pox Documentation

Release 0.3.6

Mike McKerns

Jun 03, 2025

CONTENTS

CHAPTER

ONE

POX MODULE DOCUMENTATION

1.1 shutils module

shell utilities for user environment and filesystem exploration

env(*variable*, *all=True*, *minimal=False*)

get dict of environment variables of the form `{variable:value}`

Parameters

- **variable** (*str*) – name or partial name for environment variable.
- **all** (*bool, default=True*) – if False, only return the first match.
- **minimal** (*bool, default=False*) – if True, remove all duplicate paths.

Returns

dict of strings of environment variables.

Warning

selecting all=False can lead to unexpected matches of *variable*.

Examples

```
>>> env('*PATH')
{'PYTHONPATH': '.', 'PATH': '.:/usr/bin:/bin:/usr/sbin:/sbin'}
```

find(*patterns*, *root=None*, *recurse=True*, *type=None*, *verbose=False*)

get the path to a file or directory

Parameters

- **patterns** (*str*) – name or partial name of items to search for.
- **root** (*str, default=None*) – path of top-level directory to search.
- **recurse** (*bool, default=True*) – if True, recurse downward from *root*.
- **type** (*str, default=None*) – a search filter.
- **verbose** (*bool, default=False*) – if True, be verbose about the search.

Returns

a list of string paths.

Notes

on some OS, *recursion* can be specified by recursion depth (*int*), and *patterns* can be specified with basic pattern matching. Also, multiple patterns can be specified by splitting patterns with a `;`. The *type* can be one of `{file, dir, link, socket, block, char}`.

Examples

```
>>> find('pox*', root='..')
['/Users/foo/pox/pox', '/Users/foo/pox/scripts/pox_launcher.py']
>>>
>>> find('*shutils*;*init*')
['/Users/foo/pox/pox/shutils.py', '/Users/foo/pox/pox/__init__.py']
```

homedir()

get the full path of the user's home directory

Parameters
None

Returns
string path of the directory, or None if home can not be determined.

minpath(*path*, *pathsep=None*)

remove duplicate paths from given set of paths

Parameters

- **path** (*str*) – path string (e.g. '/Users/foo/bin:/bin:/sbin:/usr/bin').
- **pathsep** (*str, default=None*) – path separator (e.g. `:`).

Returns
string composed of one or more paths, with duplicates removed.

Examples

```
>>> minpath('.:/Users/foo/bin:.:/Users/foo/bar/bin:/Users/foo/bin')
'.:/Users/foo/bin:/Users/foo/bar/bin'
```

mkdir(*path*, *root=None*, *mode=None*)

create a new directory in the root directory

create a directory at *path* and any necessary parents (i.e. `mkdir -p`). Default mode is read/write/execute for 'user' and 'group', and then read/execute otherwise.

Parameters

- **path** (*str*) – string name of the new directory.
- **root** (*str, default=None*) – path at which to build the new directory.
- **mode** (*str, default=None*) – octal read/write permission [default: 0o775].

Returns
string absolute path for new directory.

rmtree(*path*, *self=True*, *ignore_errors=False*, *onerror=None*)

remove directories in the given path

Parameters

- **path** (*str*) – path string of root of directories to delete.
- **self** (*bool, default=True*) – if False, delete subdirectories, not path.
- **ignore_errors** (*bool, default=False*) – if True, silently ignore errors.
- **onerror** (*function, default=None*) – custom error handler.

Returns
None

Notes

If self=False, the directory indicated by path is left in place, and its subdirectories are erased. If self=True, path is also removed.

If ignore_errors=True, errors are ignored. Otherwise, onerror is called to handle the error with arguments `(func, path, exc_info)`, where *func* is `os.listdir`, `os.remove`, or `os.rmdir`; *path* is the argument to the function that caused it to fail; and *exc_info* is a tuple returned by `sys.exc_info()`. If ignore_errors=False and onerror=None, an exception is raised.

`rootdir()`

get the path corresponding to the root of the current drive

Parameters
None

Returns
string path of the directory.

`sep(type='')`

get the separator string for the given type of separator

Parameters
type (*str, default=''*) – one of `{sep,line,path,ext,alt}`.

Returns
separator string.

`shellsub(command)`

parse the given command to be formatted for remote shell invocation

secure shell (`ssh`) can be used to send and execute commands to remote machines (using `ssh <hostname> <command>`). Additional escape characters are needed to enable the command to be correctly formed and executed remotely. *shellsub* attemps to parse the given command string correctly so that it can be executed remotely with ssh.

Parameters
command (*str*) – the command to be executed remotely.

Returns
the parsed command string.

`shelltype()`

get the name (e.g. `bash`) of the current command shell

Parameters
None

Returns
string name of the shell, or None if name can not be determined.

username()

get the login name of the current user

Parameters
None

Returns
string name of the user.

walk(*root*, *patterns='*'*, *recurse=True*, *folders=False*, *files=True*, *links=True*)

walk directory tree and return a list matching the requested pattern

Parameters

- **root** (*str*) – path of top-level directory to search.
- **patterns** (*str, default='*'*) – (partial) name of items to search for.
- **recurse** (*bool, default=True*) – if True, recurse downward from *root*.
- **folders** (*bool, default=False*) – if True, include folders in the results.
- **files** (*bool, default=True*) – if True, include files in results.
- **links** (*bool, default=True*) – if True, include links in results.

Returns
a list of string paths.

Notes

patterns can be specified with basic pattern matching. Additionally, multiple patterns can be specified by splitting patterns with a `;`.

Examples

```
>>> walk('..', patterns='pox*')
['/Users/foo/pox/pox', '/Users/foo/pox/scripts/pox_launcher.py']
>>>
>>> walk('.', patterns='*shutils*;*init*')
['/Users/foo/pox/pox/shutils.py', '/Users/foo/pox/pox/__init__.py']
```

where(*name*, *path*, *pathsep=None*)

get the full path for the given name string on the given search path.

Parameters

- **name** (*str*) – name of file, folder, etc to find.
- **path** (*str*) – path string (e.g. '/Users/foo/bin:/bin:/sbin:/usr/bin').
- **pathsep** (*str, default=None*) – path separator (e.g. `:`)

Returns
the full path string.

Notes

if pathsep is not provided, the OS default will be used.

whereis(*prog*, *all=False*)

get path to the given program

search the standard binary install locations for the given executable.

Parameters

- **prog** (*str*) – name of an executable to search for (e.g. `python`).
- **all** (*bool, default=True*) – if True, return a list of paths found.

Returns
string path of the executable, or list of path strings.

which(*prog*, *allow_links=True*, *ignore_errors=True*, *all=False*)

get the path of the given program

search the user's paths for the given executable.

Parameters

- **prog** (*str*) – name of an executable to search for (e.g. `python`).
- **allow_links** (*bool, default=True*) – if False, replace link with fullpath.
- **ignore_errors** (*bool, default=True*) – if True, ignore search errors.
- **all** (*bool, default=False*) – if True, get list of paths for executable.

Returns
if all=True, get a list of string paths, else return a string path.

1.2 utils module

higher-level shell utilities for user environment and filesystem exploration

convert(*files*, *platform=None*, *pathsep=None*, *verbose=True*)

convert text files to given platform type

Ensure given files use the appropriate `os.linesep` and other formatting.

Parameters

- **files** (*list(str)*) – a list of filenames.
- **platform** (*str, default=None*) – platform name as in `os.name`.
- **pathsep** (*str, default=None*) – the path separator string.
- **verbose** (*bool, default=True*) – if True, print debug statements..

Returns
0 if converted, otherwise return 1.

disk_used(*path*)

get the disk usage for the given directory

Parameters
path (*str*) – path string.

Returns
int corresponding to disk usage in blocks.

expandvars(*string*, *ref=None*, *secondref={}*)

expand shell variables in string

Expand shell variables of form `$var` and `${var}`. Unknown variables are left unchanged. If a reference dictionary (*ref*) is provided, restrict lookups to *ref*. A second reference dictionary (*secondref*) can also be provided for failover searches. If *ref* is not provided, lookup variables are defined by the user's environment variables.

Parameters

- **string** (*str*) – a string with shell variables.
- **ref** (*dict(str), default=None*) – a dict of lookup variables.
- **secondref** (*dict(str), default={}*) – a failover reference dict.

Returns

string with the selected shell variables substituted.

Examples

```
>>> expandvars('found:: $PYTHONPATH')
'found:: .:/Users/foo/lib/python3.4/site-packages'
>>>
>>> expandvars('found:: $PYTHONPATH', ref={})
'found:: $PYTHONPATH'
```

findpackage(*package*, *root=None*, *all=False*, *verbose=True*, *recurse=True*)

retrieve the path(s) for a package

Parameters

- **package** (*str*) – name of the package to search for.
- **root** (*str, default=None*) – path string of top-level directory to search.
- **all** (*bool, defualt=False*) – if True, return everywhere package is found.
- **verbose** (*bool, default=True*) – if True, print messages about the search.
- **recurse** (*bool, default=True*) – if True, recurse down the root directory.

Returns

string path (or list of paths) where package is found.

Notes

On some OS, recursion can be specified by recursion depth (an integer). `findpackage` will do standard pattern matching for package names, attempting to match the head directory of the distribution.

getvars(*path*, *ref=None*, *sep=None*)

get a dictionary of all variables defined in path

Extract shell variables of form `$var` and `${var}`. Unknown variables will raise an exception. If a reference dictionary (*ref*) is provided, first try the lookup in *ref*. Failover from *ref* will lookup variables defined in the user's environment variables. Use *sep* to override the path separator (`os.sep`).

Parameters

- **path** (*str*) – a path string with shell variables.
- **ref** (*dict(str), default=None*) – a dict of lookup variables.
- **sep** (*str, default=None*) – the path separator string.

Returns
dict of shell variables found in the given path string.

Examples

```
>>> getvars('$HOME/stuff')
{'HOME': '/Users/foo'}
```

index_join(*sequence*, *start*, *stop*, *step=1*, *sequential=True*, *inclusive=True*)

slice a list of strings, then join the remaining strings

If *start* is not found in the sequence, slice from the beginning. If *stop* is not found in the sequence, slice to the end.

Parameters

- **sequence** (*list*) – an ordered sequence of elements.
- **start** (*int*) – index for start of the slice.
- **stop** (*int*) – index for stop position in the sequence.
- **step** (*int, default=1*) – indices until next member of the slice.
- **sequential** (*bool, default=True*) – if True, *start* must preceed *stop*.
- **inclusive** (*bool, default=True*) – if True, include *stop* in the slice.

Returns
string produced by slicing the given sequence and joining the elements.

index_slice(*sequence*, *start*, *stop*, *step=1*, *sequential=False*, *inclusive=False*)

get the slice for a given sequence

Slice indicies are determined by the positions of *start* and *stop*. If *start* is not found in the sequence, slice from the beginning. If *stop* is not found in the sequence, slice to the end.

Parameters

- **sequence** (*list*) – an ordered sequence of elements.
- **start** (*int*) – index for start of the slice.
- **stop** (*int*) – index for stop position in the sequence.
- **step** (*int, default=1*) – indices until next member of the slice.
- **sequential** (*bool, default=False*) – if True, *start* must preceed *stop*.
- **inclusive** (*bool, default=False*) – if True, include *stop* in the slice.

Returns
slice corresponding to given *start*, *stop*, and *step*.

kbytes(*text*)

convert memory text to the corresponding value in kilobytes

Parameters
text (*str*) – string corresponding to an abbreviation of size.

Returns
int representation of text.

Examples

```
>>> kbytes('10K')
10
>>>
>>> kbytes('10G')
10485760
```

parse_remote(*path*, *loopback=False*, *login_flag=False*)

parse remote connection string of the form `[[user@]host:]path`

Parameters

- **path** (*str*) – remote connection string.
- **loopback** (*bool, default=False*) – if True, ensure *host* is used.
- **login_flag** (*bool, default=False*) – if True, prepend user with `-l`.

Returns

a tuple of the form `(user, host, path)`.

Notes

if loopback=True and host=None, then host will be set to localhost.

pattern(*list=[]*, *separator=';'*)

generate a filter pattern from list of strings

Parameters

- **list** (*list(str), default=[]*) – a list of filter elements.
- **separator** (*str, default=';'*) – the separator string.

Returns

a string composed of filter elements joined by the separator.

remote(*path*, *host=None*, *user=None*, *loopback=False*)

build string for a remote connection of the form `[[user@]host:]path`

Parameters

- **path** (*str*) – path string for location of target on (remote) filesystem.
- **host** (*str, default=None*) – string name/ip address of (remote) host.
- **user** (*str, default=None*) – user name on (remote) host.
- **loopback** (*bool, default=False*) – if True, ensure *host* is used.

Returns

a remote connection string.

Notes

if loopback=True and host=None, then host will be set to localhost.

replace(*file*, *sub={}*, *outfile=None*)

make text substitutions given by *sub* in the given file

Parameters

- **file** (*str*) – path to original file.

- **sub** (*dict(str)*) – dict of string replacements {old:new}.
- **outfile** (*str, default=None*) – if given, don't overwrite original file.

Returns
None

Notes

`replace` uses regular expressions, thus a pattern may be used as *old* text. `replace` can fail if order of substitution is important.

select(*iterable*, *counter=''*, *minimum=False*, *reverse=False*, *all=True*)

find items in iterable with the max (or min) count of the given counter.

Find the items in an iterable that have the maximum number of *counter* (e.g. counter='3' counts occurances of '3'). Use `minimum=True` to search for the minimum number of occurances of the *counter*.

Parameters

- **iterable** (*list*) – an iterable of iterables (e.g. lists, strings, etc).
- **counter** (*str, default=''*) – the item to count.
- **minimum** (*bool, default=False*) – if True, find min count (else, max).
- **reverse** (*bool, default=False*) – if True, reverse order of the results.
- **all** (*bool, default=True*) – if False, only return the first result.

Returns
list of items in the iterable with the min (or max) count.

Examples

```
>>> z = ['zero','one','two','three','4','five','six','seven','8','9/81']
>>> select(z, counter='e')
['three', 'seven']
>>> select(z, counter='e', minimum=True)
['two', '4', 'six', '8', '9/81']
>>>
>>> y = [[1,2,3],[4,5,6],[1,3,5]]
>>> select(y, counter=3)
[[1, 2, 3], [1, 3, 5]]
>>> select(y, counter=3, minumim=True, all=False)
[4, 5, 6]
```

selectdict(*dict*, *counter=''*, *minimum=False*, *all=True*)

return a dict of items with the max (or min) count of the given counter.

Get the items from a dict that have the maximum number of the *counter* (e.g. counter='3' counts occurances of '3') in the values. Use `minimum=True` to search for minimum number of occurances of *counter*.

Parameters

- **dict** (*dict*) – dict with iterables as values (e.g. lists, strings, etc).
- **counter** (*str, default=''*) – the item to count.
- **minimum** (*bool, default=False*) – if True, find min count (else, max).
- **all** (*bool, default=True*) – if False, only return the first result.

Returns
dict of items composed of the entries with the min (or max) count.

Examples

```
>>> z = ['zero','one','two','three','4','five','six','seven','8','9/81']
>>> z = dict(enumerate(z))
>>> selectdict(z, counter='e')
{3: 'three', 7: 'seven'}
>>> selectdict(z, counter='e', minimum=True)
{8: '8', 9: '9/81', 2: 'two', 4: '4', 6: 'six'}
>>>
>>> y = {1: [1,2,3], 2: [4,5,6], 3: [1,3,5]}
>>> selectdict(y, counter=3)
{1: [1, 2, 3], 3: [1, 3, 5]}
>>> selectdict(y, counter=3, minumim=True)
{2: [4, 5, 6]}
```

wait_for(*path*, *sleep=1*, *tries=150*, *ignore_errors=False*)

block execution by waiting for a file to appear at the given path

Parameters

- **path** (*str*) – the path string to watch for the file.
- **sleep** (*float, default=1*) – the time between checking results.
- **tries** (*int, default=150*) – the number of times to try.
- **ignore_errors** (*bool, default=False*) – if True, ignore timeout error.

Returns
None

Notes

if the file is not found after the given number of tries, an error will be thrown unless `ignore_error=True`.

using `subproc = Popen(...)` and `subproc.wait()` is usually a better approach. However, when a handle to the subprocess is unavailable, waiting for a file to appear at a given path is a decent last resort.

which_python(*version=False*, *lazy=False*, *fullpath=True*, *ignore_errors=True*)

get the command to launch the selected version of python

`which_python` composes a command string that can be used to launch the desired python executable. The user's path is searched for the executable, unless `lazy=True` and thus only a lazy-evaluating command (e.g. `which python`) is produced.

Parameters

- **version** (*bool, default=False*) – if True, include the version of python.
- **lazy** (*bool, default=False*) – if True, build a lazy-evaluating command.
- **fullpath** (*bool, default=True*) – if True, provide the full path.
- **ignore_errors** (*bool, default=True*) – if True, ignore path search errors.

Returns
string of the implicit or explicit location of the python executable.

Notes

if version is given as an int or float, include the version number in the command string.

if the executable is not found, an error will be thrown unless `ignore_error=True`.

CHAPTER

TWO

POX SCRIPTS DOCUMENTATION

2.1 pox script

run any of the pox commands from the command shell prompt

Notes

- To get help, type `$ pox` at a shell terminal prompt.
- For a list of available functions, type `$ pox "help('pox')"`.
- Incorrect function invocation will print the function's documentation.

Examples:

```
$ pox "which('python')"
/usr/bin/python
```

citation()

print the citation

convert(*files*, *platform=None*, *pathsep=None*, *verbose=True*)

convert text files to given platform type

Ensure given files use the appropriate `os.linesep` and other formatting.

Parameters

- **files** (*list(str)*) – a list of filenames.
- **platform** (*str, default=None*) – platform name as in `os.name`.
- **pathsep** (*str, default=None*) – the path separator string.
- **verbose** (*bool, default=True*) – if True, print debug statements..

Returns

0 if converted, otherwise return 1.

disk_used(*path*)

get the disk usage for the given directory

Parameters

path (*str*) – path string.

Returns

int corresponding to disk usage in blocks.

env(*variable*, *all=True*, *minimal=False*)

get dict of environment variables of the form `{variable:value}`

Parameters

- **variable** (*str*) – name or partial name for environment variable.
- **all** (*bool, default=True*) – if False, only return the first match.
- **minimal** (*bool, default=False*) – if True, remove all duplicate paths.

Returns

dict of strings of environment variables.

Warning

selecting all=False can lead to unexpected matches of *variable*.

Examples

```
>>> env('*PATH')
{'PYTHONPATH': '.', 'PATH': '.:/usr/bin:/bin:/usr/sbin:/sbin'}
```

expandvars(*string*, *ref=None*, *secondref={}*)

expand shell variables in string

Expand shell variables of form `$var` and `${var}`. Unknown variables are left unchanged. If a reference dictionary (*ref*) is provided, restrict lookups to *ref*. A second reference dictionary (*secondref*) can also be provided for failover searches. If *ref* is not provided, lookup variables are defined by the user's environment variables.

Parameters

- **string** (*str*) – a string with shell variables.
- **ref** (*dict(str), default=None*) – a dict of lookup variables.
- **secondref** (*dict(str), default={}*) – a failover reference dict.

Returns

string with the selected shell variables substituted.

Examples

```
>>> expandvars('found:: $PYTHONPATH')
'found:: .:/Users/foo/lib/python3.4/site-packages'
>>>
>>> expandvars('found:: $PYTHONPATH', ref={})
'found:: $PYTHONPATH'
```

find(*patterns*, *root=None*, *recurse=True*, *type=None*, *verbose=False*)

get the path to a file or directory

Parameters

- **patterns** (*str*) – name or partial name of items to search for.
- **root** (*str, default=None*) – path of top-level directory to search.
- **recurse** (*bool, default=True*) – if True, recurse downward from *root*.

- **type** (*str, default=None*) – a search filter.
- **verbose** (*bool, default=False*) – if True, be verbose about the search.

Returns
a list of string paths.

Notes

on some OS, *recursion* can be specified by recursion depth (*int*), and *patterns* can be specified with basic pattern matching. Also, multiple patterns can be specified by splitting patterns with a `;`. The *type* can be one of `{file, dir, link, socket, block, char}`.

Examples

```
>>> find('pox*', root='..')
['/Users/foo/pox/pox', '/Users/foo/pox/scripts/pox_launcher.py']
>>>
>>> find('*shutils*;*init*')
['/Users/foo/pox/pox/shutils.py', '/Users/foo/pox/pox/__init__.py']
```

findpackage(*package*, *root=None*, *all=False*, *verbose=True*, *recurse=True*)

retrieve the path(s) for a package

Parameters

- **package** (*str*) – name of the package to search for.
- **root** (*str, default=None*) – path string of top-level directory to search.
- **all** (*bool, defualt=False*) – if True, return everywhere package is found.
- **verbose** (*bool, default=True*) – if True, print messages about the search.
- **recurse** (*bool, default=True*) – if True, recurse down the root directory.

Returns
string path (or list of paths) where package is found.

Notes

On some OS, recursion can be specified by recursion depth (an integer). `findpackage` will do standard pattern matching for package names, attempting to match the head directory of the distribution.

getvars(*path*, *ref=None*, *sep=None*)

get a dictionary of all variables defined in path

Extract shell variables of form `$var` and `${var}`. Unknown variables will raise an exception. If a reference dictionary (*ref*) is provided, first try the lookup in *ref*. Failover from *ref* will lookup variables defined in the user's environment variables. Use *sep* to override the path separator (`os.sep`).

Parameters

- **path** (*str*) – a path string with shell variables.
- **ref** (*dict(str), default=None*) – a dict of lookup variables.
- **sep** (*str, default=None*) – the path separator string.

Returns
dict of shell variables found in the given path string.

Examples

```
>>> getvars('$HOME/stuff')
{'HOME': '/Users/foo'}
```

help(*function=None*)

homedir()

get the full path of the user's home directory

Parameters
None

Returns
string path of the directory, or None if home can not be determined.

index_join(*sequence*, *start*, *stop*, *step=1*, *sequential=True*, *inclusive=True*)

slice a list of strings, then join the remaining strings

If *start* is not found in the sequence, slice from the beginning. If *stop* is not found in the sequence, slice to the end.

Parameters

- **sequence** (*list*) – an ordered sequence of elements.
- **start** (*int*) – index for start of the slice.
- **stop** (*int*) – index for stop position in the sequence.
- **step** (*int, default=1*) – indices until next member of the slice.
- **sequential** (*bool, default=True*) – if True, *start* must preceed *stop*.
- **inclusive** (*bool, default=True*) – if True, include *stop* in the slice.

Returns
string produced by slicing the given sequence and joining the elements.

index_slice(*sequence*, *start*, *stop*, *step=1*, *sequential=False*, *inclusive=False*)

get the slice for a given sequence

Slice indicies are determined by the positions of *start* and *stop*. If *start* is not found in the sequence, slice from the beginning. If *stop* is not found in the sequence, slice to the end.

Parameters

- **sequence** (*list*) – an ordered sequence of elements.
- **start** (*int*) – index for start of the slice.
- **stop** (*int*) – index for stop position in the sequence.
- **step** (*int, default=1*) – indices until next member of the slice.
- **sequential** (*bool, default=False*) – if True, *start* must preceed *stop*.
- **inclusive** (*bool, default=False*) – if True, include *stop* in the slice.

Returns
slice corresponding to given *start*, *stop*, and *step*.

isfunction(*object*)

Return true if the object is a user-defined function.

Function objects provide these attributes:

__doc__ documentation string __name__ name with which this function was defined __qualname__ qualified name of this function __module__ name of the module the function was defined in or None __code__ code object containing compiled function bytecode __defaults__ tuple of any default values for arguments __globals__ global namespace in which this function was defined __annotations__ dict of parameter annotations __kwdefaults__ dict of keyword only parameters with defaults __dict__ namespace which is supporting arbitrary function attributes __closure__ a tuple of cells or None __type_params__ tuple of type parameters

kbytes(*text*)

convert memory text to the corresponding value in kilobytes

Parameters

text (*str*) – string corresponding to an abbreviation of size.

Returns

int representation of text.

Examples

```
>>> kbytes('10K')
10
>>>
>>> kbytes('10G')
10485760
```

license()

print the license

minpath(*path*, *pathsep=None*)

remove duplicate paths from given set of paths

Parameters

- **path** (*str*) – path string (e.g. '/Users/foo/bin:/bin:/sbin:/usr/bin').
- **pathsep** (*str, default=None*) – path separator (e.g. `:`).

Returns

string composed of one or more paths, with duplicates removed.

Examples

```
>>> minpath('.:/Users/foo/bin:.:/Users/foo/bar/bin:/Users/foo/bin')
'.:/Users/foo/bin:/Users/foo/bar/bin'
```

mkdir(*path*, *root=None*, *mode=None*)

create a new directory in the root directory

create a directory at *path* and any necessary parents (i.e. `mkdir -p`). Default mode is read/write/execute for 'user' and 'group', and then read/execute otherwise.

Parameters

- **path** (*str*) – string name of the new directory.

- **root** (*str, default=None*) – path at which to build the new directory.
- **mode** (*str, default=None*) – octal read/write permission [default: 0o775].

Returns
string absolute path for new directory.

parse_remote(*path*, *loopback=False*, *login_flag=False*)

parse remote connection string of the form `[[user@]host:]path`

Parameters

- **path** (*str*) – remote connection string.
- **loopback** (*bool, default=False*) – if True, ensure *host* is used.
- **login_flag** (*bool, default=False*) – if True, prepend user with `-l`.

Returns
a tuple of the form `(user, host, path)`.

Notes

if loopback=True and host=None, then host will be set to localhost.

pattern(*list=[]*, *separator=';'*)

generate a filter pattern from list of strings

Parameters

- **list** (*list(str), default=[]*) – a list of filter elements.
- **separator** (*str, default=';'*) – the separator string.

Returns
a string composed of filter elements joined by the separator.

remote(*path*, *host=None*, *user=None*, *loopback=False*)

build string for a remote connection of the form `[[user@]host:]path`

Parameters

- **path** (*str*) – path string for location of target on (remote) filesystem.
- **host** (*str, default=None*) – string name/ip address of (remote) host.
- **user** (*str, default=None*) – user name on (remote) host.
- **loopback** (*bool, default=False*) – if True, ensure *host* is used.

Returns
a remote connection string.

Notes

if loopback=True and host=None, then host will be set to localhost.

replace(*file*, *sub={}*, *outfile=None*)

make text substitutions given by *sub* in the given file

Parameters

- **file** (*str*) – path to original file.
- **sub** (*dict(str*) – dict of string replacements `{old:new}`.

- **outfile** (*str, default=None*) – if given, don't overwrite original file.

Returns
None

Notes

`replace` uses regular expressions, thus a pattern may be used as *old* text. `replace` can fail if order of substitution is important.

rmtree(*path*, *self=True*, *ignore_errors=False*, *onerror=None*)

remove directories in the given path

Parameters

- **path** (*str*) – path string of root of directories to delete.
- **self** (*bool, default=True*) – if False, delete subdirectories, not path.
- **ignore_errors** (*bool, default=False*) – if True, silently ignore errors.
- **onerror** (*function, default=None*) – custom error handler.

Returns
None

Notes

If self=False, the directory indicated by path is left in place, and its subdirectories are erased. If self=True, path is also removed.

If ignore_errors=True, errors are ignored. Otherwise, onerror is called to handle the error with arguments `(func, path, exc_info)`, where *func* is `os.listdir`, `os.remove`, or `os.rmdir`; *path* is the argument to the function that caused it to fail; and *exc_info* is a tuple returned by `sys.exc_info()`. If ignore_errors=False and onerror=None, an exception is raised.

rootdir()

get the path corresponding to the root of the current drive

Parameters
None

Returns
string path of the directory.

select(*iterable*, *counter=''*, *minimum=False*, *reverse=False*, *all=True*)

find items in iterable with the max (or min) count of the given counter.

Find the items in an iterable that have the maximum number of *counter* (e.g. counter='3' counts occurances of '3'). Use `minimum=True` to search for the minimum number of occurances of the *counter*.

Parameters

- **iterable** (*list*) – an iterable of iterables (e.g. lists, strings, etc).
- **counter** (*str, default=''*) – the item to count.
- **minimum** (*bool, default=False*) – if True, find min count (else, max).
- **reverse** (*bool, default=False*) – if True, reverse order of the results.
- **all** (*bool, default=True*) – if False, only return the first result.

Returns
list of items in the iterable with the min (or max) count.

Examples

```
>>> z = ['zero','one','two','three','4','five','six','seven','8','9/81']
>>> select(z, counter='e')
['three', 'seven']
>>> select(z, counter='e', minimum=True)
['two', '4', 'six', '8', '9/81']
>>>
>>> y = [[1,2,3],[4,5,6],[1,3,5]]
>>> select(y, counter=3)
[[1, 2, 3], [1, 3, 5]]
>>> select(y, counter=3, minumim=True, all=False)
[4, 5, 6]
```

selectdict(*dict*, *counter=''*, *minimum=False*, *all=True*)

return a dict of items with the max (or min) count of the given counter.

Get the items from a dict that have the maximum number of the *counter* (e.g. counter='3' counts occurances of '3') in the values. Use `minimum=True` to search for minimum number of occurances of *counter*.

Parameters

- **dict** (*dict*) – dict with iterables as values (e.g. lists, strings, etc).
- **counter** (*str, default=''*) – the item to count.
- **minimum** (*bool, default=False*) – if True, find min count (else, max).
- **all** (*bool, default=True*) – if False, only return the first result.

Returns
dict of items composed of the entries with the min (or max) count.

Examples

```
>>> z = ['zero','one','two','three','4','five','six','seven','8','9/81']
>>> z = dict(enumerate(z))
>>> selectdict(z, counter='e')
{3: 'three', 7: 'seven'}
>>> selectdict(z, counter='e', minimum=True)
{8: '8', 9: '9/81', 2: 'two', 4: '4', 6: 'six'}
>>>
>>> y = {1: [1,2,3], 2: [4,5,6], 3: [1,3,5]}
>>> selectdict(y, counter=3)
{1: [1, 2, 3], 3: [1, 3, 5]}
>>> selectdict(y, counter=3, minumim=True)
{2: [4, 5, 6]}
```

sep(*type=''*)

get the separator string for the given type of separator

Parameters
type (*str, default=''*) – one of `{sep,line,path,ext,alt}`.

Returns
separator string.

shellsub(*command*)

parse the given command to be formatted for remote shell invocation

secure shell (ssh) can be used to send and execute commands to remote machines (using ssh <hostname> <command>). Additional escape characters are needed to enable the command to be correctly formed and executed remotely. *shellsub* attemps to parse the given command string correctly so that it can be executed remotely with ssh.

Parameters
command (*str*) – the command to be executed remotely.

Returns
the parsed command string.

shelltype()

get the name (e.g. bash) of the current command shell

Parameters
None

Returns
string name of the shell, or None if name can not be determined.

username()

get the login name of the current user

Parameters
None

Returns
string name of the user.

wait_for(*path*, *sleep=1*, *tries=150*, *ignore_errors=False*)

block execution by waiting for a file to appear at the given path

Parameters

- **path** (*str*) – the path string to watch for the file.
- **sleep** (*float, default=1*) – the time between checking results.
- **tries** (*int, default=150*) – the number of times to try.
- **ignore_errors** (*bool, default=False*) – if True, ignore timeout error.

Returns
None

Notes

if the file is not found after the given number of tries, an error will be thrown unless ignore_error=True.

using subproc = Popen(...) and subproc.wait() is usually a better approach. However, when a handle to the subprocess is unavailable, waiting for a file to appear at a given path is a decent last resort.

walk(*root*, *patterns='*'*, *recurse=True*, *folders=False*, *files=True*, *links=True*)

walk directory tree and return a list matching the requested pattern

Parameters

- **root** (*str*) – path of top-level directory to search.

- **patterns** (*str, default='*'*) – (partial) name of items to search for.
- **recurse** (*bool, default=True*) – if True, recurse downward from *root*.
- **folders** (*bool, default=False*) – if True, include folders in the results.
- **files** (*bool, default=True*) – if True, include files in results.
- **links** (*bool, default=True*) – if True, include links in results.

Returns
a list of string paths.

Notes

patterns can be specified with basic pattern matching. Additionally, multiple patterns can be specified by splitting patterns with a `;`.

Examples

```
>>> walk('..', patterns='pox*')
['/Users/foo/pox/pox', '/Users/foo/pox/scripts/pox_launcher.py']
>>>
>>> walk('.', patterns='*shutils*;*init*')
['/Users/foo/pox/pox/shutils.py', '/Users/foo/pox/pox/__init__.py']
```

where(*name*, *path*, *pathsep=None*)

get the full path for the given name string on the given search path.

Parameters

- **name** (*str*) – name of file, folder, etc to find.
- **path** (*str*) – path string (e.g. '/Users/foo/bin:/bin:/sbin:/usr/bin').
- **pathsep** (*str, default=None*) – path separator (e.g. `:`)

Returns
the full path string.

Notes

if pathsep is not provided, the OS default will be used.

whereis(*prog*, *all=False*)

get path to the given program

search the standard binary install locations for the given executable.

Parameters

- **prog** (*str*) – name of an executable to search for (e.g. `python`).
- **all** (*bool, default=True*) – if True, return a list of paths found.

Returns
string path of the executable, or list of path strings.

which(*prog*, *allow_links=True*, *ignore_errors=True*, *all=False*)

get the path of the given program

search the user's paths for the given executable.

Parameters

- **prog** (*str*) – name of an executable to search for (e.g. `python`).
- **allow_links** (*bool, default=True*) – if False, replace link with fullpath.
- **ignore_errors** (*bool, default=True*) – if True, ignore search errors.
- **all** (*bool, default=False*) – if True, get list of paths for executable.

Returns

if all=True, get a list of string paths, else return a string path.

which_python(*version=False*, *lazy=False*, *fullpath=True*, *ignore_errors=True*)

get the command to launch the selected version of python

`which_python` composes a command string that can be used to launch the desired python executable. The user's path is searched for the executable, unless `lazy=True` and thus only a lazy-evaluating command (e.g. `which python`) is produced.

Parameters

- **version** (*bool, default=False*) – if True, include the version of python.
- **lazy** (*bool, default=False*) – if True, build a lazy-evaluating command.
- **fullpath** (*bool, default=True*) – if True, provide the full path.
- **ignore_errors** (*bool, default=True*) – if True, ignore path search errors.

Returns

string of the implicit or explicit location of the python executable.

Notes

if version is given as an int or float, include the version number in the command string.

if the executable is not found, an error will be thrown unless `ignore_error=True`.

CHAPTER

THREE

POX: UTILITIES FOR FILESYSTEM EXPLORATION AND AUTOMATED BUILDS

3.1 About Pox

`pox` provides a collection of utilities for navigating and manipulating filesystems. This module is designed to facilitate some of the low level operating system interactions that are useful when exploring a filesystem on a remote host, where queries such as *"what is the root of the filesystem?"*, *"what is the user's name?"*, and *"what login shell is preferred?"* become essential in allowing a remote user to function as if they were logged in locally. While `pox` is in the same vein of both the `os` and `shutil` builtin modules, the majority of its functionality is unique and compliments these two modules.

`pox` provides Python equivalents of several unix shell commands such as `which` and `find`. These commands allow automated discovery of what has been installed on an operating system, and where the essential tools are located. This capability is useful not only for exploring remote hosts, but also locally as a helper utility for automated build and installation.

Several high-level operations on files and filesystems are also provided. Examples of which are: finding the location of an installed Python package, determining if and where the source code resides on the filesystem, and determining what version the installed package is.

`pox` also provides utilities to enable the abstraction of commands sent to a remote filesystem. In conjunction with a registry of environment variables and installed utilites, `pox` enables the user to interact with a remote filesystem as if they were logged in locally.

`pox` is part of `pathos`, a Python framework for heterogeneous computing. `pox` is in active development, so any user feedback, bug reports, comments, or suggestions are highly appreciated. A list of issues is located at https://github.com/uqfoundation/pox/issues, with a legacy list maintained at https://uqfoundation.github.io/project/pathos/query.

3.2 Major Features

`pox` provides utilities for discovering the user's environment:

- return the user's name, current shell, and path to user's home directory
- strip duplicate entries from the user's `$PATH`
- lookup and expand environment variables from `${VAR}` to `value`

`pox` also provides utilities for filesystem exploration and manipulation:

- discover the path to a file, exectuable, directory, or symbolic link
- discover the path to an installed package
- parse operating system commands for remote shell invocation

- convert text files to platform-specific formatting

3.3 Current Release

The latest released version of `pox` is available from:

https://pypi.org/project/pox

`pox` is distributed under a 3-clause BSD license.

3.4 Development Version

You can get the latest development version with all the shiny new features at:

https://github.com/uqfoundation

If you have a new contribution, please submit a pull request.

3.5 Installation

`pox` can be installed with `pip`:

```
$ pip install pox
```

3.6 Requirements

`pox` requires:

- `python` (or `pypy`), **>=3.8**
- `setuptools`, **>=42**

3.7 Basic Usage

`pox` includes some basic utilities to connect to and automate exploration on local and remote filesystems. There are some basic functions to discover important locations:

```
>>> import pox
>>> pox.homedir()
'/Users/mmckerns'
>>> pox.rootdir()
'/'
```

or, you can interact with local and global environment variables:

```
>>> local = {'DEV':'${HOME}/dev', 'FOO_VERSION':'0.1', 'BAR_VERSION':'1.0'}
>>> pox.getvars('${DEV}/lib/foo-${FOO_VERSION}', local)
{'DEV': '${HOME}/dev', 'FOO_VERSION': '0.1'}
>>> pox.expandvars('${DEV}/lib/foo-${FOO_VERSION}', local)
'${HOME}/dev/lib/foo-0.1'
>>> pox.expandvars('${HOME}/dev/lib/foo-0.1')
'/Users/mmckerns/dev/lib/foo-0.1'
>>> pox.env('HOME')
{'HOME': '/Users/mmckerns'}
```

and perform some basic search functions:

```
>>> pox.find('python3.9', recurse=5, root='/opt')
['/opt/local/bin/python3.9']
>>> pox.which('python3.9')
'/opt/local/bin/python3.9'
```

pox also has a specialized *which* command just for Python:

```
>>> pox.which_python()
'/opt/local/bin/python3.9'
>>> pox.which_python(lazy=True, version=True)
'`which python3.9`'
```

Any of the pox functions can be launched from the command line, which facilitates executing commands across parallel and distributed pipes (such as *pathos.connection.Pipe* and *pathos.secure.connection.Pipe*):

```
>>> import pathos
>>> p = pathos.connection.Pipe()
>>> p(command="python -m pox 'which_python()'")
>>> p.launch()
>>> print(p.response())
'/usr/bin/python'
>>> p.kill()
```

The functions in pox that help make interactions with filesystems and environment varialbles programmatic and abstract become especially relevant when trying to execute complex commands remotely.

3.8 More Information

Probably the best way to get started is to look at the documentation at http://pox.rtfd.io. Also see `pox.tests` for a set of scripts that demonstrate how pox can be used to interact with the operating system. You can run the test suite with `python -m pox.tests`. All pox utilities can be launched directly from an operating system terminal, using the pox script (or with `python -m pox`). The source code is also generally well documented, so further questions may be resolved by inspecting the code itself. Please feel free to submit a ticket on github, or ask a question on stackoverflow (**@Mike McKerns**). If you would like to share how you use pox in your work, please send an email (to **mmckerns at uqfoundation dot org**).

3.9 Citation

If you use pox to do research that leads to publication, we ask that you acknowledge use of pox by citing the following in your publication:

```
M.M. McKerns, L. Strand, T. Sullivan, A. Fang, M.A.G. Aivazis,
"Building a framework for predictive science", Proceedings of
the 10th Python in Science Conference, 2011;
http://arxiv.org/pdf/1202.1056

Michael McKerns and Michael Aivazis,
"pathos: a framework for heterogeneous computing", 2010- ;
https://uqfoundation.github.io/project/pathos
```

Please see https://uqfoundation.github.io/project/pathos or http://arxiv.org/pdf/1202.1056 for further information.

citation()

print the citation

convert(*files*, *platform=None*, *pathsep=None*, *verbose=True*)

convert text files to given platform type

Ensure given files use the appropriate `os.linesep` and other formatting.

Parameters

- **files** (*list(str)*) – a list of filenames.
- **platform** (*str, default=None*) – platform name as in `os.name`.
- **pathsep** (*str, default=None*) – the path separator string.
- **verbose** (*bool, default=True*) – if True, print debug statements..

Returns

0 if converted, otherwise return 1.

disk_used(*path*)

get the disk usage for the given directory

Parameters

path (*str*) – path string.

Returns

int corresponding to disk usage in blocks.

env(*variable*, *all=True*, *minimal=False*)

get dict of environment variables of the form `{variable:value}`

Parameters

- **variable** (*str*) – name or partial name for environment variable.
- **all** (*bool, default=True*) – if False, only return the first match.
- **minimal** (*bool, default=False*) – if True, remove all duplicate paths.

Returns

dict of strings of environment variables.

Warning

selecting all=False can lead to unexpected matches of *variable*.

Examples

```
>>> env('*PATH')
{'PYTHONPATH': '.', 'PATH': '.:/usr/bin:/bin:/usr/sbin:/sbin'}
```

expandvars(*string*, *ref=None*, *secondref={}*)

expand shell variables in string

Expand shell variables of form `$var` and `${var}`. Unknown variables are left unchanged. If a reference dictionary (*ref*) is provided, restrict lookups to *ref*. A second reference dictionary (*secondref*) can also be provided for failover searches. If *ref* is not provided, lookup variables are defined by the user's environment variables.

Parameters

- **string** (*str*) – a string with shell variables.
- **ref** (*dict(str), default=None*) – a dict of lookup variables.
- **secondref** (*dict(str), default={}*) – a failover reference dict.

Returns
string with the selected shell variables substituted.

Examples

```
>>> expandvars('found:: $PYTHONPATH')
'found:: .:/Users/foo/lib/python3.4/site-packages'
>>>
>>> expandvars('found:: $PYTHONPATH', ref={})
'found:: $PYTHONPATH'
```

find(*patterns*, *root=None*, *recurse=True*, *type=None*, *verbose=False*)

get the path to a file or directory

Parameters

- **patterns** (*str*) – name or partial name of items to search for.
- **root** (*str, default=None*) – path of top-level directory to search.
- **recurse** (*bool, default=True*) – if True, recurse downward from *root*.
- **type** (*str, default=None*) – a search filter.
- **verbose** (*bool, default=False*) – if True, be verbose about the search.

Returns
a list of string paths.

Notes

on some OS, *recursion* can be specified by recursion depth (*int*), and *patterns* can be specified with basic pattern matching. Also, multiple patterns can be specified by splitting patterns with a `;`. The *type* can be one of `{file, dir, link, socket, block, char}`.

Examples

```
>>> find('pox*', root='..')
['/Users/foo/pox/pox', '/Users/foo/pox/scripts/pox_launcher.py']
>>>
>>> find('*shutils*;*init*')
['/Users/foo/pox/pox/shutils.py', '/Users/foo/pox/pox/__init__.py']
```

findpackage(*package*, *root=None*, *all=False*, *verbose=True*, *recurse=True*)

retrieve the path(s) for a package

Parameters

- **package** (*str*) – name of the package to search for.
- **root** (*str, default=None*) – path string of top-level directory to search.
- **all** (*bool, defualt=False*) – if True, return everywhere package is found.
- **verbose** (*bool, default=True*) – if True, print messages about the search.

- **recurse** (*bool, default=True*) – if True, recurse down the root directory.

Returns
string path (or list of paths) where package is found.

Notes

On some OS, recursion can be specified by recursion depth (an integer). `findpackage` will do standard pattern matching for package names, attempting to match the head directory of the distribution.

getvars(*path*, *ref=None*, *sep=None*)

get a dictionary of all variables defined in path

Extract shell variables of form `$var` and `${var}`. Unknown variables will raise an exception. If a reference dictionary (*ref*) is provided, first try the lookup in *ref*. Failover from *ref* will lookup variables defined in the user's environment variables. Use *sep* to override the path separator (`os.sep`).

Parameters

- **path** (*str*) – a path string with shell variables.
- **ref** (*dict(str), default=None*) – a dict of lookup variables.
- **sep** (*str, default=None*) – the path separator string.

Returns
dict of shell variables found in the given path string.

Examples

```
>>> getvars('$HOME/stuff')
{'HOME': '/Users/foo'}
```

homedir()

get the full path of the user's home directory

Parameters
None

Returns
string path of the directory, or None if home can not be determined.

index_join(*sequence*, *start*, *stop*, *step=1*, *sequential=True*, *inclusive=True*)

slice a list of strings, then join the remaining strings

If *start* is not found in the sequence, slice from the beginning. If *stop* is not found in the sequence, slice to the end.

Parameters

- **sequence** (*list*) – an ordered sequence of elements.
- **start** (*int*) – index for start of the slice.
- **stop** (*int*) – index for stop position in the sequence.
- **step** (*int, default=1*) – indices until next member of the slice.
- **sequential** (*bool, default=True*) – if True, *start* must preceed *stop*.
- **inclusive** (*bool, default=True*) – if True, include *stop* in the slice.

Returns
string produced by slicing the given sequence and joining the elements.

index_slice(*sequence*, *start*, *stop*, *step=1*, *sequential=False*, *inclusive=False*)

get the slice for a given sequence

Slice indicies are determined by the positions of *start* and *stop*. If *start* is not found in the sequence, slice from the beginning. If *stop* is not found in the sequence, slice to the end.

Parameters

- **sequence** (*list*) – an ordered sequence of elements.
- **start** (*int*) – index for start of the slice.
- **stop** (*int*) – index for stop position in the sequence.
- **step** (*int, default=1*) – indices until next member of the slice.
- **sequential** (*bool, default=False*) – if True, *start* must preceed *stop*.
- **inclusive** (*bool, default=False*) – if True, include *stop* in the slice.

Returns

slice corresponding to given *start*, *stop*, and *step*.

kbytes(*text*)

convert memory text to the corresponding value in kilobytes

Parameters

text (*str*) – string corresponding to an abbreviation of size.

Returns

int representation of text.

Examples

```
>>> kbytes('10K')
10
>>>
>>> kbytes('10G')
10485760
```

license()

print the license

minpath(*path*, *pathsep=None*)

remove duplicate paths from given set of paths

Parameters

- **path** (*str*) – path string (e.g. '/Users/foo/bin:/bin:/sbin:/usr/bin').
- **pathsep** (*str, default=None*) – path separator (e.g. :).

Returns

string composed of one or more paths, with duplicates removed.

Examples

```
>>> minpath('.:/Users/foo/bin:.:/Users/foo/bar/bin:/Users/foo/bin')
'.:/Users/foo/bin:/Users/foo/bar/bin'
```

mkdir(*path*, *root=None*, *mode=None*)

create a new directory in the root directory

create a directory at *path* and any necessary parents (i.e. `mkdir -p`). Default mode is read/write/execute for 'user' and 'group', and then read/execute otherwise.

Parameters

- **path** (*str*) – string name of the new directory.
- **root** (*str, default=None*) – path at which to build the new directory.
- **mode** (*str, default=None*) – octal read/write permission [default: 0o775].

Returns

string absolute path for new directory.

parse_remote(*path*, *loopback=False*, *login_flag=False*)

parse remote connection string of the form `[[user@]host:]path`

Parameters

- **path** (*str*) – remote connection string.
- **loopback** (*bool, default=False*) – if True, ensure *host* is used.
- **login_flag** (*bool, default=False*) – if True, prepend user with `-l`.

Returns

a tuple of the form `(user, host, path)`.

Notes

if loopback=True and host=None, then host will be set to localhost.

pattern(*list=[]*, *separator=';'*)

generate a filter pattern from list of strings

Parameters

- **list** (*list(str), default=[]*) – a list of filter elements.
- **separator** (*str, default=';'*) – the separator string.

Returns

a string composed of filter elements joined by the separator.

remote(*path*, *host=None*, *user=None*, *loopback=False*)

build string for a remote connection of the form `[[user@]host:]path`

Parameters

- **path** (*str*) – path string for location of target on (remote) filesystem.
- **host** (*str, default=None*) – string name/ip address of (remote) host.
- **user** (*str, default=None*) – user name on (remote) host.
- **loopback** (*bool, default=False*) – if True, ensure *host* is used.

Returns

a remote connection string.

Notes

if loopback=True and host=None, then host will be set to localhost.

replace(*file*, *sub={}*, *outfile=None*)

make text substitutions given by *sub* in the given file

Parameters

- **file** (*str*) – path to original file.
- **sub** (*dict(str*) – dict of string replacements {old:new}.
- **outfile** (*str, default=None*) – if given, don't overwrite original file.

Returns
None

Notes

replace uses regular expressions, thus a pattern may be used as *old* text. replace can fail if order of substitution is important.

rmtree(*path*, *self=True*, *ignore_errors=False*, *onerror=None*)

remove directories in the given path

Parameters

- **path** (*str*) – path string of root of directories to delete.
- **self** (*bool, default=True*) – if False, delete subdirectories, not path.
- **ignore_errors** (*bool, default=False*) – if True, silently ignore errors.
- **onerror** (*function, default=None*) – custom error handler.

Returns
None

Notes

If self=False, the directory indicated by path is left in place, and its subdirectories are erased. If self=True, path is also removed.

If ignore_errors=True, errors are ignored. Otherwise, onerror is called to handle the error with arguments (func, path, exc_info), where *func* is os.listdir, os.remove, or os.rmdir; *path* is the argument to the function that caused it to fail; and *exc_info* is a tuple returned by sys.exc_info(). If ignore_errors=False and onerror=None, an exception is raised.

rootdir()

get the path corresponding to the root of the current drive

Parameters
None

Returns
string path of the directory.

select(*iterable*, *counter=''*, *minimum=False*, *reverse=False*, *all=True*)

find items in iterable with the max (or min) count of the given counter.

Find the items in an iterable that have the maximum number of *counter* (e.g. counter='3' counts occurances of '3'). Use minimum=True to search for the minimum number of occurances of the *counter*.

Parameters

- **iterable** (*list*) – an iterable of iterables (e.g. lists, strings, etc).
- **counter** (*str, default=''*) – the item to count.
- **minimum** (*bool, default=False*) – if True, find min count (else, max).
- **reverse** (*bool, default=False*) – if True, reverse order of the results.
- **all** (*bool, default=True*) – if False, only return the first result.

Returns

list of items in the iterable with the min (or max) count.

Examples

```
>>> z = ['zero','one','two','three','4','five','six','seven','8','9/81']
>>> select(z, counter='e')
['three', 'seven']
>>> select(z, counter='e', minimum=True)
['two', '4', 'six', '8', '9/81']
>>>
>>> y = [[1,2,3],[4,5,6],[1,3,5]]
>>> select(y, counter=3)
[[1, 2, 3], [1, 3, 5]]
>>> select(y, counter=3, minumim=True, all=False)
[4, 5, 6]
```

selectdict(*dict*, *counter=''*, *minimum=False*, *all=True*)

return a dict of items with the max (or min) count of the given counter.

Get the items from a dict that have the maximum number of the *counter* (e.g. counter='3' counts occurances of '3') in the values. Use `minimum=True` to search for minimum number of occurances of *counter*.

Parameters

- **dict** (*dict*) – dict with iterables as values (e.g. lists, strings, etc).
- **counter** (*str, default=''*) – the item to count.
- **minimum** (*bool, default=False*) – if True, find min count (else, max).
- **all** (*bool, default=True*) – if False, only return the first result.

Returns

dict of items composed of the entries with the min (or max) count.

Examples

```
>>> z = ['zero','one','two','three','4','five','six','seven','8','9/81']
>>> z = dict(enumerate(z))
>>> selectdict(z, counter='e')
{3: 'three', 7: 'seven'}
>>> selectdict(z, counter='e', minimum=True)
{8: '8', 9: '9/81', 2: 'two', 4: '4', 6: 'six'}
>>>
>>> y = {1: [1,2,3], 2: [4,5,6], 3: [1,3,5]}
>>> selectdict(y, counter=3)
```

(continues on next page)

(continued from previous page)

```
{1: [1, 2, 3], 3: [1, 3, 5]}
>>> selectdict(y, counter=3, minumim=True)
{2: [4, 5, 6]}
```

sep(*type=''*)

get the separator string for the given type of separator

Parameters
type (*str, default=''*) – one of {sep,line,path,ext,alt}.

Returns
separator string.

shellsub(*command*)

parse the given command to be formatted for remote shell invocation

secure shell (ssh) can be used to send and execute commands to remote machines (using ssh <hostname> <command>). Additional escape characters are needed to enable the command to be correctly formed and executed remotely. *shellsub* attemps to parse the given command string correctly so that it can be executed remotely with ssh.

Parameters
command (*str*) – the command to be executed remotely.

Returns
the parsed command string.

shelltype()

get the name (e.g. bash) of the current command shell

Parameters
None

Returns
string name of the shell, or None if name can not be determined.

username()

get the login name of the current user

Parameters
None

Returns
string name of the user.

wait_for(*path*, *sleep=1*, *tries=150*, *ignore_errors=False*)

block execution by waiting for a file to appear at the given path

Parameters

- **path** (*str*) – the path string to watch for the file.
- **sleep** (*float, default=1*) – the time between checking results.
- **tries** (*int, default=150*) – the number of times to try.
- **ignore_errors** (*bool, default=False*) – if True, ignore timeout error.

Returns
None

Notes

if the file is not found after the given number of tries, an error will be thrown unless `ignore_error=True`.

using `subproc = Popen(...)` and `subproc.wait()` is usually a better approach. However, when a handle to the subprocess is unavailable, waiting for a file to appear at a given path is a decent last resort.

walk(*root*, *patterns='*'*, *recurse=True*, *folders=False*, *files=True*, *links=True*)

walk directory tree and return a list matching the requested pattern

Parameters

- **root** (*str*) – path of top-level directory to search.
- **patterns** (*str, default='*'*) – (partial) name of items to search for.
- **recurse** (*bool, default=True*) – if True, recurse downward from *root*.
- **folders** (*bool, default=False*) – if True, include folders in the results.
- **files** (*bool, default=True*) – if True, include files in results.
- **links** (*bool, default=True*) – if True, include links in results.

Returns

a list of string paths.

Notes

patterns can be specified with basic pattern matching. Additionally, multiple patterns can be specified by splitting patterns with a `;`.

Examples

```
>>> walk('..', patterns='pox*')
['/Users/foo/pox/pox', '/Users/foo/pox/scripts/pox_launcher.py']
>>>
>>> walk('.', patterns='*shutils*;*init*')
['/Users/foo/pox/pox/shutils.py', '/Users/foo/pox/pox/__init__.py']
```

where(*name*, *path*, *pathsep=None*)

get the full path for the given name string on the given search path.

Parameters

- **name** (*str*) – name of file, folder, etc to find.
- **path** (*str*) – path string (e.g. '/Users/foo/bin:/bin:/sbin:/usr/bin').
- **pathsep** (*str, default=None*) – path separator (e.g. `:`)

Returns

the full path string.

Notes

if pathsep is not provided, the OS default will be used.

whereis(*prog*, *all=False*)

get path to the given program

search the standard binary install locations for the given executable.

Parameters

- **prog** (*str*) – name of an executable to search for (e.g. `python`).
- **all** (*bool, default=True*) – if True, return a list of paths found.

Returns

string path of the executable, or list of path strings.

which(*prog*, *allow_links=True*, *ignore_errors=True*, *all=False*)

get the path of the given program

search the user's paths for the given executable.

Parameters

- **prog** (*str*) – name of an executable to search for (e.g. `python`).
- **allow_links** (*bool, default=True*) – if False, replace link with fullpath.
- **ignore_errors** (*bool, default=True*) – if True, ignore search errors.
- **all** (*bool, default=False*) – if True, get list of paths for executable.

Returns

if all=True, get a list of string paths, else return a string path.

which_python(*version=False*, *lazy=False*, *fullpath=True*, *ignore_errors=True*)

get the command to launch the selected version of python

`which_python` composes a command string that can be used to launch the desired python executable. The user's path is searched for the executable, unless `lazy=True` and thus only a lazy-evaluating command (e.g. `which python`) is produced.

Parameters

- **version** (*bool, default=False*) – if True, include the version of python.
- **lazy** (*bool, default=False*) – if True, build a lazy-evaluating command.
- **fullpath** (*bool, default=True*) – if True, provide the full path.
- **ignore_errors** (*bool, default=True*) – if True, ignore path search errors.

Returns

string of the implicit or explicit location of the python executable.

Notes

if version is given as an int or float, include the version number in the command string.

if the executable is not found, an error will be thrown unless `ignore_error=True`.

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX

Symbols

C

D

E

F

G

H

I

K

L

M

P

R

S

U

W