A Python Quick Reference

Postscript version v1.1.1, by Anthony Baxter,
`<anthony@aaii.oz.au>`
`Based on:`
`ASCII v1.0; 1994/09/27`
Author: Chris Hoffmann,
`choffman@vicorp.com`
Based on
- Python Bestiary,
  by Ken Manheimer,
  `(ken.manheimer@nist.gov)`
- Python manuals,
  by Guido van Rossum,
  `(guido@cwi.nl)`
- python-mode.el,
  by Tim Peters,
  `(tim@ksr.com)`

Invocation Options

`python [-diuv] [-c command | script | - ] [args]`

`-d` Turn on parser debugging output (for wizards only, depending on compilation options).

-i When a script is passed as first argument or the -c option is used, enter interactive mode after executing the script or the command. It does not read the `$PYTHONSTARTUP` file. This can be useful to inspect global variables or a stack trace when a script raises an exception.

`-u` Force stdout and stderr to be totally unbuffered.

`-v` Print a message each time a module is initialized, showing the place (filename or built-in module) from which it is loaded.

-c `command`
Specify the command to execute *(see next section)*. This terminates the option list (following options are passed as arguments to the command).

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

script is the name of a python file to execute

`args` are passed to script or command (in `sys.argv`)

If no script or command, Python enters interactive mode. Uses “`readline`” package for input, if available.

Environment Variables

`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 colons.

`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.

`PYTHONDEBUG`
If non-empty, same as `-d` option

`PYTHONINSPECT`
If non-empty, same as `-i` option

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

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

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: not literally the string “x” but some 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 elif from lambda return
break else global not try class
except if or while continue
exec import pass def finally in
print del or is raise
```

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.*
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|"_")*`

Strings:

```
"a string" 'another string'
'''a string containing embedded
newlines, and quote (') marks,
can be delimited with triple
quotes.'''
```

String Literal Escapes

\newline	Ignored (escape newline)
`\\`	Literal backslash (\)
`\e`	Escape (ESC)
`\v`	Vertical Tab (VT)
`\'`	Single quote (')
`\f`	Formfeed (FF)
`\OOO`	(zero) char with value 0
`\"`	Double quote (")
`\n`	Linefeed (LF)
\octal value OO	
`\a`	Bell (BEL)
`\r`	Carriage Return (CR)
`\xXX`	char with hex value XX
\b	Backspace (BS)
\t	Horizontal Tab (TAB)

\<any other char> is left as-is
`NULL` byte (`\000`) is <u>not</u> an end-of-string marker; `NULL`'s may be imbedded in strings
Strings (and tuples) are immutable: they cannot be modified.

Other types:

long integer (unlimited precision):
    `1234567890L`
octal integer:
    `0177, 0177777777777777L`
hex integer:
    `0xFF, 0xFFFFFFFFFFFFFL`
float:
    `3.14e-10`
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]`
dictionary of length 0, 1, 2, etc:
    `{} {1 : 'one'} {1 : 'one', 'next': '2nd'}`
(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.)

Basic Types and Their Operations

Comparisions (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 <u>objects</u> 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)*

Predefined object of special type: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
- `pow(x,y)` x to the power y
- `divmod(x,y)` the tuple `(x/y,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

Numeric exceptions

`TypeError`: raised on application of arithemetic opertion to non-number

`OverflowError`: numeric bounds exceeded

`ZeroDivisionError`: raised when zero second argument of div or modulo op

Operators on all sequence types (lists, tuples, strings)

- `len(s)` length of s
- `min(s)` smallest item of s
- `max(s)` largest item of s
- `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
- `s[i:j]` slice of s from i to j *(slice from index `i` up to but not including index `j`. `i` defaults to `0`, `j` to `len(s)`.Negative goes from right-end of sequence)*

Operators on mutable sequences (lists)

- `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]` delete slice (*same as* `s[i:j]=[]`)
- `s.append(x)` add `x` to end of `s`
- `s.count(x)` return number of `i`’s for which `s[i] == x`
- `s.index(x)` return smallest `i` such that `s[i] == x1)`

`s.insert(i, x)` item `i` becomes `x`, old item `i` is now at `i+1`, etc.
`s.remove(x)` same as `del s[s.index(x)]`
`s.reverse()` reverses the items of `s` (in place)
`s.sort()` sorts the list (in place) *(Optional parameter: function of two arguments returning* `-1`, `0` *or* `1` *depending on whether arg1 is >, ==, < arg2)*

`IndexError` is raised on out-of-range sequence subscript

Operations on mappings (dictionaries)

`len(a)`	the number of items in `a`
`a[k]`	the item of a with key `k`
`a[k] = x`	set `a[k]` to `x`
`del a[k]`	remove `a[k]` from `a`
`a.items()`	a copy of `a`'s list of (key, item) pairs
`a.keys()`	a copy of `a`'s list of keys
`a.values()`	a copy of `a`'s list of values
`a.has_key(k)`	`1` if `a` has a key `k`, else `0`

`TypeError` is raised if key not acceptable.
`KeyError` is raised if attempt is made to read with non-existent key

Format operator for strings (%)

Uses sprintf 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:

`f.close(x)` close file
`f.flush(x)` 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 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.

`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
Type 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
- Frame Objects
- Traceback Objects

Statements

pass Null statement

= assignment operator. Can unpack tuples, first,second = a[0:2]

del Unbind name from object, or attributes from objects, etc.

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 <glob> [,<loc>]]
executes <x> in namespace provided. Defaults to current namespace. <glob> is a dictionary containing global namespace, <loc> contains local namespace. <x> can be a string, file object or a function object.

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>. 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

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 an exception did occur, executes <suite2> and them immediately reraises exception.

raise <exception> [,<value>]
raises <exception> with optional parameter <value>.

An exception is simply a string (object). Create a new one simply by creating a new string:

```
my_exception = 'it went wrong'
try: if bad:
    raise my_exception, bad
except my_exception, value:
    print 'Oops', value
```

Name Space Statements

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

from <module_id> import <id1> [, <id2>]*
: imports names from module <module_id>. Names are not qualified:
```
from sys import argv
print argv"
```

from <module_id> import *
: imports all names in module <module_id>, except those starting with _

global <id1> [,<id2>]*
: ids are from global scope (usually meaning from module) rather than local (usually meaning only in function).
In a function with no "global" statements, assume a is name that hasn't been used in fcn 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 fcn, but is in module, then:
Try to read from a, gets value from module.
Try to write to a, changes a in module

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>]
: Parameters with "=" have default values (<v> is evaluated when function defined). If list ends with "*<id>" then <id> is assigned a tuple of all remaining args passed to function.*(allows vararg functions)*.

Class Definition

class <class_id> [(<super_class1> [,<super_class2>]*)]:
<suite>

Creates a class object and assigns it name <class_id>. <suite> may contain "def"s of class methods and assignments to class attributes

E.g.

class my_cl (cl1, cl_lst[3]):

Creates a class object inheriting from both cl1 and whatever class object cl_lst[3] evaluates to. Assigns new class object to name my_class.

First arg to class methods is always instance object. By convention this is called "self".

Special method __init__() called when instance created. Create instance by "calling" class object, possibly with args. In current implementation, you can't subclass off built-in classes.

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.

Others

lambda [<param_list>]: <condition>
: Create an anonymous function.

<condition> 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

abs(x)
: Return the absolute value of a number

apply(f,args)
: Call func/method <f> with args <args>

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 for error reporting, can be any string. <kind> is either 'eval' if <string> is a single stmt, else it should be 'exec'.

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')

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

float(x)
: Convert a number to floating point.

getattr(object, name)
: Get attr called <name> from <object>. getattr(x, 'foobar') ↔ x.foobar

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 to a plain integer.

len(s)
: Return the length (the number of items) of an object.

long(x)
: Convert a number 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, nega-

tive for sys-default, all else, of (about) given size.

`ord(c)` Return integer `ASCII` value of <`c`> (a string of len `1`).

`pow(x, y)` Return `x` to power `y`.

`range(start, end, step)`
: return list of ints from >= start and < end. With 1 arg, list from 0 to <`arg`>`-1`. With 2 args, list from <`start`> to <`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.

`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 "pre-pended" 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`

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

`type(object)`
: Return type of an object.

E.g.,

```
  if type(x) == type(''):
    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. Don't simply type `vars()` at interactive prompt! (But `print vars()` is fine.)

`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

`AttributeError`
: On attribute reference or assignment failure

`EOFError`
: Immediate end-of-file hit by input() or raw_input()

`IOError`
: I/O-related I/O operation failure

`ImportError`
: On failure of \`import' to find module or name

`IndexError`
: On out-of-range sequence subscript

`KeyError`
: On reference to a non-existent mapping (dict) key

`KeyboardInterrupt`
: On user entry of the interrupt key (often \`Control-C')

`MemoryError`
: On recoverable memory exhaustion

`NameError`
: On failure to find a local or global (unqualified) name

`OverflowError`
: On excessively large arithmetic operation

`RuntimeError`
: Obsolete catch-all; define a suitable error instead

`SyntaxError`
: On parser encountering a syntax error

`SystemError`
: On non-fatal interpreter error - bug - report it

`SystemExit`
: On `sys.exit()'

`TypeError`
: On passing inappropriate type to built-in op or func

`ValueError`
: On arg error not covered by TypeError or more precise

`ZeroDivisionError`
: On division or modulo operation with 0 as 2nd arg

Special Methods For User-Defined Classes

E.g.

```
class x:
  def __init__(self, v):
    self.value = v def
  __add__(self, r):
    return self.value + r
a = x(3)
```

(like calling `x.__init__(a,3))`

```
a + 4
```

(equivalent to `a.__add__(4))`

Special methods for any type

(s: self, o: other)

`__init__(s, args)`
: object instantiation

`__del__(s)`
: called on object demise

`__repr__(s)`
: `repr()` and `...` conversions

`__str__(s)`
: `str()` and '`print`' statement

`__cmp__(s, o)`
: implements <, ==, >, <=, <>, `!=`, >=, `is [not]`

`__hash__(s)`
: `hash()` and dict operations

Numeric operations vs 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)
```

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]=[]
```

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:

`X.__dict__`
: dict used to store object's writeable attributes

`X.__methods__`
: list of X's methods; on many built-in types.

`X.__members__`
: lists of X's data attributes

`X.__class__`
: class to which X belongs

`X.__bases__`
: tuple of X base classes

`M.__name__`
: r/o attr, module's name as string

Important Modules

sys

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.

`exc_type`
`exc_value`
`exc_traceback` Set when in an exception handler. Are last exception, last exception value, and traceback object of call stack when exception occured.

`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.

`modules` List of modules that have already been loaded.

`path` Search path for external modules. Can be modified by program.

`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()" taking string argument).

`tracebacklimit` Maximum levels of tb info printed on error.

Functions:

`exit(n)` Exit with status <n>. Raises SystemExit exception. (Hence can be caught and ignored by program)

`settrace(func)` Sets a trace function: called before each line of code is exited.

`setprofile(func)` Sets a profile function for performance profiling.

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` or `mac`)

`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 ('/')

posix

Variables:

`environ` dictionary of environment variables,
E.g.. `posix.environ['HOME']`

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

Some Functions (see doc for more):

`chdir(path)`
Go to `<path>`.

`close(fd)` Close file descriptor `<fd>`.

`_exit(n)` Immediate exit, with no clean-ups, no `SystemExit`, etc. Should use this to exit a child process.

`exec(p, args)`
"Become" executable <p> with args `<args>`

`fork()` Like C's `fork()`. Returns 0 to child, child pid to parent.

`kill(pid, signal)`
Like C's kill

`listdir(path)`
List names of entries in directory <path>.

`open(file, flags, mode)`
Like C's `open()`. Returns file descriptor.

`pipe()` Creates pipe. Returns pair of file descriptors (r, w).

`popen(command, mode)`
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'.

`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`.

`system(command)`
Execute string `<command>` in a subhell. Returns exit status of subshell.

`unlink(path)`
Unlink ("delete") path/file.

`wait()` Wait for child process completion. Returns tuple of pid, exit_status

`waitpid(pid, options)`
Wait for process pid to complete. Returns tuple of pid, exit_status

`write(fd, str)`
Write `<str>` to `<fd>`. Returns num bytes written.

posixpath

Some Functions (see doc for more):

`exists(p)` True if string `<p>` is an existing path

`expanduser(p)`
Returns string that is `<p>` with "~" expansion done.

`isabs(p)`
True if string `<p>` is an absolute path.

`isdir(p)`
True if string <p> is a directory.

`isfile(p)` True if string <p> is a regular file.

`islink(p)` True if string <p> is a symbolic link.

`isfile(p)` True if string <p> is a regular file.

`ismount(p)` True if string <p> is a mount point.

`split(p)` Splits into (head, tail) where `<tail>` is last pathname component and `<head>` is everything leading up to that.

`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 in the directory tree rooted at `<p>` 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.

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)`	
`ldexp(x,y)`	`log(x)`	`log10(x)`
`pow(x,y)`	`sin(x)`	`sinh(x)`
`sqrt(x)`	`tan(x)`	`tanh(x).`

`frexp(x)` – *Different than C:* `(float,int)=frexp(float)`

`modf(x)` – *Different than C:* `(float,float)=modf(float)`

string

Some Variables:

`digits` The string '0123456789'

`uppercase`

`lowercase`

`whitespace`

Strings containing the appropriate characters

`index_error`

Exception raised by `index()` if `substr` not found.

Some Functions:

`index(s, sub, i=0)`

Return the lowest index in <s> not smaller than <i> where the substring <sub> is found.

`lower(s)` Return a string that is <s> in lowercase

`splitfields(s, sep)`

Returns a list containing the fields of the string <s>, using the string <sep> as a separator.

`joinfields(words, sep)`

Concatenate a list or tuple of words with intervening separators.

`strip(s)` Return a string that is <s> without leading and trailing whitespace.

`upper(s)` Return a string that is <s> in uppercase

regex

Patterns are specified as strings. Default syntax is emacs-style.

Variables:

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

Functions:

`match(pattern, string)`

Return how many characters at the beginning of <string> match regexp <pattern>. −1 if none.

`search(pattern, string)`

Return the first position in <string> that matches regexp <pattern>. Return −1 if none.

`compile(pattern [,translate])`

Create regexp object that has methods match() and search() working as above. Also `group(i1, [,i2]*)`

E.g.

```
p= \
compile('id\([a-z]\)\([a-z]\)')
p.match('idab') ==> 4
p.group(1, 2) ==> ('a','b')
```

`set_sytax(flag)`

Set syntax flags for future calls to `match()`, `search()` and `compile()`. Returns current value. Flags in module `regex_syntax`.

`symcomp(pattern [,translate])`

Like compile but with symbolic group names. Names in angle brackets. Access through group method.

E.g.

```
p = \
symcomp('id\(<l1>[a-z]\)\(<l2>[
a-z]\)')
p.match('idab') ==> 4
p.group('l1') ==> 'a'
```

regex_syntax

Flags for regex.set_syntax().
BitOr the flags you want together and pass to function.
Variables:

RE_NO_BK_PARENS
: if set, (means grouping, \(is literal "("
: if not, vice versa

RE_NO_BK_VBAR
: if set, | means or, \| is literal "|"
: if not, vice versa

RE_BK_PLUS_QM
: if set, + or ? are operator, \+, \? are literal
: if not, vice versa

RE_TIGHT_VBAR -- if set, | binds tighter than ^ or $
: if not, vice versa

RE_NEWLINE_OR
: if set, \n is an OR operator
: if not, it is a normal char

RE_CONTEXT_INDEP_OPS
: if not set, special chars always have special meaning
: if set, depends on context:

^ - only special at the beginning, or after (or |
$ - only special at the end, or before) or |
*, +, ? - only special when not after the beginning, (, or |

```
RE_SYNTAX_AWK =
        (RE_NO_BK_PARENS |
        RE_NO_BK_VBAR |
        RE_CONTEXT_INDEP_OPS)
RE_SYNTAX_EGREP =
        (RE_SYNTAX_AWK |
        RE_NEWLINE_OR)
RE_SYNTAX_GREP = (RE_BK_PLUS_QM
        | RE_NEWLINE_OR)
RE_SYNTAX_EMACS = 0
```

reg_sub

Functions:

sub(pattern, rep, str)
: Replace 1st occur of <pattern> in <str> by <rep> and return this.

gsub(pattern, rep, str)
: Replace <u>all</u> occurances of <pattern> in <str> by <rep> and return this.

split(str, pattern)
: Split <str> into fields seperated by delimiters matching <pattern> and return as list of strings.

Other Modules In Base Distribution

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
rand	Pseudo-random generator, like C rand()
regsub	Functions useful for working with regular expressions
string	Useful string and characters functions and exceptions
whrandom	Wichmann-Hill pseudo-random number generator

Unix

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
`thread`	Low-level primitives for working with process threads

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

SUNOS

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
vars()	DO NOT type at interactive prompt! You get infinite loop (C-c will exit).

Python Mode for Emacs

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

The Python Debugger

Accessing

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

`run(string)`
interpret string in the debugger

`runctx(string, globals, locals)`
interpret string using globals and locals for namespace

`runcall(fun, arg1, arg2, ...)`
run function object <fun> with args given.

`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.

`Pdb` defines methods

`run(string)`
interpret string in the debugger

`runctx(string, globals, locals)`
interpret string using globals and locals for namespace

`runcall(fun, arg1, arg2, ...)`
run function object with args runs until a breakpoint hit, exception, or end of program. If an exception occurs, 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 (the 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, same as the exclamation mark (“`!`”) command does.

Example

```
(1394) python
Python 1.0.3 (Sep 26 1994)
>>> 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”.