <suite>
[else: <suite>]
               -- iterates over sequence "<condition-list>",
                  assigning each element to "<target>". Use built-in range                  function to iterate a number of times.
                  "else" suite executed at end unless loop exited
                  with "break"

break          -- immediately exit "for" or "while" loop

continue       -- immediately do next iteration of "for" or "while" loop

return [<result>]
               -- return from function (or method) and return
                  "<result>". If no result given, then returns None.
```

Exception Statements

```
assert expr[, message]
                  -- expr is evaluated. if false, raise exception AssertionError
                     with message. Inhibited if __debug__ is 0.
```

```
try: <suite1>
[except [<exception> [, <value>]: <suite2>]+
[else: <suite3>]
                -- statements in <suite1> are executed. If an
                   exception occurs, look in "except" clauses for
                   matching <exception>. If matches or bare "except"
                   execute suite of that clause. If no exception happens
                   suite in "else" clause is executed after <suite1>.
                   If <exception> has a value, it is put in <value>.
                   <exception> can also be tuple of exceptions, e.g.
                   "except (KeyError, NameError), val: print val"

try: <suite1>
finally: <suite2>
                -- statements in <suite1> are executed. If no
                   exception, execute <suite2> (even if <suite1> is
                   exited with a "return", "break" or "continue"
                   statement). If exception did occur, executes
                  <suite2> and them immediately reraises exception.

raise <exception> [,<value> [,<traceback>]]
                -- raises <exception> with optional value
                   <value>. Parameter <traceback>
                   specifies a traceback 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 either a *string* (object) or a *class instance*.
Can create a new one simply by creating a new string:

```
my_exception = 'You did something wrong'
try:
    if bad:
        raise my_exception, bad
except my_exception, value:
    print 'Oops', value
```

Exception classes must be derived from the predefined class: Exception, eg:

```
class text_exception(Exception):
try:
    if bad:
        raise text_exception()
        # This is a shorthand for the form
        # "raise <class>, <instance>"
 except Exception:
     print 'Oops'
     # This will be printed because
     # text_exception is a subclass of Exception
```

When an error message is printed for an unhandled exception which is a class, 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

[1.51: On Mac & Windows, the case of module file names must now match the case as used in the *import* statement]

```
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/dirs can be nested. You address a
                 module's symbol via '[package.[package...]module.symbol's.

import <module_id1> [, <module_id2>]*
                -- imports modules. Members of module must be
                   referred to by qualifying with [package.]module name:
                   "import sys; print sys.argv:"
                   "import package1.subpackage.module; package1.subpackage.module.foo()"

from <module_id> import <id1> [, <id2>]*
                -- imports names from module <module_id>. Names
                   are not qualified:
                   "from sys import argv; print argv"
                   "from package1 import module; module.foo()"
                   "from package1.module import foo; foo()"

from <module_id> import *               -- imports all names in module <module_id>, except
                   those starting with "_"; *to be used sparsely* :
                   "from sys import *; print argv"
                   "from package.module import *; print x'

global <id1> [,<id2>]*
                -- ids are from global scope (usually meaning from module)
                   rather than local (usually meaning only in function).
                -- E.g. in fct without "global" statements, assuming
                   "a" is name that hasn't been used in fct or module
                   so far:
                   -Try to read from "a" -> NameError
                   -Try to write to "a" -> creates "a" local to fcn
                   -If "a" not defined in fct, but is in module, then
                       -Try to read from "a", gets value from module
                       -Try to write to "a", creates "a" local to fct

                   But note "a[0]=3" starts with search for "a",
                   will use to global "a" if no local "a".
```

Function Definition

```
def <func_id> ([<param_list>]): <suite>
                -- creates a function object and assigns it name
                   <func_id>.

<param_list> ==> [<id> [, <id>]*] [<id> =
                                          <v> [,
                                          <id> =
                                          <v>]*]
                                          [*<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]
```

```
E.g.
        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.
                   (allows vararg functions).
                   If list has "**<id>" then <id> is assigned
                   a dictionary of all arguments passed as keywords.
```

Class Definition

```
class <class_id> [(<super_class1> [,<super_class2>]*)]: <suite>
        -- Creates a class object and assigns it name <class_id>
           <suite> may contain local "defs" of class methods and
           assignments to class attributes.
E.g.
       class my_class (class1, class_list[3]): ...
                  Creates a class object inheriting from both "class1" and whatever
                  class object "class_list[3]" evaluates to. Assigns new
                  class object to name "my_class".

        -First arg to class methods is always 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!)
        -In current implementation, can't subclass off built-in
         classes. But can "wrap" them, see __getattr__() below.


E.g.
        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.
```

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.
Then c.__doc__ <=> "A description of 'c'".
Then c.__init__.__doc__ <=> "A description of the constructor".
```

Others

```
lambda [<param_list>]: <returnedExpr>
                -- Create 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.
```

Built-In Functions

```
__import__(name[, globals[, locals[, fromlist]]])
                Import module within the given context (see lib ref for more details)

abs(x)          Return the absolute value of a number

apply(f, args[, keywords])
                Call func/method <f> with args <args> and optional keywords

callable(x)     Returns 1 if x callable, else 0.

chr(i)          Return one-character string whose ASCII code is
                integer i

cmp(x,y)        Return neg, zero, pos if x <, ==, > to y

coerce(x,y)     Return a tuple of the two numeric arguments converted to
                a common type.

compile(string, filename, kind)
                Compile <string> into a code object.
                <filename> is used in error message, can be any string. It is
                usually the file from which the code was read, or eg. '<string>'
                if not read from file.
                <kind> can be 'eval' if <string> is a single stmt, or 'single'
                which prints the output of expression statements that
                evaluate to something else than "None" or be 'exec'.

complex(real[, image])
                Build a complex object (can also be done using J or j suffix,
                e.g. 1+3J)

delattr(obj, name)
                deletes attribute named <name> of object <obj> <=> del obj.name

dir([object])   If no args, return the list of names in current local
                symbol table. With a module, class or class instance
                object as arg, return list of names in its attr dict.

divmod(a,b)     Returns tuple of (a/b, a%b)
```

eval(s, globals, locals)
Eval string <s> in (optional) <globals>, <locals>.
<s> must have no NULL's or newlines. <s> can also be a
code object.

E.g.: x = 1; incr_x = eval('x + 1')

execfile(file[, globals[, locals]])
Execute a file without creating a new module, unlike import.

filter(function, sequence)
Construct a list from those elements of <sequence> for which
<function> returns true. <function> takes one parameter.

float(x) Convert a number or a string to floating point.

getattr(object, name[, default]) [<default> arg added in 1.5.2] Get attr called <name> from <object>, e.g. getattr(x, 'f') <=> x.f).
If not found, raise AttributeError or return <default> if specified.

globals() Returns a dictionary containing current global variables.

hasattr(object, name)
Returns true if <object> has attr called <name>.

hash(object) Return the hash value of the object (if it has one)

hex(x) Convert a number to a hexadecimal string.

id(object) Return a unique 'identity' integer for an object.

input([prompt]) Prints prompt, if given. Reads input and evaluates it.

int(x) Convert a number or a string to a plain integer.

intern(aString)
Enter <String> in the table of "interned strings" and
return the string. Interned strings are 'immortals'.

isinstance(obj, class)
return true if <obj> is an instance of <class>

issubclass(class1, class2)
return true if <class1> is derived from <class2>

len(s) Return the length (the number of items) of an object
(sequence or dictionary).

list(sequence)
Convert <sequence> into a list. If already a list,
return a copy of it.

locals() Return a dictionary containing current local variables.

long(x) Convert a number or a string to a long integer.

map(function, list, ...)
Apply <function> to every item of <list> and return a list
of the results. If additional arguments are passed,
<function> must take that many arguments and it is given
to <function> on each call.

max(s) Return the largest item of a non-empty sequence.

min(s) Return the smallest item of a non-empty sequence.

oct(x) Convert a number to an octal string.

open(filename [, mode='r', [bufsize=<implementation dependent>]])
Return a new file object. First two args are same as
those for C's "stdio open" function. <bufsize> is 0
for unbuffered, 1 for line-buffered, negative for
sys-default, all else, of (about) given size.

ord(c) Return integer ASCII value of <c> (str of len 1).

pow(x, y [, z]) Return x to power y [modulo z]. See also ** operator.

range(start [,end [, step]])
  return list of ints from >= start and < end.
  With 1 arg, list from 0..<arg>-1
  With 2 args, list from <start>..<end>-1
  With 3 args, list from <start> up to <end> by <step>

raw_input([prompt])
  Print prompt if given, then read string from std
  input (no trailing \n).

reduce(f, list [, init])
  Apply the binary function <f> to the items of
  <list> so as to reduce the list to a single value.
  If <init> given, it is "prepended" to <list>.

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

repr(object) Return a string containing a printable representation
  of an object. Equivalent to `object` (using
  backquotes).

round(x,n=0) Return the floating point value x rounded to n digits
    after the decimal point.

setattr(object, name, value)
  This is the counterpart of getattr().
  setattr(o, 'foobar', 3) <=> o.foobar = 3
  Create attribute if it doesn't exist!

slice([start,] stop[, step])
  Return a *slice object* representing a range, with R/O
  attributes: start, stop, step.

str(object) Return a string containing a nicely printable
  representation of an object.

tuple(sequence) Creates a tuple with same elements as <sequence>. If
  already a tuple, return itself (not a copy).

type(obj) Return a type object [see module *types*]representing the
  type of <obj>. E.g., import types
  if type(x) == types.StringType: print 'It is a string'

vars([object]) Without arguments, return 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 "%" 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.

Built-In Exceptions

Exception
Root class for all exceptions
SystemExit
On 'sys.exit()'
StandardError
Base class for all built-in exceptions; derived from Exception root class.
ArithmeticError
Base class for OverflowError, ZeroDivisionError, FloatingPointError
OverflowError
On excessively large arithmetic operation
ZeroDivisionError
On division or modulo operation with 0 as 2nd arg
FloatingPointError
when a floating point operation fails.

AssertionError
When an *assert* statement fails.
AttributeError
On attribute reference or assignment failure
EOFError
Immediate end-of-file hit by input() or raw_input()
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.
ImportError
On failure of 'import' to find module or name
KeyboardInterrupt
On user entry of the interrupt key (often 'Control-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
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
SystemError
On non-fatal interpreter error - bug - report it
TypeError
On passing inappropriate type to built-in op or func
ValueError
On arg error not covered by TypeError or more precise

Standard methods & operators redefinition in classes

```
Standard methods & operators map to methods in user-defined classes and thus may be redefined, e.g.
    class x:
         def __init__(self, v): self.value = v
         def __add__(self, r): return self.value + r
    a = x(3) # sort of like calling x.__init__(a, 3)
    a + 4    # is equivalent to a.__add__(4)
```

Special methods for any class

(s: self, o: other)

```
__init__(s, args) object instantiation

__del__(s)        called on object demise (refcount becomes 0)
__repr__(s)       repr() and `...` conversions
__str__(s)        str() and 'print' statement
__cmp__(s, o)     implements <, ==, >, <=, <>, !=, >=, is [not]
__hash__(s)       hash() and dictionary operations
__getattr__(s, name)  called when attr lookup doesn't find <name>
__setattr__(s, name, val) called when setting an attr
                          (inside, don't use "self.name = value"
                           use "self.__dict__[name] = val")
__delattr__(s, name)  called to delete attr <name>

__call__(self, *args) called when an instance is called as function.
```

Operators

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

Numeric operations special methods
(s: self, o: other)

```
s+o       =  __add__(s,o)          s-o        =  __sub__(s,o)
s*o       =  __mul__(s,o)          s/o        =  __div__(s,o)
s%o       =  __mod__(s,o)          divmod(s,o) = __divmod__(s,o)
pow(s,o)  =  __pow__(s,o)
s&o       =  __and__(s,o)
s^o       =  __xor__(s,o)          s|o        =  __or__(s,o)
s<<o      =  __lshift__(s,o)       s>>o       =  __rshift__(s,o)
nonzero(s) = __nonzero__(s) (used in boolean testing)
-s        =  __neg__(s)            +s         =  __pos__(s)
abs(s)    =  __abs__(s)            ~s         =  __invert__(s)  (bitwise)
```

```
int(s)    =  __int__(s)          long(s)    =  __long__(s)
float(s)  =  __float__(s)
oct(s)    =  __oct__(s)          hex(s)     =  __hex__(s)
coerce(s,o) = __coerce__(s,o)

Right-hand-side equivalents for all binary operators exist;
are called when class instance is on r-h-s of operator:
a + 3  calls __add__(a, 3)
3 + a  calls __radd__(a, 3)
```

All seqs and maps, general operations plus:
(s: self, i: index or key)

```
len(s)    = __len__(s)        length of object, >= 0.  Length 0 == false
s[i]      = __getitem__(s,i)  Element at index/key i, origin 0
```

Sequences, general methods, plus:

```
  s[i]=v          = __setitem__(s,i,v)
  del s[i]        = __delitem__(s,i)
  s[i:j]          = __getslice__(s,i,j)
  s[i:j]=seq      = __setslice__(s,i,j,seq)
  del s[i:j]      = __delslice__(s,i,j)   == s[i:j] = []
  seq * n         = __repeat__(seq, n)
  s1 + s2         = __concat__(s1, s2)
```

Mappings, general methods, plus

```
  hash(s)         = __hash__(s) - hash value for dictionary references
  s[k]=v          = __setitem__(s,k,v)
  del s[k]        = __delitem__(s,k)
```

Special informative state attributes for some types:

Lists & Dictionaries:
__methods__ (list, R/O): list of method names of the object

Modules:
__dict__ (dict, R/O): module's name space
__doc__ (string/None, R/O): doc string (<=> __dict__['__doc__'])
__name__(string, R/O): module name (also in __dict__['__name__'])
__file__(string/undefined, R/O): pathname of .pyc, .pyo or .pyd (undef for
modules statically linked to the interpreter)

Classes: [bold: writable since 1.5.2]
__bases__ (tuple, R/**W**): parent classes
__dict__ (dict, R/**W**): attributes (class name space)
__name__(string, R/**W**): class name (also in __dict__['__name__'])
__doc__ (string/None, R/W): doc string (<=> __dict__['__doc__'])

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

User-defined functions: [bold: writable since 1.5.2]
__doc__ (string/None, R/**W**): doc string
__name__(string, R/O): function name
func_defaults (tuple/None, R/**W**): default args values if any
func_code (code, R/**W**): code object representing the compiled function body
func_globals (dict, R/O): ref to dictionary of func global variables
func_doc (R/**W**): same as __doc__

User-defined Methods:

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

Built-in Functions & methods:

```
    __doc__ (string/None, R/O): doc string
    __name__ (string, R/O): function name
    __self__ : [methods only] target object
    __members__ = list of attr names: ['__doc__','__name__','__self__']
```

Codes:

```
    co_argcount (int, R/0): number of positional args
    co_nlocals (int, R/O): number of local vars (including args)
    co_varnames (tuple, R/O): names of local vars (starting with args)
    co_code (string, R/O): sequence of bytecode instructions
    co_consts (tuple, R/O): litterals used by the bytecode, 1st one is
                            fct doc (or None)
    co_name (string, R/O): name of function
    co_names (tuple, R/O): names used by the bytecode
    co_filename (string, R/O): filename from which the code was compiled
    co_flags (int, R/O): flags for the interpreter
                      bit 2 set if fct uses "*arg" syntax
                      bit 3 set if fct uses '**keywords' syntax
       HINT: use codehack.getlineno() to get 1st lineNo of a code object
```

Frames:

```
    f_back (frame/None, R/O): previous stack frame (towards 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
```

Tracebacks:

```
    tb_next (frame/None, R/O): next level in stack trace (towards 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 occured
    tb_lasti (int, R/O): precise instruction (index into bytecode)
```

Slices:

```
    start (any/None, R/O): lowerbound
    stop (any/None, R/O): upperbound
    step (any/None, R/O): step value
```

Complex numbers:

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

XRanges:

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

Important Modules

sys

Some variables:

```
argv            -- The list of command line arguments passed to a
                   Python script. sys.argv[0] is the script name.
builtin_module_names
                -- A list of strings giving the names of all modules
                   written in C that are linked into this interpreter.

check_interval  -- How often to check for thread switches or signals
                   (measured in number of virtual machine instructions)
exc_type
exc_value
exc_traceback   -- Deprecated since release 1.5. Use exc_info() instead.

exitfunc        -- User can set to a parameterless fcn. It will get
                   called before interpreter exits.
last_type
last_value
last_traceback  -- Set only when an exception not handled and
                   interpreter prints an error. Used by debuggers.

maxint          -- maximum positive value for integers

modules         -- Dictionary of modules that have already been loaded.

path            -- Search path for external modules. Can be modified
                   by program.
platform        -- The current platform, e.g. "sunos5", "win32"
ps1
ps2             -- prompts to use in interactive mode.

stdin
stdout
stderr          -- File objects used for I/O. User can redirect by
                   assigning a new file object to them (or any object:
                     .with a method write(string) for stdout/stderr,
                     .with a method readline() for stdin)

version         -- string containing version info about Python interpreter.

(and also: copyright, dllhandle, exec_prefix, prefix)
```

Functions:

```
exit(n)         -- Exit with status <n>. Raises SystemExit exception.
                   (Hence can be caught and ignored by program)
getrefcount(object)
                -- Return the reference count of the object. Generally 1 higher
                   than you might expect, because of <object> arg temp reference.
setcheckinterval(interval)
                -- Set the interpreter's thread switching interval (in number of virtual
                   code instructions, default:10).
settrace(func)  -- Sets a trace function: called before each line of
                   code is exited.
setprofile(func)
                -- Sets a profile function for performance profiling.
```

```
exc_info()      -- info on exception currently being handled; this is a
                   tuple (exc_type, exc_value, exc_traceback).
                   Warning: assigning the traceback return value to a local
                   variable in a function handling an exception will cause
                   a circular reference.
```

---

os

"synonym" for whatever O/S-specific module is proper for current environment. Uses "posix" whenever possible.

Variables

name -- name of O/S-specific module (e.g. "posix", "mac", "nt")
path -- O/S-specific module for path manipulations.
on Unix, os.path.split() <=> posixpath.split()
curdir -- string used to represent current directory ('.')
pardir -- string used to represent parent directory ('..')
sep -- string used to separate directories ('/' or '\')
pathsep -- character used to separate search path components (as in $PATH), eg. ';' for windows.
linesep -- [1.5.2] line separator as used in **binary** files, ie '\n' on Unix, '\r\n' on Dos/Win, '\r' on Mac

Functions

makedirs(path[, mode=0777]) [**new** in 1.5.2]
-- Recursive directory creation (create required intermediary dirs); os.error if fails.
removedirs(path) [**new** in 1.5.2]
-- Recursive directory creation (create required intermediary dirs); os.error if fails.
renames(old, new) [**new** in 1.5.2]
-- Recursive directory or file renaming; os.error if fails.

---

posix

don't import this module directly, import *os* instead !

Variables:

```
environ         -- dictionary of environment variables, e.g.
                   posix.environ['HOME']. [Windows: before release 1.52 case is significant; from 1.52,
                   os.environ is all uppercase but accesses are case insensitive]

error           -- exception raised on POSIX-related error.
                   Corresponding value is tuple of errno code and
                   perror() string.
```

Some Functions (see doc for more):

```
chdir(path)     -- change current directory to <path>.
chmod(path, mode)
                -- change the mode of <path> to the numeric <mode>
close(fd)       -- Close file descriptor <fd> opened with posix.open.
_exit(n)        -- Immediate exit, with no cleanups, no SystemExit,
                   etc. Should use this to exit a child process.
execv(p, args)  -- "Become" executable <p> with args <args>
getcwd()        -- return a string representing the current working directory
getpid()        -- return the current process id
```

```
fork()          -- Like C's fork(). Returns 0 to child, child pid to parent.
                   [Not on Windows]
kill(pid, signal)
                -- Like C's kill [Not on Windows]
listdir(path)   -- List names of entries in directory <path>.
lseek(fd, pos, how)
                -- set current position in file <fd> to position <pos>, expressed
                   as an offset relative to beginning of file (how=0), to
                   current position (how=1), or to end of file (how=2)
mkdir(path[, mode])
                -- Creates a directory named <path> with numeric <mode>
                   (default 0777)
open(file, flags, mode)
                -- Like C's open(). Returns file descriptor. Use file object fcts
                   rather than this low level ones.
pipe()          -- Creates pipe. Returns pair of file descriptors (r, w) [Not on Windows].
popen(command, mode='r', buffSize=0)
                -- Open a pipe to or from <command>. Result is a file
                   object to read to or write from, as indicated by
                   <mode> being 'r' or 'w'.
remove(path)    -- see unlink.
rename(src, dst)-- Rename/move the file or directory <src> to <dst>. [error if
                   target name already exists]
rmdir(path)     -- Remove the directory <path>
read(fd, n)     -- Read <n> bytes from <fd> and return as string.
stat(path)      -- Returns st_mode, st_ino, st_dev, st_nlink, st_uid,
                   st_gid, st_size, st_atime, st_mtime, st_ctime.
                   [st_ino, st_uid, st_gid are dummy on Windows]
system(command) -- Execute string <command> in a subshell. Returns exit
                   status of subshell.
times()         -- return accumulated CPU times in sec (user, system, children's user,
                   children's sys, elapsed real time). [3 last not on Windows]
unlink(path)    -- Unlink ("delete") path/file. same as: remove
utime(path, (aTime, mTime))
                -- Set the access & modified time of the file to the given tuple of values.
wait()          -- Wait for child process completion. Returns tuple of
                   pid, exit_status [Not on Windows]
waitpid(pid, options)
                -- Wait for process pid to complete. Returns tuple of
                   pid, exit_status [Not on Windows]
write(fd, str)  -- Write <str> to <fd>. Returns num bytes written.
```

---

posixpath

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

Some Functions (see doc for more):

```
exists(p)       -- True if string <p> is an existing path
expanduser(p)   -- Returns string that is (a copy of) <p> with "~" expansion done.
expandvars(p)   -- Returns string that is (a copy of) <p> with environment vars expanded.
                   [Windows: case significant; Use Unix: $var notation,not %var%]
getsize(filename)          [new in 1.5.2]              -- return the size in bytes of <filename>. raise os.error.
getmtime(filename)         [new in 1.5.2]              -- return last modification time of <filename> (integer nb of seconds since epoch).
getatime(filename)         [new in 1.5.2]              -- return last access time of <filename> (integer nb of seconds since epoch).
isabs(p)        -- True if string <p> is an absolute path.
isdir(p)        -- True if string <p> is a directory.
islink(p)       -- True if string <p> is a symbolic link.
ismount(p)      -- True if string <p> is a mount point [true for all dirs on Windows].
join(p[,q[,...]])
                -- Joins one or more path components intelligently.
split(p)        -- Splits into (head, tail) where <tail> is last
                   pathname component and <head> is everything leading
                   up to that. <=> (dirname(p), basename(p))
splitdrive(p)   -- Splits path in a pair ('drive:', tail) [on Windows]
splitext(p)     -- Splits into (root, ext) where last comp of <root>
                   contains no periods and <ext> is empty or starts
                   with a period.
walk(p, visit, arg)
```

```
               -- Calls the function <visit> with arguments
                  (<arg>, <dirname>, <names>) for each directory recursively in
                   the directory tree rooted at <p> (including p itself if it's a dir)
                  The argument <dirname> specifies the visited directory, the argument
                  <names> lists the files in the directory.  The <visit> function may
                  modify <names> to influence the set of directories visited below
                  <dirname>, e.g., to avoid visiting certain parts of the tree.
[1.52, NT version: samefile(), sameopenfile(), samestat() deprecated because not reliable]
```

---

time

Variables

```
altzone         -- signed  offset of local DST timezone in sec west of the 0th meridian.
daylight        -- nonzero if a DST timezone is specified
```

Functions

```
time()         -- return a float representing UTC time in seconds since the epoch.
gmtime(secs), localtime(secs)
               -- return 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))
asctime(timeTuple),
strftime(format, timeTuple)
               -- return a formated string representing time.
mktime(tuple)  -- inverse of localtime(). Return a float.
strptime(string[, format])       [new in 1.5.2]            -- parse a formated string representing time, return tuple as in gmtime().
sleep(secs)    -- Suspend execution for <secs> seconds. <secs> can be a float.
```

and also: clock, ctime.

---

string

Some Variables:

```
digits                    -- The string '0123456789'
hexdigits, octdigits      -- legal hexadecimal & octal digits
letters
uppercase
lowercase
whitespace
                          -- Strings containing the appropriate characters
index_error               -- Exception raised by index() if substr not found.
```

Some Functions:

```
expandtabs(s, tabSize)  -- returns a copy of string <s> with tabs expanded.
find/rfind(s, sub[, start=0[, end=0])
                        -- Return the lowest/highest index in <s> where the substring
                           <sub> is found such that <sub> is wholly contained in
                           s[start:end]. Return -1 if <sub> not found.
ljust/rjust/center(s, width)
                        -- Return a copy of string <s> left/right justified/centerd in a
                           field of given width, padded with spaces. <s> is never
                           truncated.
lower/upper(s)          -- Return a string that is (a copy of) <s> in lowercase/uppercase
split(s[, sep=whitespace[, maxsplit=0]])
                        -- Return a list containing the words of the string <s>,
                           using the string <sep> as a separator.
join(words[, sep=' '])  -- Concatenate a list or tuple of words with intervening
                           separators; inverse of split.
replace(s, old, new[, maxsplit=0]
                        -- Returns a copy of string <s> with all occurences of substring
```

```
                          <old> replaced by <new>. Limits to <maxsplit> first
                          substitutions if specified.
strip(s)                -- Return a string that is (a copy of) <s> without leading
                          and trailing whitespace. see also lstrip, rstrip.
```

---

re

Patterns are specified as strings. Tip: Use raw strings (e.g. r'\w*') to litteralize backslashes.

Regular Expression Syntax:

```
.       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 occurence 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 \w \S)
[^ ]    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 '\\' or
        r'\' in the pattern string.
\\      matches a litteral '\'; due to Python string rules, write as '\\\\'
        in pattern string, or better using raw string: r'\\'.
|       specifies alternative: 'foo|bar' matches 'foo' or 'bar'
(...)   matches any RE inside (), and delimits a group.
(?:...) idem but does'nt delimit a group.
(?=...) 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 (?=...)
(?P<name>...) matches any RE inside (), and delimits a named group.
              (e.g. r'(?P<id>[a-zA-Z_]\w*)' defines a group named id)
(?P=name) matches whatever text was matched by the earlier group named name.
(?#...) A comment; ignored.
(?letter) letter is one of 'i','L', 'm', 's', 'x'. Set the corresponding flags
        (re.I, re.L, re.M, re.S, re.X) for the entire RE.

                            SPECIAL SEQUENCES:
\number matches content of the group of the same number; groups are numbered
        starting from 1
\A      matches only at the start of the string
\b      empty str at beg or end of word: '\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 (<=> [^O-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:

```
error           -- Exception when pattern string isn't a valid regexp.
```

Functions:

```
compile(pattern[, flags=0])
            -- Compile a RE pattern string into a regular expression object.
               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)

escape(string)    -- return (a copy of) string with all non-alphanumerics backslashed.
match(pattern, string[, flags])
         -- if 0 or more chars at beginning of <string> match the RE pattern string,
            return a corresponding MatchObject instance, or None if no match.
search(pattern, string[, flags])
         --scan thru <string> for a location matching <pattern>, return a
           corresponding MatchObject instance, or None if no match.
split(pattern, string[, maxsplit=0])
         --split <string> by occurrences of <pattern>. If capturing () are used in
           pattern, then occurrences of patterns or subpatterns are also returned.
findall(pattern, string)  [new in 1.5.2]
         --return a list of non-overlapping matches in <pattern>, either a list of
           groups or a list of tuples if the pattern has more than 1 group.
sub(pattern, repl, string[, count=0])
         --return string obtained by replacing the (<count> first) lefmost non-
           overlapping occurrences of <pattern> (a string or a RE object) in <string>
           by <repl>; <repl> can be a string or a fct called with a single MatchObj          arg, which must return the replacement string.
subn(pattern, repl, string[, count=0])

         --same as sub(), but returns a tuple (newString, numberOfSubsMade)
```

Regular Expression Objects

(RE objects are returned by the compile fct)

Attributes:

```
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
```

Methods:

```
match(string[, pos][, endpos])
search(string[, pos][, endpos])
split(string[, maxsplit=0])
findall(string)    [new in 1.5.2]
sub(repl, string[, count=0])
subn(repl, string[, count=0]) -- see equivalent functions.
```

Match Objects

(Match objects are returned by the match & search functions)

Attributes:

```
pos         -- value of pos passed to search or match functions; index into
               string at which RE engine started search.
endpos      -- value of endpos passed to search or match functions; index into
               string beyond which RE engine won't go.
re          -- RE object whose match or search fct produced this MatchObj instance
string      -- string passed to match() or search()
```

Methods:

```
group([g1, g2, ...])
             -- 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 gi is 0,
                return value is entire matching string; if 1 <= gi <= 99, return
                string matching group #gi (or None if no such group); gi may also be
                a group name.
groups()     -- returns a tuple of all groups of the match; groups not participating
                to the match have a value of None. Returns a string instead of tuple
                if len(tuple)=1
start(group)
end(group)   -- returns indices of start & end of substring matched by group (or None
                if group exists but doesn't contribute to the match)
span(group)  -- returns the 2-tuple (start(group), end(group)); can be (None, None)
                if group didn't contibute to the match.
```

---

math

Variables:

```
pi
e
```

Functions (see ordinary C man pages for info):

```
acos(x)
asin(x)
atan(x)
atan2(x, y)
ceil(x)
cos(x)
cosh(x)
exp(x)
fabs(x)
floor(x)
fmod(x, y)
frexp(x)         -- Unlike C: (float, int) = frexp(float)
ldexp(x, y)
log(x)
log10(x)
modf(x)          -- Unlike C: (float, float) = modf(float)
pow(x, y)
sin(x)
sinh(x)
sqrt(x)
tan(x)
tanh(x)
```

---

getopt

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 in base distribution

Contents of Lib directory
(Python 1.52 NT distribution, may be slightly different in other distributions)

```
aifc            --  Stuff to parse AIFF-C and AIFF files.
anydbm          --  Generic interface to all dbm clones. (dbhash, gdbm, dbm,dumbdbm)
asynchat        --  Support for 'chat' style protocols
asyncore        --  Asynchronous File I/O (in select style)
audiodev        --  Audio support for a few platforms.
base64          --  Conversions to/from base64 RFC-MIME transport encoding .
BaseHTTPServer  --  Base class forhttp services.
Bastion         --  "Bastionification" utility (control access to instance vars)
bdb             --  A generic Python debugger base class.
binhex          --  Macintosh binhex compression/decompression.
bisect          --  List bisection algorithms.
calendar        --  Calendar printing functions.
cgi             --  Wraps the WWW Forms Common Gateway Interface (CGI).
CGIHTTPServer   --  CGI http services.
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
colorsys        --  Conversion functions between RGB and other color systems.
commands        --  Tools for executing UNIX commands .
compileall      --  Force "compilation" of all .py files in a directory.
ConfigParser    --  Configuration file parser (much like windows .ini files)
copy            --  Generic shallow and deep copying operations.
copy_reg        --  Helper to provide extensibility for pickle/cPickle.
dbhash          --  (g)dbm-compatible interface to bsdhash.hashopen.
dircache        --  Sorted list of files in a dir, using a cache.
dircmp          --  Defines a class to build directory diff tools on.
dis             --  Bytecode disassembler.
dospath         --  Common operations on DOS pathnames.
dumbdbm         --  A dumb and slow but simple dbm clone.
dump            --  Print python code that reconstructs a variable.
exceptions      --  Class based built-in exception hierarchy.
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       --  A test formatter.
fpformat        --  General floating point formatting functions.
ftplib          --  An FTP client class.  Based on RFC 959.
getopt          --  Standard command line processing.
getpass         --  Utilities to get a password and/or the current user name.
glob            --  filename globbing.
gopherlib       --  Gopher protocol client interface.
grep            --  'grep' utilities.
```

```
gzip            --  Read & write gzipped files.
htmlentitydefs  --  Proposed entity definitions for HTML.
htmllib         --  HTML parsing utilities.
httplib         --  HTTP client class.
ihooks          --  Hooks into the "import" mechanism.
imaplib         --  IMAP4 client.Based on RFC 2060.
imghdr          --  Recognizing image files based on their first few bytes.
keyword         --  List of Python keywords.
knee            --  A Python re-implementation of hierarchical module import.
linecache       --  Cache lines from files.
locale          --  Support for number formatting using the current locale settings.
macpath         --  Pathname (or related) operations for the Macintosh.
macurl2path     --  Mac specific module for conversion between pathnames and URLs.
mailbox         --  A class to handle a unix-style or mmdf-style mailbox.
mailcap         --  Mailcap file handling (RFC 1524).
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.
mimify          --  Mimification and unmimification of mail messages.
multifile       --  Class to make multi-file messages easier to handle.
mutex           --  Mutual exclusion -- for use with module sched.
netrc           --
nntplib         --  An NNTP client class.  Based on RFC 977.
ntpath          --  Common operations on DOS pathnames.
nturl2path      --  Mac specific module for conversion between pathnames and URLs.
os              --  Either mac, dos or posix depending system.
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: cPickle).
pipes           --  Conversion pipeline templates.
poly            --  Polynomials.
popen2          --  variations on pipe open.
poplib          --  A POP3 client class. Based on the J. Myers POP3 draft.
posixfile       --  Extended (posix) file operations.
posixpath       --  Common operations on POSIX pathnames.
pprint          --  Support to pretty-print lists, tuples, & dictionaries recursively.
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.
Queue           --  A multi-producer, multi-consumer queue.
quopri          --  Conversions to/from quoted-printable transport encoding.
rand            --  Don't use unless you want compatibility with C's rand().
random          --  Random variable generators (obsolete, use whrandom)
re              --  Regular Expressions.
reconvert       --  Convert old ("regex") regular expressions to new syntax ("re").
regex_syntax    --  Flags for regex.set_syntax().
regexp          --  Backward compatibility for module "regexp" using "regex".
regsub          --  Regular expression subroutines.
repr            --  Redo repr() but with limits on most sizes.
rexec           --  Restricted execution facilities ("safe" exec, eval, etc).
rfc822          --  RFC-822 message manipulation class.
rlcompleter     --  Word completion for GNU readline 2.0.
sched           --  A generally useful event scheduler class.
sgmllib         --  A parser for SGML.
shelve          --  Manage shelves of pickled objects.
shlex           --  Lexical analyzer class for simple shell-like syntaxes.
shutil          --  Utility functions usable in a shell-like program.
SimpleHTTPServer--  Simple extension to base http class
site            --  Append module search paths for third-party packages to sys.path.
smtplib         --  SMTP Client class (RFC 821)
sndhdr          --  Several routines that help recognizing sound.
```

```
SocketServer    --  Generic socket server classes.
stat            --  Constants and functions for interpreting stat/lstat struct.
statcache       --  Maintain a cache of file stats.
statvfs         --  Constants for interpreting statvfs struct as returned by os.statvfs()
                    and os.fstatvfs() (if they exist).
string          --  A collection of string operations.
StringIO        --  File-like objects that read/write a string buffer (a faster
                    C implementation exists in built-in module: cStringIO).
sunau           --  Stuff to parse Sun and NeXT audio files.
sunaudio        --  Interpret sun audio headers.
symbol          --  Non-terminal symbols of Python grammar (from "graminit.h").
telnetlib       --  TELNET client class. Based on RFC 854.
tempfile        --  Temporary file name allocation.
threading       --  Proposed new higher-level threading interfaces
threading_api   --  (doc of the threading module)
toaiff          --  Convert "arbitrary" sound files to AIFF files .
token           --  Tokens (from "token.h").
tokenize        --  Compiles a regular expression that recognizes Python tokens.
traceback       --  Format and print Python stack traces.
tty             --  Terminal utilities.
turtle          --  LogoMation-like turtle graphics
types           --  Define names for all type symbols in the std interpreter.
tzparse         --  Parse a timezone specification.
urllib          --  Open an arbitrary URL.
urlparse        --  Parse URLs according to latest draft of standard.
user            --  Hook to allow user-specified customization code to run.
UserDict        --  A wrapper to allow subclassing of built-in dict class.
UserList        --  A wrapper to allow subclassing of built-in list class.
util            --  some useful functions that don't fit elsewhere !!
uu              --  UUencode/UUdecode.
wave            --  Stuff to parse WAVE files.
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.
zmod            --  Demonstration of abstruse mathematical concepts.
```

(following list not revised)

* Built-ins *

```
sys                 Interpreter state vars and functions
__built-in__        Access to all built-in python identifiers
__main__            Scope of the interpreters main program, script or stdin
array               Obj efficiently representing arrays of basic values
math                Math functions of C standard
time                Time-related functions
regex               Regular expression matching operations
marshal             Read and write some python values in binary format
struct              Convert between python values and C structs
```

* Standard *

```
getopt              Parse cmd line args in sys.argv.  A la UNIX 'getopt'.
os                  A more portable interface to OS dependent functionality
re                  Functions useful for working with regular expressions
string              Useful string and characters functions and exceptions
whrandom            Wichmann-Hill pseudo-random number generator
thread              Low-level primitives for working with process threads
threading           idem, new recommanded interface.
```

* Unix/Posix *

```
dbm                 Interface to Unix ndbm database library
grp                 Interface to Unix group database
posix               OS functionality standardized by C and POSIX standards
posixpath           POSIX pathname functions
pwd                 Access to the Unix password database
select              Access to Unix select multiplex file synchronization
socket              Access to BSD socket interface
```

* Tk User-interface Toolkit *

```
tkinter             Main interface to Tk
```

* Multimedia *

```
audioop             Useful operations on sound fragments
imageop             Useful operations on images
jpeg                Access to jpeg image compressor and decompressor
rgbimg              Access SGI imglib image files
```

* Cryptographic Extensions *

```
md5         Interface to RSA's MD5 message digest algorithm
mpz         Interface to int part of GNU multiple precision library
rotor               Implementation of a rotor-based encryption algorithm
```

* Stdwin * Standard Window System

```
stdwin              Standard Window System interface
stdwinevents        Stdwin event, command, and selection constants
rect                Rectangle manipulation operations
```

* SGI IRIX * (4 & 5)

```
al          SGI audio facilities
AL          al constants
fl          Interface to FORMS library
FL          fl constants
flp Functions for form designer
fm          Access to font manager library
gl          Access to graphics library
GL          Constants for gl
DEVICE      More constants for gl
imgfile     Imglib image file interface
```

* Suns *

```
sunaudiodev Access to sun audio interface
```

Workspace exploration and idiom hints

```
dir(<module>)   list functions, variables in <module>
dir()           get object keys, defaults to local name space
X.__methods__   list of methods supported by X (if any)
X.__members__   List of X's data attributes
if __name__ == '__main__': main()            invoke main if running as script
map(None, lst1, lst2, ...)                   merge lists
b = a[:]                                     create copy of seq structure
_                       in interactive mode, is last value printed
```

Python Mode for Emacs

(Not revised, possibly not up to date)

```
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
```

The Python Debugger

(Not revised, possibly not up to date, see 1.5.2 Library Ref section 9.1; in 1.5.2, you may also use debugger integrated

Accessing

```
import pdb      (it's a module written in Python)

        -- defines functions :           run(statement[,globals[, locals]])
                        -- execute statement string under debugger control, with optional
                           global & local environment.
           runeval(expression[,globals[, locals]])
                        -- same as run, but evaluate expression and return value.
           runcall(function[, argument, ...])
                        -- run function object with given arg(s)
           pm()         -- run postmortem on last exception (like debugging a core file)
           post_mortem(t)
                        -- run postmortem on traceback object <t>

        -- defines class Pdb :           use Pdb to create reusable debugger objects. Object
           preserves state (i.e. break points) between calls.


        runs until a breakpoint hit, exception, or end of program
        If exception, variable '__exception__' holds (exception,value).
```

Commands

```
h, help
        brief reminder of commands
b, break [<arg>]
        if <arg> numeric, break at line <arg> in current file
        if <arg> is function object, break on entry to fcn <arg>
        if no arg, list breakpoints
cl, clear [<arg>]
        if <arg> numeric, clear breakpoint at <arg> in current file
        if no arg, clear all breakpoints after confirmation
w, where
        print current call stack
```

```
u, up
        move up one stack frame (to top-level caller)
d, down
        move down one stack frame
s, step
        advance one line in the program, stepping into calls
n, next
        advance one line, stepping over calls
r, return
        continue execution until current function returns
        (return value is saved in variable "__return__", which
        can be printed or manipulated from debugger)
c, continue
        continue until next breakpoint
a, args
        print args to current function
rv, retval
        prints return value from last function that returned
p, print <arg>
        prints value of <arg> in current stack frame
l, list [<first> [, <last>]]
               List source code for the current file.
               Without arguments, list 11 lines around the current line
               or continue the previous listing.
               With one argument, list 11 lines starting at that line.
               With two arguments, list the given range;
               if the second argument is less than the first, it is a count.
whatis <arg>
        prints type of <arg>
!
        executes rest of line as a Python statement in the current stack frame
q quit
        immediately stop execution and leave debugger

<return>
        executes last command again

Any input debugger doesn't recognize as a command is assumed to be a
Python statement to execute in the current stack frame, the same way
the exclamation mark ("!") command does.
```

Example

```
(1394) python
Python 1.0.3 (Sep 26 1994)
Copyright 1991-1994 Stichting Mathematisch Centrum, Amsterdam
>>> import rm
>>> rm.run()
Traceback (innermost last):
         File "<stdin>", line 1
         File "./rm.py", line 7
           x = div(3)
         File "./rm.py", line 2
           return a / r
ZeroDivisionError: integer division or modulo
>>> import pdb
>>> pdb.pm()
> ./rm.py(2)div: return a / r
(Pdb) list
         1     def div(a):
         2  ->     return a / r
```

```
          3
          4     def run():
          5         global r
          6         r = 0
          7         x = div(3)
          8         print x
[EOF]
(Pdb) print r
0
(Pdb) q
>>> pdb.runcall(rm.run)
etc.
```

Quirks

Breakpoints are stored as filename, line number tuples. If a module is reloaded after editing, any remembered breakpoints are likely to be wrong.

Always single-steps through top-most stack frame. That is, "c" acts like "n".